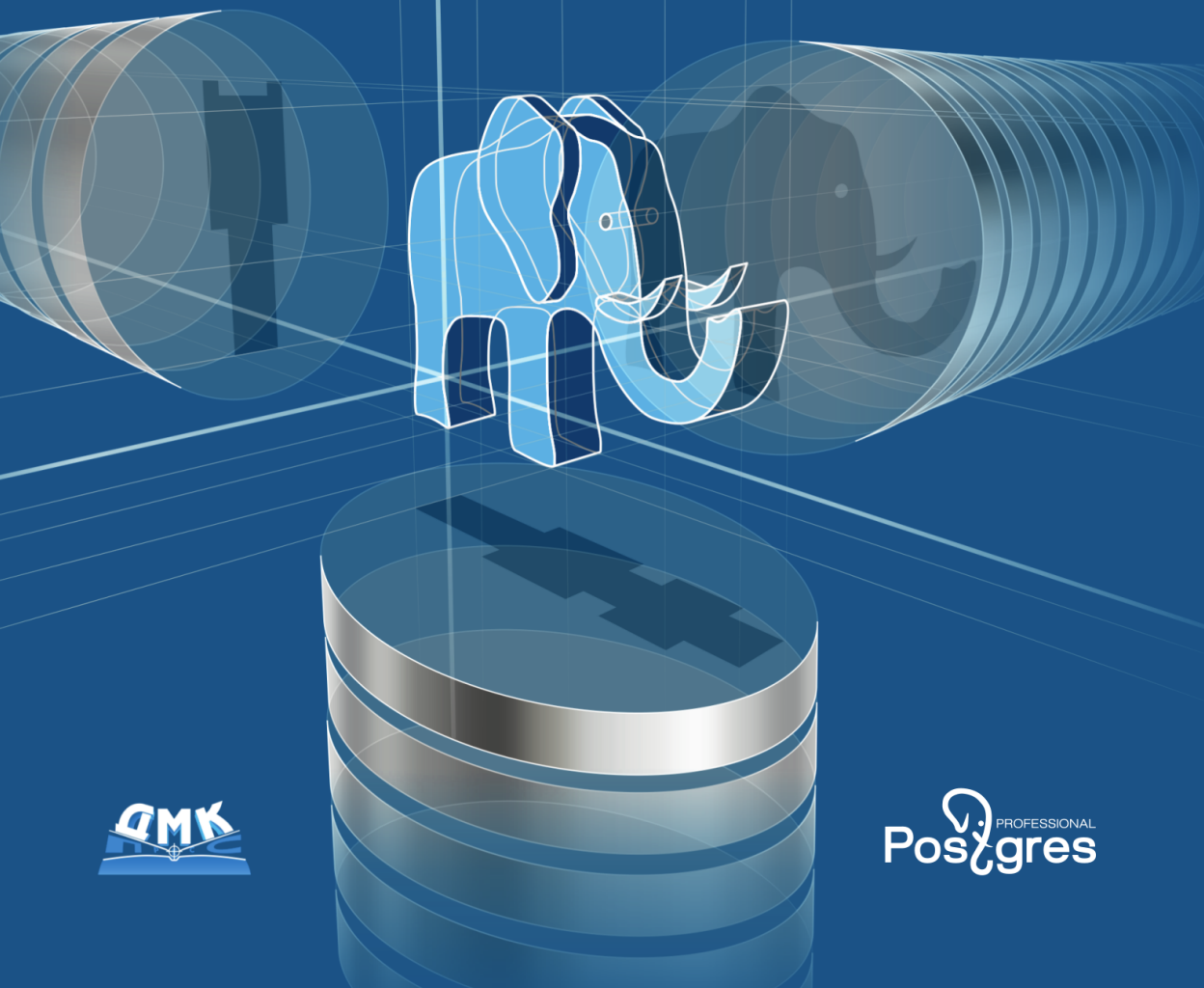
Б. А. Новиков, Е. А. Горшкова

ОСНОВЫ ТЕХНОЛОГИЙ БАЗ ДАННЫХ



ДМК Пресс
PROFESSIONAL Postgres

Компания Postgres Professional

Б. А. Новиков,
Е. А. Горшкова

Основы технологий баз данных

ДМК ПРЕСС
издательство
Москва, 2019

УДК 004.65
ББК 32.972.134
Н73

Новиков Б. А.

Н73 Основы технологий баз данных: учеб. пособие / Б. А. Новиков, Е. А. Горшкова; под ред. Е. В. Рогова. — М.: ДМК Пресс, 2019. — 240 с.

ISBN 978-5-94074-820-5

Представлены основы теории баз данных, методы и алгоритмы, применяемые при реализации систем управления базами данных, а также особенности этих методов и алгоритмов, реализованные в СУБД PostgreSQL.

Материал книги составляет основу для базового учебного курса и содержит краткий обзор требований и критериев оценки СУБД и баз данных, теоретическую реляционную модель данных, основные конструкции языка запросов SQL, организацию доступа к базе данных PostgreSQL, вопросы проектирования приложений и основные расширения, доступные в системе PostgreSQL.

Сайт книги: https://postgrespro.ru/education/books/dbtech.

Для программистов и студентов

УДК 004.655
ББК 32.973.134

ISBN 978-5-94074-820-5
ISBN 978-5-6041193-3-4

Оглавление

О курсе

На кого ориентирован курс

Курс рассчитан на студентов младших курсов (бакалавриата) классических и технических университетов, а также других вузов, имеющих базовую подготовку по программированию и продолжающих специализироваться в областях, близких к программированию.

Какие знания будут получены

В курсе подробно рассматриваются основные понятия, устройство и принципы работы СУБД, а также технологии (архитектура, алгоритмы, структуры данных), лежащие в их основе.

Прослушавшие курс получат уверенные знания и практические навыки по следующим вопросам:

- устройство и принципы работы СУБД;
- проектирование баз данных;
- работа с SQL — составление и оптимизация запросов;
- разработка серверных приложений;
- использование различных типов индексов;
- обработка транзакций и одновременный доступ;
- основы эксплуатации баз данных;
- обеспечение надежности хранения, отказоустойчивости и высокой доступности;
- принципы организации и работы параллельных и распределенных СУБД;
- работа со слабоструктурированными данными (JSON, XML).

Такая подготовка позволит на старших курсах (в магистратуре) специализироваться на разработке и настройке приложений баз данных либо в областях проектирования и разработки СУБД.

Структура курса

Курс состоит из двух частей.

В первой части, представленной в этой книге, рассматриваются основные сведения о базах данных и системах управления базами данных: реляционная модель данных, язык SQL, обработка транзакций.

Во второй части подробно рассматриваются технологии, лежащие в основе функционирования СУБД, а также современные направления и тенденции развития СУБД, основные аспекты их практического применения. При этом некоторые темы, рассмотренные в первой части, изучаются повторно на более глубоком уровне.

Курс в основном касается классических реляционных и объектно-реляционных СУБД, но затрагивает также тематику неклассических СУБД.

Практические занятия не только помогают закрепить пройденный на лекциях материал. Они содержат много дополнительной информации, закрепляющей и расширяющей знания, изложенные в теоретической части. В качестве СУБД для практических занятий используется PostgreSQL.

Как первая, так и вторая части курса могут быть выделены в самостоятельные курсы. Отдельные разделы курса могут быть скомбинированы так, чтобы получить более практическую или более фундаментальную направленность либо адаптировать курс к конкретному учебному плану вуза.

Программные средства, используемые в курсе

Для эффективного освоения материала курса и для выполнения упражнений необходимо установить на компьютере ряд программных продуктов. Набор этих продуктов может зависеть от используемой операционной системы и от других обстоятельств, но в любом случае понадобятся:

- система управления базами данных PostgreSQL;

- демонстрационная база данных, которая используется в большинстве примеров;
- текстовый редактор для подготовки запросов на языке SQL.

Установка PostgreSQL и демобазы рассматривается в главе 3.

Для выполнения упражнений по созданию и редактированию моделей данных может потребоваться инструмент для редактирования диаграмм.

Для разработки приложений на императивных языках программирования (C, C++, Java, Python и др.) потребуются соответствующие среды разработки, однако такие упражнения не включены в состав этого курса.

При работе с PostgreSQL можно использовать ряд приложений, предоставляющих графические интерфейсы для работы с базами данных. Многие из этих приложений могут работать с различными СУБД. Как правило, в таких системах ограничены возможности работы с особенностями любой конкретной СУБД. При освоении материала этого курса более целесообразно использовать непосредственно средства системы PostgreSQL или другие программы, спроектированные специально для работы с PostgreSQL.

Благодарности

Подготовка этого курса была бы невозможна без активной поддержки со стороны компании Postgres Professional и ее руководства, в частности О. Бартунова и И. Панченко. Качество материала существенно улучшилось благодаря усилиям Е. Рогова, взявшего на себя огромный труд по редактированию курса.

Главы 7 и 9 написаны совместно Е. Горшковой и Б. Новиковым. В подготовке упражнений принимали участие В. Бусаров, К. Секереш, Г. Шалыгина и Е. Михайлова.

Глава 1

Введение

Компьютеры используются повсеместно: невозможно найти предприятие или учреждение, которые не использовали бы их для решения производственных или управленческих задач. Подобные высказывания не слишком заметны в средствах массовой информации потому, что они уже давно не являются новостью. Профессионалы, однако, понимают, что на самом деле важны не компьютеры, а информационные системы, которые на них работают, а в центре любой информационной системы находятся данные.

Эта книга о том, как хранить данные, обеспечивать их корректность и сохранность и как их обрабатывать эффективно.

1.1. Базы данных и СУБД

Появление и относительно широкое распространение в начале 60-х годов XX века запоминающих устройств достаточно большой емкости с возможностью доступа к произвольным участкам памяти — магнитных дисков, — открыло широкие возможности для создания сложных структур долговременно хранимых данных. Высокая скорость обновления небольших объемов данных (доли секунды) создала условия для создания приложений, способных функционировать в режиме оперативной работы (on-line). В отличие от систем предшествующих поколений, время ответа стало измеряться не сутками, а секундами или долями секунды.

Эти возможности, однако, привели к существенному усложнению кода приложений и, как следствие, к удорожанию их разработки и снижению надежности. В связи с этим появилась идея централизации функций управления данными, которая привела к появлению систем, предоставляющих приложениям услуги по обработке данных. Такие системы получили название систем управления базами данных (СУБД).

Поскольку СУБД используются многими приложениями, ожидается, что они могут обеспечивать более высокие значения эксплуатационных характеристик, таких как надежность хранения и эффективность обработки, недостижимые при индивидуальной разработке средств управления данными для каждого приложения.

Важно отметить, что многие особенности и характеристики, присущие ранним системам управления базами данных, связаны с требованиями тех областей применения и классов приложений, которые были наиболее актуальны в то время. В первую очередь это приложения, работающие в режиме оперативной обработки (on-line transaction processing, OLTP) в банковской и финансовой сферах.

Прежде чем обсуждать, каким образом эти области применения повлияли на характеристики СУБД, уточним значение некоторых терминов, которые будут использоваться в дальнейшем.

Система управления базами данных (СУБД) — это программный комплекс, обеспечивающий централизованное хранение данных и предоставляющий приложениям услуги по обработке данных.

Совокупность данных, хранимых под управлением СУБД, называется *базой данных*. В оригинальном английском варианте словосочетание data base означает «основание, состоящее из данных». Этот смысл несколько искажается в русском словосочетании «база данных». На самом деле это — фундамент, на котором строятся приложения и который состоит из данных. Действительно, данные (а следовательно, база данных) являются очень существенной частью практически любой информационной системы.

Система управления базами данных, находящаяся в фазе выполнения, связанная с некоторой конкретной базой данных и готовая выполнять запросы на обработку этой базы данных, называется *экземпляром* (instance) или *сервером базы данных*. На самом деле экземпляр и сервер — разные понятия: один сервер баз данных может управлять несколькими экземплярами баз данных, однако это различие станет важным только начиная с главы 5.

1.2. Требования к СУБД

Ранние системы управления данными очень сильно различались как по своей внутренней организации, так и по предоставляемым возможностям. Потребо-

валось несколько лет, для того чтобы определить, каковы основные функции систем управления базами данных и какие требования следует предъявлять к таким системам.

Основные требования к системам управления базами данных были сформулированы в документе, опубликованном в 1971 году комитетом по системам и языкам обработки данных (CODASYL) [13], русский перевод которого издан в 1975 году [17]. Основой для этих требований послужил анализ систем, применявшихся в период подготовки отчета, и особенностей прикладных областей, в которых эти системы использовались.

В дальнейшем круг областей применения СУБД непрерывно расширялся, появлялись новые системы и уходили старые, однако многие из этих требований остались актуальными и сегодня, и большинство современных СУБД в той или иной форме реализует значительную часть этих требований. Однако далеко не все классы приложений, в которых используются современные системы, предъявляют те же требования к обработке данных, поэтому и системы реализуются иначе.

Приложения, относящиеся к классу оперативной обработки (OLTP), характеризуются тем, что:

- каждое выполнение приложения занимает мало времени (в идеале — не больше долей секунды);
- данные используются совместно многими приложениями;
- при каждом выполнении приложение использует ничтожную долю общего объема хранимых данных, и обычно количество используемых данных не зависит от общего объема базы.

Важно также отметить, что процессы обработки данных и структуры данных в тех областях, в которых использовались ранние СУБД, фактически были формализованы задолго до появления электронных вычислительных систем. Так, правила бухгалтерского учета в основном сложились в XIV веке и мало изменились в последующем. Возможно, это привело к тому, что СУБД, как правило, ориентированы на обработку структурированных данных.

Перечислим основные требования к системам управления базами данных.

Разделение программ и данных. Описание структуры данных должно быть отделено от кода приложений, и система должна допускать независимое

изменение структуры данных и кода приложения. В документе [17] использовался термин «независимость» (independence), однако «разделение» лучше отражает существо этого требования.

Высокоуровневый язык запросов. Система должна предоставлять средства для обработки данных, не включенные в какое-либо приложение.

Целостность. Система должна предотвращать запись данных, нарушающих заранее специфицированные ограничения.

Согласованность. Система должна предотвращать появление некорректностей в данных вследствие параллельной или псевдопараллельной работы нескольких приложений.

Отказоустойчивость. СУБД не должна допускать потери данных даже в случае отказов оборудования.

Защита и разграничение доступа. Система должна предотвращать несанкционированный доступ к данным и предоставлять каждому пользователю (или приложению) доступ только к части данных в соответствии с правами этого пользователя.

Заметим, что в ранние годы существования СУБД предполагалось, что все данные, необходимые для информационных систем предприятия, будут храниться в единой базе данных. Безусловно, это очень хорошая идея, поскольку при этом создаются возможности для исключения избыточности данных, проверки их корректности и предотвращается рассогласованность данных, используемых различными подразделениями предприятия. Почти все учебники по базам данных написаны с учетом этого предположения.

Однако на практике это никогда не было реализовано: как правило, для каждого приложения или небольшой группы приложений создается отдельная база данных. Основная причина, по-видимому, состоит в том, что структуры данных, необходимые для обеспечения информационных потребностей предприятия, слишком сложны для того, чтобы их можно было описать в рамках одной базы данных. Это обстоятельство существенно влияет на значение и использование как перечисленных требований, так и других особенностей систем управления базами данных.

Далее в этой главе данные требования и возможности СУБД обсуждаются более детально, а в последующих главах показано, как эти возможности реализуются в современных системах и, в частности, в системе PostgreSQL.

1.3. Разделение данных и программ

Каждая система управления данными создает некоторый уровень абстракции для других программных компонент, которые используют ее услуги. Для того чтобы реализовать определенный уровень абстракции данных, необходимо, чтобы система могла работать с описанием данных, соответствующих требуемому уровню абстракции.

Так, современные файловые системы представляют файлы как последовательности байтов. Соответственно, операции над содержимым файлов выражаются в терминах позиций байтов внутри файла. Никакие более сложные операции не могут быть определены, потому что файловая система не имеет информации о структуре данных внутри файла.

Поскольку ожидается, что системы управления базами данных предоставляют операции над данными сложной структуры, необходимо, чтобы описание этой структуры было доступно СУБД и было бы общим для всех программ (приложений), использующих эти данные. Это приводит к идее отделения описания структуры данных от программ. Такое описание хранится в самой базе данных и называется *схемой базы данных*.

Для определения схем используются языки описания данных. Чем больше возможностей у такого языка, тем больше услуг может предоставить система управления базами данных. В то же время необходимость подготовки детализированного описания данных на фазе проектирования прикладной системы может в некоторых случаях оказаться чрезмерно трудоемкой.

Само по себе отделение описания данных от приложений не дает достаточной гибкости. Для того чтобы в полной мере реализовать идею разделения данных и программ, в 1975 году (тем же комитетом CODASYL) была подготовлена обобщенная модель языка описания данных. Описание этой модели стало известно под названием «модель данных ANSI/SPARC», так как предполагалось, что эта модель будет иметь статус стандарта. Схематически основные компоненты этой модели представлены на рис. 1.3.1.

Модель включает:

внешнюю схему, содержащую описание данных в таком виде, в котором они будут использоваться приложением (отдельно для каждого приложения), а также отображение логической структуры данных во внешнюю схему;

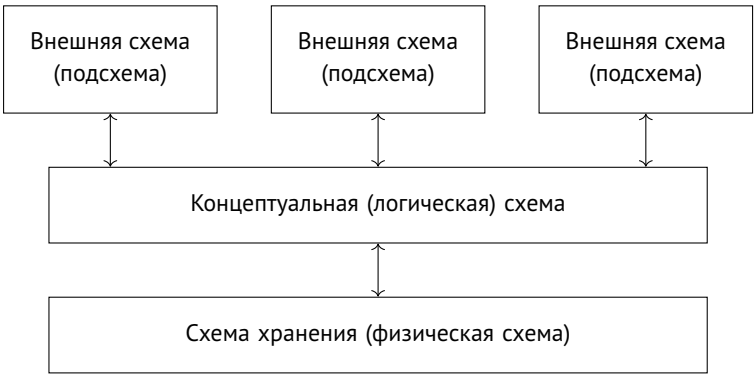


Рис. 1.3.1. Трехуровневая модель данных ANSI/SPARC

концептуальную схему, содержащую полное описание логической структуры данных, доступное СУБД (этот уровень описания было бы правильнее называть логической схемой базы данных);

схему хранения, описывающую, как организовано хранение логических структур данных.

Фактически эта модель никогда не была полностью реализована ни в одной системе, однако ее удобно использовать для того, чтобы определять назначение тех или иных составляющих языка описания данных.

В идеале, трехуровневая модель обеспечивает возможности относительно независимой эволюции приложений и системы в целом. Так, при появлении новых приложений, использующих те же данные, достаточно определить новую внешнюю схему. В результате внедрение нового приложения не повлияет на работу других приложений.

Если новая версия приложения использует дополнительные элементы или структуры данных, достаточно определить новую внешнюю схему. Тогда старая и новая версии приложения смогут сосуществовать, что существенно упрощает постепенный и безопасный переход на новую версию.

Если для работы нового приложения требуются дополнительные структуры данных, эти структуры могут быть добавлены в концептуальную схему, что теоретически не повлияет на работу других приложений, так как их внешние схемы не будут содержать новых элементов данных.

Конечно, такая идеальная картина не всегда может реализоваться. Далеко не все изменения логической схемы могут остаться незаметными даже для приложений, которым не нужны измененные части, потому что изменения могут влиять не только на элементы данных, но и на взаимосвязи между ними.

В реальности эти потенциальные возможности могут реализоваться только в том случае, если разработчики как базы данных, так и приложений их тщательно учитывают. Эволюция базы данных имеет смысл в тех случаях, когда ценность накопленных данных высока. В других случаях создание новой базы данных может оказаться более оправданным решением. Далее в данном курсе будут рассмотрены примеры, иллюстрирующие оба подхода.

Развитые СУБД, в том числе PostgreSQL, предоставляют большое разнообразие методов хранения и поиска данных. Выбор этих методов влияет на производительность приложений, но не влияет на логическую структуру и на результаты выполняемых операций. Поэтому изменение структуры хранения можно использовать для того, чтобы повлиять на характеристики производительности отдельных приложений и системы в целом. Процесс внесения изменений, направленных на улучшение характеристик производительности и не затрагивающих логику работы приложений, называется *настройкой*.

В последние годы значение разделения описаний данных и программ зачастую недооценивается. Многие методологии разработки приложений предполагают генерацию схемы базы данных на основе объектной модели приложения. В частности, это характерно для методологий быстрой разработки прототипов, не предполагающих проведения тщательного предварительного анализа предметной области. Конечно, сложность при этом не исчезает, и задачи, связанные с проектированием схемы базы данных, все равно приходится решать, даже если они не выделены как отдельная единица работы.

Особенности проектирования схемы базы данных существенно зависят от применяемой модели данных. Различные типы моделей обсуждаются в главе 2. Здесь мы только отметим, что сложность проектирования может по-разному распределяться между базой данных и приложением: чем беднее модель данных, тем больше требуется делать на уровне приложения.

Наконец, заметим, что в некоторых подходах описание данных не оформляется как отдельная единица хранения (схема). Вместо этого описание данных может храниться вместе с самими данными (например, при использовании XML или JSON).

1.4. Языки запросов

Наличие описания логической структуры данных открывает возможности для выполнения достаточно сложных операций манипулирования данными внутри СУБД. Такие операции записываются на *языке запросов*.

Входящие в состав современных СУБД языки запросов являются декларативными, то есть позволяют описывать требуемый результат вычислений, но не способ выполнения этих вычислений. Благодаря этому СУБД может выбрать наиболее эффективные (по некоторому критерию производительности) алгоритмы получения результата. Это оказывается наиболее полезным при массовой обработке данных, так как, с одной стороны, позволяет исключить передачу промежуточных результатов между сервером базы данных и приложением и, с другой стороны, выбрать оптимальный способ выполнения вычислений с учетом характеристик фактически хранимых данных, что недостижимо при программировании эквивалентных операций в коде приложения.

Мощные средства обработки декларативных запросов, предоставляемые современными СУБД, в том числе PostgreSQL, зачастую используются недостаточно. Это происходит по многим причинам, среди которых можно выделить плохую совместимость декларативных средств языков запросов с императивными средствами массово применяемых языков программирования.

Практика разработки приложений без использования возможностей языков запросов привела к появлению ряда систем, не предоставляющих такие средства (так называемые NoSQL-системы). При использовании подобных систем для хранения данных, очевидно, часть функций СУБД переносится в приложение. Потенциально это может приводить к увеличению сложности приложения и стоимости его разработки либо к снижению качества — что во многих случаях оказывается допустимым.

1.5. Целостность и согласованность

В состав логической схемы базы данных могут включаться не только описания структур данных и зависимостей между ними, но и дополнительные условия, которым хранимые данные обязательно должны удовлетворять. Такие условия называются *ограничениями целостности* (integrity constraints). Система управления базами данных проверяет ограничения целостности при выполнении

любых изменений хранимых данных и не допускает выполнения операций, нарушающих эти ограничения.

Поддержка ограничений целостности на уровне СУБД позволяет существенно упростить разработку приложений и одновременно улучшить их качество, так как исключает необходимость обработки некорректных данных. Такие данные просто не могут быть записаны в базу данных и, следовательно, никогда не будут выданы в качестве ответа на запрос приложения.

В качестве ограничений целостности можно задавать только условия, которые не могут нарушаться ни при каких обстоятельствах. Существуют, однако, условия корректности другого типа, обычно связывающие значения нескольких элементов данных. В качестве примера таких условий чаще всего приводится правило, согласно которому суммарный баланс при переводе средств с одного бухгалтерского счета на другой не может измениться. Такие условия могут нарушаться для отдельных операций, однако удовлетворяются для набора из нескольких операций.

Состояния базы данных, в которых выполняются подобные условия, называются *согласованными* (consistent), а сами правила — условиями согласованности. Конечный набор операций, который переводит согласованное состояние в другое согласованное, называется *транзакцией*. Несмотря на то что каждое приложение обеспечивает согласованность при выполнении своих транзакций, при неконтролируемом параллельном или псевдопараллельном выполнении нескольких транзакций согласованность может нарушаться. Одной из важных функций СУБД является предотвращение нарушений согласованности при одновременной работе многих приложений (или одного и того же приложения от имени разных пользователей).

Различие между целостностью и согласованностью можно пояснить следующим образом. Ограничения целостности описываются условиями в базе данных, и СУБД отвечает за то, чтобы эти ограничения выполнялись. Условия согласованности определяются приложением и не могут быть проверены СУБД, однако она гарантирует, что результаты выполнения приложения (транзакции) не будут зависеть от факторов, находящихся вне контроля приложения, в том числе от работы других приложений, сбоев и отказов вычислительной системы.

Литература по базам данных изобилует упрощенными примерами из финансовой области приложений, однако ни в коем случае не следует отождествлять понятия транзакции в базах данных с банковскими транзакциями или другими транзакциями в смысле прикладных предметных областей. В реальности

даже самые простые банковские транзакции реализуются в информационных системах как комбинации из нескольких транзакций в базах данных.

В русскоязычной литературе зачастую термин *consistency* переводится как «целостность», что приводит к путанице, так как термин *integrity* переводится точно так же. В этой книге слово *целостность* всегда относится к ограничениям целостности (integrity constraints), а термин *согласованность* обозначает понятие, выражаемое термином *consistency*.

1.6. Отказоустойчивость

В наши дни информационные системы используются повсеместно: едва ли найдется предприятие или организация, в которой они бы не применялись. Во многих случаях работа информационной системы стала жизненно важной для основных производственных процессов, то есть отказы информационной системы приводят к остановке бизнес-процессов, а потеря данных приводит к катастрофическим последствиям (зачастую не только для производственных функций, но и для жизни людей или состояния окружающей среды).

Поэтому при разработке систем управления базами данных с самого начала очень большое внимание уделялось средствам, обеспечивающим отказоустойчивость данных и их выживаемость. В результате длительного развития технологий, связанных с базами данных, эта цель была достигнута.

Современные системы при соответствующей конфигурации могут гарантировать полную сохранность данных и восстановление после отказов оборудования в корректном (согласованном) состоянии. При необходимости система может быть организована таким образом, чтобы восстановление занимало доли секунды. Другими словами, СУБД способны обеспечить значительно более высокую надежность и доступность данных, чем надежность или доступность оборудования, на котором эти данные хранятся и на котором работают эти системы.

Однако создание высоконадежных систем неизбежно оказывается весьма дорогостоящим. Это связано с необходимостью многократного дублирования используемых средств на всех уровнях, начиная от оборудования, что приводит к существенному усложнению системы. В случае использования внешних сервисов, например при размещении данных в облачной среде, необходимо иметь запасные ресурсы, способные обеспечить выживание при отказе этой среды,

даже если она позиционируется как высоконадежная. Поэтому при проектировании системы следует выбрать такой уровень защищенности от отказов, который для нее действительно необходим.

Технологических ограничений, которые не позволяли бы в достаточной мере защитить данные, не существует; все случаи, когда потеря данных происходила, связаны с недооценкой рисков при проектировании системы.

1.7. Безопасность и разграничение доступа

Данные нуждаются в защите не только от отказов оборудования и природных явлений, но и от несанкционированного доступа. Все развитые системы содержат средства как для предотвращения доступа к базе данных от имени лиц, не имеющих на это права, так и для разграничения доступа к данным тех, кто такое право имеет. Допускается обработка (чтение или модификация) только тех данных, для которых соответствующие операции разрешены лицу, от имени которого они выполняются.

Подобные средства защиты реализуются не только на уровне СУБД: защитой приходится заниматься практически на всех уровнях и во всех компонентах информационной системы. Во многих простых системах средства защиты, предоставляемые СУБД, вообще не используются. Однако по-настоящему надежная защита должна быть многоуровневой, а некоторые виды защиты вообще невозможно реализовать без использования СУБД.

1.8. Производительность

Сравнение различных систем управления базами данных и оценка их применимости невозможны без учета их производительности. Для того чтобы такой учет был по возможности объективным, необходимы количественные метрики для измерения производительности.

Наиболее важными интегральными (то есть учитывающими несколько различных факторов) метриками являются *пропускная способность* и *время отклика* системы. Обе характеристики измеряются на определенной нагрузке системы. Обе метрики имеют смысл для очень широкого класса систем; уточнение

того, какие нагрузки имеет смысл рассматривать, зависит, конечно, от класса системы и от требований к ней. Для систем управления базами данных это может быть некоторая смесь запросов или других действий разной сложности. Когда такая нагрузка определена, можно измерить среднее количество подобных действий, выполняемых за единицу времени (пропускная способность), или среднее время выполнения одного действия (время отклика). Во многих случаях имеет смысл оценивать время отклика отдельно для каждого класса действий (в зависимости от их сложности для системы).

Необходимо подчеркнуть, что для достижения высокой пропускной способности может потребоваться конфигурация системы, отличающаяся от конфигурации, необходимой для достижения низкого времени отклика, и улучшение одной из этих характеристик совсем не обязательно приводит к улучшению другой. Этот факт можно пояснить следующим образом. Для получения высокой пропускной способности следует максимально использовать имеющиеся вычислительные ресурсы, что может приводить к росту очередей заданий, ожидающих выполнения, и, следовательно, к увеличению времени отклика за счет ожидания в очереди. С другой стороны, для сокращения времени отклика необходимо сократить время ожидания, в том числе в очередях, поэтому вычислительные ресурсы должны работать с неполной нагрузкой.

Для времени отклика важно также различать измерение на стороне клиента и на стороне сервера. Время выполнения не очень сложного запроса на сервере может оказаться значительно меньше времени пересылки текста запроса и возврата результата по вычислительной сети. В связи с этим может быть важно оценивать время отклика не для отдельных запросов или других операций с базой данных, а для каких-либо функций приложения в целом. Например, для веб-приложений наиболее важным вариантом времени отклика является время, необходимое для генерации HTML-страницы в ответ на запрос пользователя.

Для параллельных систем наиболее важной характеристикой является *масштабируемость* (scalability). В действительности масштабируемость не является отдельной характеристикой, а показывает, как изменяется другая характеристика при изменении нагрузки и размеров вычислительной системы. Можно говорить о масштабируемости пропускной способности или масштабируемости времени отклика, но не о масштабируемости вообще. Для того чтобы оценить масштабируемость, необходимо оценить какую-либо характеристику для системы, состоящей из одного вычислителя, на определенном объеме базы данных и определенной нагрузке и ту же самую характеристику для системы,

в которой количество вычислителей, количество данных и нагрузка (число запросов, например) все увеличены в N раз. Тогда масштабируемость выражается отношением второго значения к первому. Поведение масштабируемости по пропускной способности показано на рис. 1.8.1.

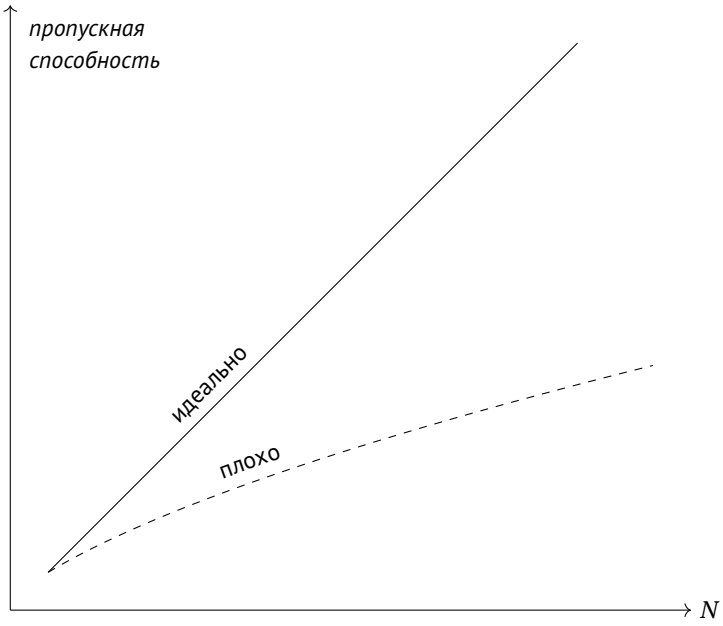


Рис. 1.8.1. Масштабируемость пропускной способности

Идеальное поведение масштабируемости по пропускной способности может быть представлено линейной зависимостью от N: система, содержащая в N раз больше оборудования и данных, способна обрабатывать в N раз больше запросов. В реальности такая масштабируемость недостижима, так как некоторая часть вычислительных ресурсов необходима для синхронизации работы параллельных вычислителей.

Для масштабируемости по времени отклика идеальная зависимость представляется константой: при увеличении количества вычислителей, объема данных и потока запросов время отклика не возрастает. Так же как и для пропускной способности, идеальная масштабируемость по времени отклика практически недостижима. Поведение масштабируемости по времени отклика иллюстрируется рис. 1.8.2.

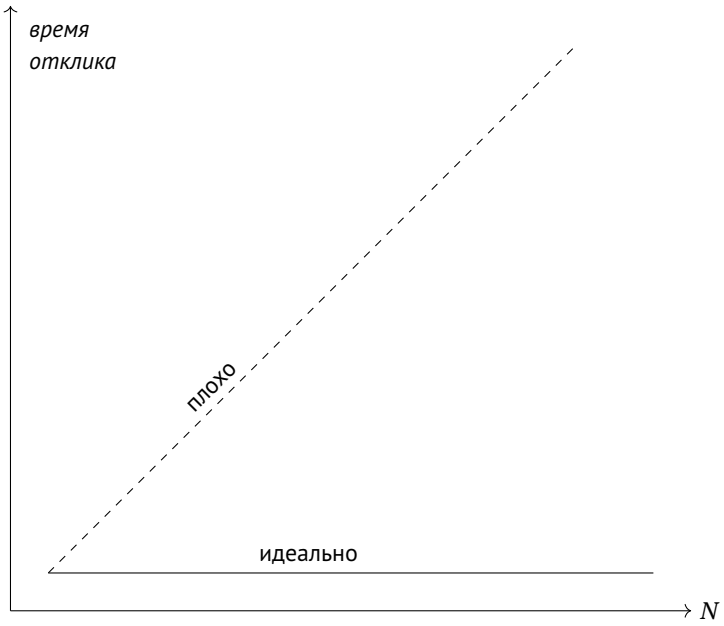


Рис. 1.8.2. Масштабируемость времени отклика

Еще одной характеристикой параллельных систем является *ускорение*, которое измеряется как отношение времени отклика на системе с одним вычислителем ко времени отклика на системе из N вычислителей.

Кроме характеристик производительности, имеется ряд других характеристик, из которых наиболее важной является *доступность* (availability). Доступность определяется как отношение времени нормальной работы системы к длине интервала времени, в течение которого измеряется доступность. Одновременное достижение высоких значений всех характеристик обычно требует значительного усложнения и удорожания системы.

Интегральные метрики, рассмотренные выше, полезны для оценки работы системы в целом, однако они мало полезны для оценки эффективности отдельных операций или запросов приложения. Поэтому в самой СУБД применяются метрики, значения которых можно предсказывать на основе информации об алгоритмах выполнения операций и статистических характеристик хранимых данных. Значением таких метрик является количество некоторых вычислительных ресурсов, необходимых для выполнения операции, например процессорное время или количество операций обмена с внешними устройствами,

а в случае параллельных или распределенных систем — еще и нагрузка на вычислительную сеть, которая может выражаться количеством сообщений, объемом передаваемых данных, количеством синхронных сообщений (с ожиданием ответа) и т. п.

Во многих случаях запросы приложений, сформулированные на высокоуровневом декларативном языке, могут быть выполнены многими различными способами, эквивалентными по результату, но использующими разное количество вычислительных ресурсов. Высокопроизводительные СУБД в таких случаях выбирают способ выполнения (план), для которого необходимое количество ресурсов минимально.

1.9. Создание приложений, взаимодействующих с базой данных

Современные СУБД поставляются с инструментами для создания и ведения баз данных. Эти инструменты предназначены для администраторов баз данных и помогают решать типовые задачи, такие как создание и изменение таблиц, редактирование записей, разграничение доступа и управление резервными копиями. Несмотря на то что эти программы обладают графическим интерфейсом, большинство задач удобнее решать, используя команды языка SQL.

Разумеется, такой интерфейс не подходит для бизнес-приложений. Система, ориентированная на массового пользователя, должна быть настолько понятна, чтобы пользователь, хорошо разбирающийся в предметной области, мог без всякого обучения начать с ней работать. Такие системы скрывают доступ к данным за графическим интерфейсом, который позволяет эффективно решить задачу и максимально ограждает пользователя от ошибок.

Для создания бизнес-приложений используется клиент-серверная архитектура. Ядро СУБД работает на сервере, а прикладная программа — на клиенте, причем сам клиент может быть сложным и состоять из нескольких уровней. Бизнес-логика может быть реализована как на сервере в виде хранимых процедур, так и на клиенте с использованием высокоуровневого языка программирования. Поскольку данные на клиенте и на сервере представлены по-разному, возникает проблема *потери соответствия* (impedance mismatch).

В случае объектно-ориентированных языков программирования и реляционных баз данных эта проблема называется объектно-реляционной потерей со-

ответствия. Реляционные базы данных не поддерживают основных концепций объектно-ориентированной парадигмы. Наибольшие затруднения при трансформации объектов в таблицы вызывает несоответствие системы типов, отображение наследования и многозначных связей, а также поддержка навигации между объектами.

Объектно-ориентированные языки пытаются решить проблемы при помощи каркасов объектно-реляционных отображений (object-relational mapping frameworks). Такие каркасы представляют собой библиотеки, написанные на объектно-ориентированном языке программирования. Разработчик приложения работает с привычными объектными моделями, которые автоматически преобразуются в таблицы и наоборот. Например, при операции сохранения каркас получает объект, отображает его в соответствующие таблицы и формирует SQL-запрос для вставки записи. При операции чтения каркас получает идентификатор объекта, формирует SQL-запрос для выборки данных, отображает найденные записи в объекты и возвращает их приложению.

Использование каркасов объектно-реляционных отображений ускоряет разработку, поскольку программисту не требуется глубоко знать ни SQL, ни реляционную теорию. Однако такие каркасы практически не используют специфические особенности конкретной СУБД, из-за чего тонкая настройка запросов становится невозможна. По-видимому, хорошим решением является использование каркасов для стандартных операций и чистого SQL для сложных запросов.

Попытка решить проблему несоответствия стала одной из причин создания баз данных NoSQL, которые представляют собой альтернативу реляционным СУБД. Такие базы данных не имеют структурированной схемы и работают напрямую с объектами. Преимущества и недостатки баз данных NoSQL рассматриваются в главе 9.

1.10. Итоги главы

В этой главе определены основные понятия, связанные с системами управления базами данных, обсуждены основные требования к таким системам, как они были определены исторически и как они эволюционировали на протяжении десятилетий существования СУБД. Более детально многие из этих тем разбираются в последующих главах.

1.11. Контрольные вопросы

Вопрос 1.1. Какие основные требования предъявляются к системам управления базами данных?

Вопрос 1.2. Какие основные компоненты содержит обобщенная трехуровневая модель данных ANSI/SPARC?

Вопрос 1.3. Каковы основные характеристики языков запросов в современных СУБД?

Вопрос 1.4. Что означает термин «независимость данных»?

Вопрос 1.5. Какие преимущества возникают при использовании независимости данных?

Вопрос 1.6. Что означает термин «согласованность данных»?

Вопрос 1.7. Что понимается под ограничением целостности в системах управления базами данных?

Вопрос 1.8. Как трактуются понятия безопасности и разграничения доступа в современных системах управления данными?

Вопрос 1.9. Какие основные метрики используются для оценки производительности?

Вопрос 1.10. Что такое архитектура клиент-сервер? Как распределяются программные компоненты?

Вопрос 1.11. Что называют объектно-реляционной потерей соответствия?

Глава 2

Теоретические основы БД

2.1. Модели данных

Будем называть *моделью данных* систему взаимосвязанных понятий и правил, предназначенную для описания структур и свойств данных, используемых (хранимых и обрабатываемых) в информационной системе. В этом курсе предполагается, что такая информационная система пользуется услугами некоторой СУБД. Поэтому можно также сказать, что модель данных задает способ описания схемы базы данных.

В состав развитой модели данных входят:

- способы описания данных: какие базовые (примитивные) типы можно использовать, каким образом строить сложные структуры данных из более простых;
- способы описания взаимосвязей между объектами данных;
- средства задания ограничений целостности;
- способы конструирования операций, которые можно использовать в рамках модели данных.

Не все модели данных, используемые в современных СУБД, содержат все перечисленные составляющие в развитой форме, однако все эти составляющие модели данных необходимы для того, чтобы удовлетворить основные требования, в частности рассмотренные в главе 1. Отсутствие каких-либо элементов или средств в модели данных, реализуемой в СУБД, обычно означает, что эти элементы должны быть реализованы на уровне приложений, использующих такую СУБД.

Например, если в модели данных СУБД отсутствует возможность описания сложных структур данных (скажем, любое значение рассматривается как последовательность байтов), то фактическая структура таких значений должна быть определена в приложении. Обычно такие данные трудно использовать

в других приложениях, и возникает необходимость преобразования в иную модель данных.

2.1.1. Идентификация и изменяемость

Многие свойства и особенности модели данных определяются тем, каким образом в рамках этой модели различаются используемые в ней объекты данных. Некоторые модели предусматривают наличие выделенного способа идентификации (например, объектного идентификатора в объектно-ориентированных моделях данных). В этом случае объекты данных считаются совпадающими, если у них совпадают идентификаторы. Другие элементы данных, входящие в тот же объект, образуют его состояние, которое может быть различным, и, следовательно, в таких моделях объекты обладают изменяемостью. Далее мы покажем, что не все модели данных обладают такой возможностью.

Многообразие методов идентификации можно условно разделить на следующие категории.

По естественным признакам объекта реального мира, который описывается объектом данных. Примером естественных идентификаторов может служить трехбуквенный код аэропорта, номер бронирования или биометрические свойства.

По искусственному значению (суррогату), которое генерируется информационной системой. Заметим, что суррогатный идентификатор, созданный в одной системе, может использоваться как естественный идентификатор в другой. В качестве примера можно назвать номер документа, идентифицирующего личность.

По связи объекта с другим, уже идентифицированным объектом. Этот метод полезен для того, чтобы различить объекты, которые в рамках модели невозможно различить другим способом. Например, «правое переднее колесо легкового автомобиля» идентифицирует колесо по его месту. Поскольку колеса взаимозаменяемы, в рамках модели они могут описываться совпадающими значениями всех атрибутов.

Другой пример такого типа идентификации можно получить, если требуется, чтобы представление некоторого объекта не зависело от того, в каком месте в памяти компьютера этот объект находится. В этом случае различные копии объекта можно различать по адресу в памяти, где находится копия.

Выбор метода идентификации особенно важен для проектирования базы данных и информационной системы, ее использующей. Для моделей данных важно наличие какого-либо идентификатора, а не его природа.

Если же способ идентификации не выделен, то единственным способом проверки на равенство является проверка на совпадение значений всех элементов данных. В таких моделях данных объекты, различающиеся значением хотя бы одного элемента, считаются различными. Следовательно, в таких моделях объекты изменяться не могут. Можно заменить объект на другой, но изменить невозможно.

Среди рассматриваемых в этой главе моделей данных в теоретической реляционной модели данных применяется идентификация по значениям атрибутов, а в объектных моделях данных и в модели данных «сущность-связь» предполагают наличие выделенных для идентификации атрибутов объектов.

Не следует думать, что поддержка изменяемости всегда является достоинством модели. Далее мы покажем, что наиболее широко используемая модель данных, реляционная, не поддерживает изменяемость. Это утверждение может показаться парадоксальным. Читателю, у которого оно вызывает протест, придется потерпеть до раздела 2.2, в котором приведено развернутое объяснение.

Вследствие неизменяемости в рамках реляционной модели данных удается построить мощные средства выполнения запросов. Создать аналогичные средства для моделей, поддерживающих изменяемость, не удается, несмотря на многолетние усилия исследователей и практиков. В некоторых моделях данных строятся две системы типов: отдельно для изменяемых и для неизменяемых объектов, в частности это было сделано в проекте стандарта объектных баз данных [3].

В некоторых случаях идентификаторы, явно не требуемые в рамках модели данных, могут быть выявлены на основе ограничений целостности. Это используется в теоретической реляционной модели данных для определения понятия ключа.

В реальности, конечно, практически все объекты обладают изменяемостью, поэтому при проектировании системы очень важно правильно выбрать способ идентификации, позволяющий взаимно-однозначно сопоставлять объекты реального мира с их представлениями в информационной системе.

2.1.2. Навигация и поиск по значениям

Поиск данных для обработки выполняется в любой системе управления базами данных, однако в рамках разных моделей данных он осуществляется поразному. Можно выделить следующие основные классы поисковых операций.

Навигационный доступ (переходы от одного объекта к другим происходят с помощью ссылок). Примером, не связанным непосредственно с базами данных, может служить использование URL для навигации в интернете.

Поиск по значениям (ассоциативный поиск). Примером, также не связанным с базами данных непосредственно, могут быть поисковые средства Яндекс или Google, хотя ассоциативный поиск вовсе не ограничивается теми видами текстового поиска, которые реализованы в этих системах.

Навигационный доступ использует явно заданные связи между объектами, например от текущего обрабатываемого объекта можно перейти к другому объекту по идентификатору, значение которого содержится в одном из атрибутов текущего объекта. Этот способ поиска характерен для объектно-ориентированных моделей данных, в том числе для объектно-ориентированных языков программирования.

Результатом поиска по значениям обычно является набор объектов данных, удовлетворяющих условиям поиска. Такой поиск может не использовать заранее заданные связи, и часто результат содержит несколько объектов данных. Этот способ часто используется в декларативных языках запросов, в том числе и в теоретической реляционной модели, и в используемых на практике системах управления базами данных, в которых реализован язык запросов SQL (обсуждаемый в главе 4).

Отметим, что навигация возможна, даже если модель данных не предусматривает явную идентификацию объектов данных. Например, можно по текущему элементу обрабатываемой коллекции найти коллекцию взаимосвязанных с ним объектов данных другого типа.

Не существует четкого различия между способами поиска, и многие модели допускают использование операций того и другого класса. Поэтому выбор способа поиска является не только свойством модели, но и отражает стиль программирования приложений.

2.1.3. Объекты и коллекции объектов

Гранулярность доступа определяет, что является единицей обмена при доступе к данным. Операции, определенные в модели данных, могут быть ориентированы на обработку отдельных объектов данных или на массовую обработку. Как правило, хотя и не обязательно, навигационный доступ предполагает обработку отдельных объектов данных, а поиск по значениям чаще всего предполагает массовую обработку.

Свойство гранулярности модели данных определяет, каким образом происходит взаимодействие между сервером базы данных и приложением и, следовательно, должно учитываться при проектировании приложения.

На уровне моделей данных невозможно четко отделить модели одного типа от моделей другого, однако в реализациях это различие становится очень существенным. Системы, ориентированные на обработку отдельных объектов данных, как правило, оказываются неэффективными при массовой обработке, и обратно, системы, достигающие очень высокой производительности при массовой обработке, могут быть крайне неэффективны, если они используются для обработки большого количества отдельных объектов.

Среди моделей, рассматриваемых в этой главе, как теоретическая реляционная модель, так и ее практически реализованные варианты ориентированы в первую очередь на массовую обработку. Системы, основанные на реляционных моделях данных, в том числе PostgreSQL, обеспечивают высокую эффективность массового извлечения данных, но могут оказаться значительно менее эффективными при обработке отдельных объектов данных.

2.1.4. Свойства моделей данных

Свойства моделей данных, перечисленные в этом разделе, могут показаться абстрактными, однако в действительности учет этих свойств оказывает очень существенное влияние на качество проектируемых систем. Более того, эти свойства важны для сравнительной оценки достоинств и недостатков той или иной модели данных.

Мы не рассматриваем детально ранние модели данных (иерархическую и сетевую), потому что по своим характеристикам и свойствам они похожи на объектные модели данных. Тем не менее заметим, что для всех таких моделей

(включая объектную) характерны навигационный способ доступа и ориентация на обработку отдельных объектов. В соответствии с этим как критерии эффективности, так и особенности реализации были ориентированы именно на эти свойства. Появление в начале 1970-х годов реляционной модели данных (рассматриваемой в следующем разделе) вызвало негативную реакцию многих практиков, так как в терминах общепринятых в то время критериев реляционная модель не могла обеспечить приемлемую эффективность реализации.

Спустя два десятилетия стало очевидно, что ориентация на обработку отдельных записей и навигацию не может обеспечить эффективную массовую обработку данных, и появились технологии и реализации реляционных систем, значительно превосходящие по производительности ранние системы, но в терминах новых критериев.

Распространение в последние годы моделей данных с ограниченными функциями (часто объединяемых под зонтиком NoSQL) связано с отказом от массовой обработки объектов на уровне базы данных и, соответственно, изменением требований к модели данных и критериев эффективности.

2.2. Реляционная модель данных

Обсуждение реляционной модели данных невозможно без небольшого исторического введения. Реляционная модель появилась в начале 1970-х годов в работах Э. Кодда (Edgar Codd) и ряда других исследователей. Большую роль в популяризации идей реляционной модели данных сыграли работы К. Дейта (Christopher Date). В течение почти десятилетия модель разрабатывалась как чисто теоретический способ описания свойств коллекций данных и языков запросов. При этом теоретики подчеркивали, что реляционная модель данных не решает вопросы организации хранения данных, а практики уверенно писали о том, что эффективная реализация реляционной модели данных невозможна.

В последующее десятилетие, однако, были исследованы методы и структуры хранения и поиска, а также способы оптимизации запросов, позволившие создать системы управления базами данных, основанные на вариантах реляционной модели и обеспечивающие высокую эффективность массовой обработки. Фактически системы такого типа широко используются до настоящего времени. К этому же классу систем относится и PostgreSQL.

Необходимо подчеркнуть, что радикальное изменение взглядов на возможность создания высокоэффективных систем на основе реляционной модели

в значительной мере определяется не технологиями хранения и индексирования, а изменением критериев эффективности и требований к системам: в 70-е годы, говоря об эффективности, подразумевали эффективность доступа к отдельным объектам данных и навигации между ними, а начиная с 80-х годов наиболее важными стали эффективность массовой обработки и поиск по значениям атрибутов (т. е. ассоциативный поиск). Именно поэтому абстрактные свойства моделей данных, рассмотренные выше в этой главе, представляются весьма важными.

Далее в этом разделе мы рассмотрим теоретическую реляционную модель данных и кратко обсудим отличия практических реализаций этой модели в системах, основанных на языке SQL. Более подробное изложение модели данных языка SQL отложим до главы 4.

2.2.1. Основные понятия реляционной модели данных

Домены

Неформально домен можно описать как множество значений, обладающих некоторыми общими свойствами. Например, можно говорить о доменах целых или вещественных чисел, домене длин, домене текстов и др.

В теоретической реляционной модели данных доменами называются множества некоторых значений. При этом предполагается, что все значения, находящиеся в доменах, являются скалярными константами, т. е. структура значений, даже если она существует, не рассматривается в рамках модели, и любые изменения приводят просто к новым значениям, а не к измененным старым. Кроме этого, требуется, чтобы на любом домене было определено бинарное отношение равенства. Это значит, что для любых двух значений из домена можно (с помощью этого отношения) определить, совпадают эти значения или нет. Необходимость в такой проверке возникает, когда значения приходят из разных источников, например значение, хранимое в базе данных, может сравниваться с константой в запросе.

Кроме этого, на доменах могут быть определены другие отношения, операции и функции. Например, на доменах, содержащих числовые значения, можно рассматривать:

- отношения упорядочивания $<$, $\leqslant$, $>$, $\geqslant$;
- арифметические операции, например $+$, $-$, $\times$, $\div$;

- функции, принимающие скалярные значения, например тригонометрические и другие.

Могут также быть определены функции, принимающие значения в другом домене, или функции нескольких аргументов из разных доменов. Например, на домене текстов можно рассматривать функцию, возвращающую длину текста, и функцию, выделяющую подстроку указанной длины начиная с указанной позиции.

Любые такие дополнительные отношения, функции и операции могут использоваться в рамках реляционной модели для вычисления новых значений при описании запросов, однако необходимым для самой реляционной модели является только отношение равенства.

Можно сказать, что в языках программирования понятию домена соответствует понятие абстрактного типа данных: как и абстрактный тип данных, домен определяет операции, которые можно выполнять над входящими в него значениями. Отличие состоит в том, что (как уже указано выше) значения, входящие в домен, считаются скалярными, т. е. не могут быть, например, массивами.

Домены могут быть взаимосвязаны. Например, домен целых чисел является подмножеством домена рациональных, или, в терминах операций и функций, один домен может быть специализацией другого (т. е. соответствовать подтипу абстрактного типа). Однако в реляционной теории все домены рассматриваются как независимые. Конечно, можно определить функции, преобразующие значения из одного домена в другой, например числовые значения можно записывать как текстовые строки, но в теоретической реляционной модели данных такие преобразования не нужны.

Ранние реализации реляционной модели допускали использование только ограниченного набора доменов, включающего числовые домены, домены дат и времени, домен литерных строк и т. п. Такое ограниченное понимание доменов вызвало критику и утверждения об ограниченности реляционной модели. Современные реализации реляционной модели, в том числе в PostgreSQL, предусматривают использование любых доменов, как это и предполагалось реляционной теорией с самого начала.

Например, можно определить домены длин и весов. Значения в этих доменах, разумеется, будут числовыми, однако наборы операций будут отличаться: длины можно складывать и вычитать, но результат перемножения длин будет уже в другом домене, возможно, в домене площадей. Умножать длины можно только на значение из числового домена, при этом результат тоже будет длиной.

Точно так же можно складывать значения веса, но вряд ли целесообразно определять операцию сложения веса с длиной.

Ничто не мешает использовать домен, содержащий точки плоскости, при этом координаты точки (декартовы или полярные) будут вычисляться с помощью функции, принимающей значения в подходящем домене.

По историческим причинам возможности определения доменов рассматриваются как расширения реализаций реляционной модели. Возможности, предоставляемые системой PostgreSQL, обсуждаются в главе 8.

Отношения, атрибуты и кортежи

Перейдем к обсуждению центрального понятия теоретической реляционной модели — понятия *отношения*.

Формально отношение определяется как n-местный (или n-арный) предикат, т. е. функция с n аргументами, принимающая булево значение: истина (true) или ложь (false). При этом предполагается, что для каждого аргумента задан домен, из которого могут выбираться значения этого аргумента.

Напомним, что в математике определение функции не предполагает наличия какой-либо формулы или алгоритма, позволяющего вычислить значение этой функции по значениям аргументов. Это справедливо и для предикатов, рассматриваемых в реляционной теории.

В отличие от обычной математической нотации, в реляционной теории используются не позиционные (т. е. занумерованные натуральными числами), а именованные аргументы. Именованные аргументы отношения называются *атрибутами отношения*. Использование имен позволяет при необходимости записывать атрибуты в произвольном порядке.

Имена атрибутов одного отношения обязательно должны быть попарно различными. С каждым атрибутом связан некоторый домен, из которого выбираются значения этого атрибута (это уже было отмечено выше), при этом один и тот же домен может быть связан с несколькими различными атрибутами одного отношения.

Множество всех атрибутов отношения называется *схемой отношения*.

Совокупность из n значений, по одному значению (из соответствующего домена) для каждого атрибута, называется *кортежем*. Можно сказать, что кортеж —

это набор значений аргументов, для которого можно вычислить предикат, соответствующий отношению. Говорят, что кортеж принадлежит отношению, если предикат принимает истинное значение на этом кортеже.

Семантика отношений состоит в том, что они описывают зависимости между своими атрибутами. Считается, что значения в кортеже взаимосвязаны, если этот кортеж принадлежит отношению. Если атрибуты соответствуют свойствам некоторого класса объектов реального мира, то принадлежность кортежа отношению может означать, что объект с такими значениями свойств существует в реальности. Другими словами, каждый кортеж, принадлежащий отношению, выражает истинность некоторого факта.

Количество кортежей в каком-либо отношении называется *кардинальностью* этого отношения.

Проиллюстрируем приведенные выше определения примерами.

Предположим, что отношение exams имеет атрибуты {name, course, grade}, принимающие значения в соответствующих доменах. Тогда истинное значение предиката на следующих кортежах указывает на то, что упомянутые студенты получили указанные оценки по этим курсам.

exams (name := 'Анна', course := 'Базы данных', grade := 5)

exams (name := 'Анна', course := 'Анализ данных', grade := 5)

exams (course := 'Анализ данных', grade := 4, name := 'Виктор')

exams (name := 'Нина', grade := 5, course := 'Базы данных')

Напомним, что предикаты (отношения) обычно не задаются какими-либо формулами. Поэтому, для того чтобы записать предикат, используется принятый в математике табличный способ представления функций: для каждого значения аргумента выписывается соответствующее ему значение функции. В нашем случае функция (предикат) имеет только два значения, поэтому достаточно выписать только те значения аргументов (кортежи), которые входят в отношение (или, что эквивалентно по приведенному выше определению, предикат принимает истинное значение).

Каждый кортеж состоит из n значений, поэтому отношения принято записывать в виде таблиц, содержащих n колонок, соответствующих атрибутам отношения (и озаглавленных именами атрибутов), а каждая строка таблицы соответствует одному кортежу. При этом порядок, в котором расположены кортежи в таблице, не имеет никакого значения.

Поскольку каждый кортеж представляет истинность некоторого факта, повторное включение любого кортежа не дает никакой новой информации. В теоретической реляционной модели данных предполагается, что все кортежи, входящие в отношение, различны, и поэтому совокупность кортежей отношения является множеством (в математическом смысле).

В табличном представлении отношение exams, приведенное выше, может выглядеть, как показано на рис. 2.2.1.

exams

name	course	grade
Анна	Базы данных	5
Анна	Анализ данных	5
Виктор	Анализ данных	4
Нина	Базы данных	5

Рис. 2.2.1. Пример отношения в табличной записи

Отметим, что представление отношений таблицами возможно только для отношений, содержащих конечное число кортежей.

Во многих реализациях реляционной модели данных отношения называются *таблицами,* атрибуты — *колонками* или *столбцами*, а кортежи — *строками*. В дальнейшем термины «колонка» и «столбец» будут использоваться как синонимы.

В отличие от теоретической реляционной модели данных, табличные реализации этой модели в системах управления базами данных допускают хранение совпадающих строк в таблицах, если уникальность не задана как ограничение целостности в схеме таблицы.

2.2.2. Реляционная алгебра

Успех реляционной модели данных, интенсивно используемой на протяжении десятилетий, определяется тем, что в рамках этой модели определены мощные и выразительные языки манипулирования данными. Один из таких языков строится на основе операций, позволяющих определять новые отношения из уже имеющихся. Это позволяет использовать результаты выполнения операций в качестве аргументов для выполнения следующих операций, т. е. задавать требуемые вычисления в форме выражений.

В математике подобные системы операций (определенных на некотором множестве) называются *алгебрами*. Операции могут быть частичными, например операция деления чисел определена, только если делитель отличается от нуля. Набор операций, рассматриваемых в этом подразделе, называется *реляционной алгеброй*. Эти операции определены на множестве всех возможных конечных отношений.

Почти все операции реляционной алгебры являются частичными, т. е. они определены не для любых аргументов или их комбинаций (для бинарных операций). В некоторых случаях, для того чтобы операция стала выполнимой, достаточно выполнить переименование атрибутов. Для этой цели можно ввести операцию, которая изменяет имена атрибутов в схеме отношения, но не меняет никаких значений в кортежах, входящих в отношение, и не меняет доменов, связанных с атрибутами.

Теоретико-множественные операции

Поскольку кортежи, входящие в отношение, образуют множества, можно применять обычные теоретико-множественные операции к любой паре отношений, имеющих одинаковые схемы. Эти операции тесно связаны с логическими операторами булевой алгебры.

В реляционной алгебре используются следующие операции:

объединение UNION: в результат включаются кортежи, входящие в первый *или* во второй аргумент;

пересечение INTERSECT: результат содержит только кортежи, входящие в первый *и* второй аргументы;

разность EXCEPT: результат содержит кортежи, входящие в первый *и не* входящие во второй аргумент. В некоторых вариантах алгебры эта операция называется MINUS.

Используя эти операции, можно строить производные, определяемые алгебраическими выражениями. Например, операция симметричной разности отношений *R* и *S*, соответствующая логической операции XOR (исключающее или), может быть определена как

(*R* UNION *S*) EXCEPT (*R* INTERSECT *S*)

или как

(*R* EXCEPT *S*) UNION (*S* EXCEPT *R*).

Заметим, что в реляционной алгебре нет теоретико-множественной операции дополнения. Это связано с тем, что отношения должны быть конечными множествами, а многие домены бесконечны.

Унарные операции

Операции с одним аргументом (унарные операции) позволяют выделить из отношения ту информацию, которая нужна данному приложению при данном выполнении.

Операция проекции PROJ включает в результат подмножество атрибутов отношения, переданного в качестве аргумента. На рис. 2.2.2 показан результат проекции отношения exams на подмножество атрибутов {name, course}. Такой результат показывает только факт сдачи экзамена, но не оценку.

PROJ [name, course] exams

name	course
Анна	Базы данных
Анна	Анализ данных
Виктор	Анализ данных
Нина	Базы данных

Рис. 2.2.2. Реляционная операция проекции

В результате исключения части атрибутов при выполнении операции проекции может оказаться, что различные кортежи исходного отношения станут совпадать по значению. Реляционная операция проекции обязательно исключает возникшие дубликаты. На рис. 2.2.3 проекция на атрибуты {course, grade} содержит меньшее количество кортежей, чем исходное отношение.

Реализации реляционной модели выполняют удаление дубликатов, только если оно явно задано в запросе.

Операция фильтрации FILTER строит новое отношение, включая в него строки исходного отношения, удовлетворяющие условию, выраженному логической формулой. В первых работах по реляционной модели данных эта операция называлась ограничением, потому что она соответствует понятию ограничения

PROJ [course, grade] exams

course	grade
Базы данных	5
Анализ данных	5
Анализ данных	4

Рис. 2.2.3. Проекция с удалением дубликатов

функции (в этом случае отношение рассматривается как предикат, т. е. функция). Зачастую эта операция называется селекцией, однако такое наименование создает ложную связь с оператором SELECT языка запросов SQL, который обладает значительно более мощными возможностями.

Условие в операции фильтрации строится по следующим правилам.

- Простое условие представляется атрибутом, который сравнивается с константой или с другим атрибутом того же отношения с помощью бинарного отношения равенства, определенного на домене этого атрибута (или атрибутов). Поскольку такое отношение определено на любом домене, любой атрибут можно сравнивать с константой из того же домена.
- Ранее определенные по этим правилам условия можно соединять логическими операторами AND, OR, NOT.
- Условие может быть заключено в круглые скобки.

На рис. 2.2.4 показан результат выполнения операции фильтрации. Заметим, что результатом выполнения фильтрации может быть пустое отношение.

FILTER [course='Анализ данных' AND grade=5] exams

name	course	grade
Анна	Анализ данных	5

Рис. 2.2.4. Операция фильтрации

Произведение и соединение

Операция прямого (или декартова) произведения PROD строит все пары кортежей из первого и второго аргументов и по каждой такой пар создает кортеж

результата, содержащий все значения атрибутов кортежей, входящих в эту пару. Необходимо, чтобы все имена атрибутов аргументов этой операции были различными. Для этого перед выполнением операции надо переименовать атрибуты с совпадающими именами в одном из отношений.

На рис. 2.2.5 показано отношение courses, содержащее информацию о зачетных единицах, которые можно получить за курс, а на рис. 2.2.6 представлено произведение отношений.

courses

title	credits
Базы данных	5
Анализ данных	10

Рис. 2.2.5. Отношение courses

exams PROD courses

name	course	grade	title	credits
Анна	Базы данных	5	Базы данных	5
Анна	Анализ данных	5	Базы данных	5
Виктор	Анализ данных	4	Базы данных	5
Нина	Базы данных	5	Базы данных	5
Анна	Базы данных	5	Анализ данных	10
Анна	Анализ данных	5	Анализ данных	10
Виктор	Анализ данных	4	Анализ данных	10
Нина	Базы данных	5	Анализ данных	10

Рис. 2.2.6. Прямое произведение отношений

Очевидно, что кардинальность произведения отношений равна произведению кардинальностей аргументов. Этот факт используется для оценки стоимости вычислений алгебраических выражений.

Само по себе произведение нельзя считать полезной операцией, потому что результат содержит кортежи, отражающие истинные, но не обязательно связанные между собой факты. Намного более полезный результат дает операция соединения JOIN, представляющая собой произведение с последующей фильтрацией. Поскольку результатом произведения является одно отношение, условие фильтрации может в этом случае содержать и сопоставлять атрибуты из разных

аргументов, выделяя, таким образом, взаимосвязанные кортежи из разных отношений.

Для соединений общего вида (часто называемых тета-соединениями) условие фильтрации (часто обозначаемое буквой θ) может быть любым, однако обычно используются условия, представляющие собой конъюнкцию условий на равенство значений пар атрибутов. Такие соединения называются эквисоединениями. Поскольку значения сравниваемых атрибутов в получаемом отношении будут одинаковыми, целесообразно выполнить проекцию, которая исключит дублирующие значения.

Пример соединения отношений exams и courses показан на рис. 2.2.7. Поскольку условие соединения состоит в проверке равенства значений атрибутов, это соединение является эквисоединением.

exams JOIN [course=title] courses

name	course	grade	title	duration
Анна	Базы данных	5	Базы данных	5
Нина	Базы данных	5	Базы данных	5
Анна	Анализ данных	5	Анализ данных	10
Виктор	Анализ данных	4	Анализ данных	10

Рис. 2.2.7. Соединение отношений

По определению, операция соединения является производной, так как она выражается через произведение и фильтрацию. Несмотря на это, операция соединения рассматривается отдельно, потому что она очень важна для приложений. Важно и то, что для операции соединения существуют намного более эффективные алгоритмы, чем алгоритмы, основанные на вычислении прямого произведения.

Существует еще несколько вариантов операции соединения; в этой книге они обсуждаются в контексте языка запросов SQL.

Свойства операций реляционной алгебры

Перечислим основные алгебраические тождества, справедливые для операций реляционной алгебры.

Коммутативность. Операции объединения, пересечения, произведения и соединения коммутативны.

Например:

R UNION $S = S$ UNION R,

R JOIN $S = S$ JOIN R.

Ассоциативность. Операции пересечения, объединения, произведения и соединения ассоциативны.

Например:

R UNION (S UNION T) = (R UNION S) UNION T,

R JOIN (S JOIN T) = (R JOIN S) JOIN T.

Дистрибутивность. Пары операций объединения, пересечения, произведения (или соединения) подчиняются дистрибутивным законам.

Пересечение дистрибутивно относительно объединения:

R INTERSECT (S UNION T) = (R INTERSECT S) UNION (R INTERSECT T).

Объединение дистрибутивно относительно пересечения:

R UNION (S INTERSECT T) = (R UNION S) INTERSECT (R UNION T).

Произведение и соединение дистрибутивны относительно операций объединения и пересечения:

R PROD (S UNION T) = (R PROD S) UNION (R PROD T);

R PROD (S INTERSECT T) = (R PROD S) INTERSECT (R PROD T).

Кроме этого, имеются тождества, связывающие унарные операции друг с другом и с бинарными: можно заменять одну операцию несколькими более простыми и обратно, заменять фильтрации теоретико-множественными операциями и др.

Существование алгебраических тождеств позволяет преобразовывать алгебраические выражения в эквивалентные. Результат вычисления эквивалентных выражений будет, конечно, одинаковым, однако сложность вычисления может отличаться очень значительно (на несколько порядков). СУБД может выбрать среди эквивалентных способов записи запроса в виде выражения такой, выполнение которого требует меньшего количества вычислительных ресурсов (таких как процессорное время или количество операций обмена данными).

2.2.3. Другие языки запросов

Одним из основных требований к СУБД, перечисленных в главе 1, является наличие языка запросов. Желательно, чтобы такой язык был декларативным, т. е. позволял описывать, какой требуется результат, не указывая способа его вычисления. Реляционная алгебра не может считаться декларативным языком, так как алгебраическое выражение определяет порядок выполнения операций.

Существуют языки запросов более высокого уровня, позволяющие записать требования к результату выполнения запроса в виде набора логических условий. Такие языки называются исчислениями. Запрос в исчислении представляется набором правил.

В левой части правила (которая называется *головой* правила) обычно определяется схема отношения, которое является результатом вычисления правила, а в правой (в *теле* правила) — условие, которому должны удовлетворять кортежи, включаемые в результат вычислений.

Условие, размещаемое в теле правила, формируется следующим образом.

- Простые условия задаются предикатами, определенными на доменах; в качестве аргументов предикатов могут использоваться переменные или константы.
- Условие может быть заключено в скобки.
- Условие может быть построено из более простых условий с помощью логических операций $\wedge$, $\vee$, $\neg$.
- Условие, содержащее свободные переменные, может быть замкнуто с помощью кванторов всеобщности $\forall$ или существования $\exists$.
- Свободные переменные могут использоваться в голове правила в качестве атрибутов результирующего отношения.

Различают исчисления с переменными на кортежах и на доменах. Для переменных на кортежах указывается принадлежность переменной к отношению; при этом значения атрибутов обозначаются именем переменной, за которым следует имя атрибута, отделенное точкой. Голова и тело правила разделяются знаком $:-$.

Например, соединение двух отношений, представленное на рис. 2.2.7, может быть записано следующей формулой в исчислении:

$$\text{JoinResult}\,(x.\text{name}, x.\text{course}, x.\text{grade}, y.\text{title}, y.\text{credits}) \;:-\; x \in \text{exams} \;\wedge\; y \in \text{courses} \;\wedge\; x.\text{course} = y.\text{title}$$

В исчислении на доменах переменные принимают значения в доменах. Из таких переменных и констант формируются кортежи, принадлежность которых к отношениям базы данных становится одним из простых условий в выражении, описывающем запрос.

Например, тело правила

$$\{\text{name}, \text{course}, \text{grade}\} \in \text{exams} \;\wedge\; \text{grade} < 5$$

описывает запрос на выборку результатов экзаменов с оценкой ниже отличной.

В отличие от исчислений общего вида, в реляционном исчислении не допускается рекурсия, т. е. отношение, находящееся в голове правила, не может использоваться в его теле, а если правил несколько, то не допускается также взаимная рекурсия.

Существует несколько вариантов реляционных исчислений, однако можно доказать, что все они эквивалентны друг другу и эквивалентны реляционной алгебре. В данном случае эквивалентность означает, что любой запрос, который можно записать на одном из этих языков, можно выразить и на другом эквивалентном языке. Ни исчисления, ни доказательство эквивалентности не рассматриваются в этой главе.

Применяемый в реализациях язык запросов SQL занимает промежуточное положение между алгеброй и исчислением и позволяет использовать формы записи запросов, близкие как к алгебраическим выражениям, так и к формулам в исчислении. Важно отметить, что независимо от выбранного способа записи при подготовке запроса к выполнению он переводится в алгебраическое выражение, при этом среди эквивалентных выбирается такое выражение, выполнение которого будет вычислительно эффективно.

2.2.4. Особенности реляционной модели данных

Сопоставим свойства реляционной модели данных с характеристическими особенностями других моделей данных, которые обсуждаются в начале этой главы.

Семантика реляционной модели определяется тем, что кортежи соответствуют фактам, точнее, утверждениям об объектах реального мира. Роль объектов данных выполняют кортежи, которые в реляционной модели не имеют никакой специально выделенной идентификации. Поэтому формально кортежи не могут быть изменяемыми в рамках теоретической реляционной модели. Можно заменить одно значение на другое, но это будет другой кортеж, а не модификация старого.

Все операции реляционной алгебры в качестве результата создают новое отношение, поэтому реляционная модель, несмотря на свой достаточно абстрактный характер, ориентирована на массовую обработку данных.

Наконец, в реляционной алгебре предусмотрены только критерии поиска, использующие значения атрибутов. Более того, взаимосвязи между кортежами различных отношений также устанавливаются в операциях соединения на основе значений атрибутов. Поэтому организация навигационного доступа в этой модели требует дополнительных соглашений об использовании атрибутов, обычно выходящих за рамки этой модели.

Можно, однако, используя ограничения целостности (которые обсуждаются ниже), организовать и навигацию, и даже обработку отдельных кортежей.

2.2.5. Нормальные формы

Функциональные зависимости и ключи

В представленном выше изложении реляционной модели данных неявно предполагалось, что атрибуты отношения независимы или зависимости между атрибутами неизвестны. Однако значительная часть реляционной теории анализирует различные типы зависимостей, часть из которых рассматривается в этом подразделе.

Пусть X и Y — некоторые множества атрибутов одного отношения. Говорят, что Y *функционально зависит* от X, и пишут $X \to Y$, если для любой комбинации значений атрибутов из X может существовать только одна комбинация значений Y, входящая в отношение.

Другими словами, функциональная зависимость просто означает, что существует некоторая функция с областью определения X и принимающая значения в Y, которая определяет значения Y для любого кортежа, который может входить в рассматриваемое отношение. Подчеркнем, что для использования в реляционной теории важно только существование такой функции, но не способ ее вычисления.

Заметим, что если $Y \subset X$, то $X \to Y$, потому что любая комбинация значений атрибутов однозначно определяет сама себя. Такие зависимости называются *тривиальными* и существуют для любого множества атрибутов. В частности, все атрибуты отношения зависят от схемы этого отношения, и каждый атрибут зависит сам от себя.

Нетривиальные зависимости не могут быть выведены формально, они должны отражать закономерности, которым подчиняются свойства объектов реального мира, представленных в СУБД. Подчеркнем, что такие закономерности должны быть справедливы для любого состояния базы данных, т. е. должны выполняться для всех без исключения объектов реального мира.

Например, в отношении exams оценка (grade) функционально зависит от пары атрибутов {name, course}, потому что каждый студент может иметь только одну итоговую оценку по любой дисциплине, хотя мы и не можем вычислить эту оценку по значениям атрибутов, от которых она зависит. В этом же отношении, очевидно, нет зависимости между атрибутами name и course, потому что любой курс может сдавать несколько студентов и любой студент может сдавать несколько курсов. Очевидно также, что ни name, ни course не зависят от оценки, даже в комбинации с другим атрибутом. Поэтому {name, course} $\to$ {grade} является единственной нетривиальной функциональной зависимостью в этом отношении.

Множество атрибутов, от которого функционально зависят все атрибуты отношения, называется *возможным ключом* отношения.

В любом отношении есть, по крайней мере, один возможный ключ, потому что любой атрибут тривиально зависит от всех атрибутов отношения. Заметим, что

любые два кортежа отношения различаются по значениям атрибутов возможного ключа. Действительно, если значения атрибутов возможного ключа совпадают, то остальные тоже должны совпадать, потому что они функционально зависят от возможного ключа.

Возможный ключ называется *минимальным ключом*, если после исключения из него любого атрибута оставшееся множество атрибутов не является возможным ключом.

В одном отношении может быть несколько минимальных ключей. Один из минимальных ключей выбирается в качестве *первичного ключа* отношения. Понятие первичного ключа не имеет значения для реляционной теории, однако оно важно для применения этой теории. Значения первичного ключа, как и любого возможного ключа, уникальны и могут поэтому использоваться для идентификации объектов реального мира, описываемых кортежами отношения.

Мы по-прежнему не можем говорить об изменяемости кортежей, но можем говорить, что разные кортежи с одинаковыми значениями первичного ключа описывают разные состояния одного реального объекта. Конечно, эти кортежи хранятся в различных состояниях отношения (теоретически — в разных отношениях, описывающих состояние реальности в разные моменты времени), т. к. сосуществовать в одном отношении они не могут.

Например, в отношении exams единственным минимальным возможным ключом является {name, course}. Эта пара атрибутов составляет первичный ключ данного отношения.

Нормализация

Наличие нетривиальных функциональных зависимостей может приводить к нежелательным эффектам, которые принято называть *аномалиями*.

Рассмотрим отношение, приведенное выше на рис. 2.2.7.

Кроме уже упомянутой зависимости {name, course} → {grade}, в нем имеется зависимость {course} → {credits}. Вследствие этой зависимости в отношении имеется избыточность: зачетные единицы указаны столько раз, сколько раз встречается курс. Более существенный недостаток состоит в том, что в таком отношении невозможно хранить информацию о курсе, который не сдавал ни один студент. Включить информацию о курсе можно только вместе с оценкой и именем студента, а удаление всех оценок по курсу приводит к потере информации о курсе.

Причина состоит в том, что атрибут credits зависит только от части первичного ключа. Для устранения аномалий в этом случае необходимо вместо такого отношения использовать две его проекции, совпадающие с нашими отношениями exams и courses. Это не приводит к потере информации, потому что исходное отношение может быть получено как результат соединения. В нашем случае, конечно, оно и было результатом соединения, но в теоретической реляционной модели хранимые и вычисленные отношения неразличимы.

По историческим причинам отношения, в которых все атрибуты имеют скалярные значения, называются отношениями в *первой нормальной форме* (1NF). В нашем изложении теории все отношения находятся в 1NF.

Отношения, в которых отсутствуют зависимости от неполного ключа, называются отношениями во *второй нормальной форме* (2NF).

Аномалии могут быть вызваны также транзитивными зависимостями. Если имеются функциональные зависимости $X \to Y$ и $Y \to Z$, то существует еще и зависимость $X \to Z$, которая является суперпозицией первых двух. Такие комбинации функциональных зависимостей приводят к аномалиям, потому что каждый комплект значений атрибутов из Z будет повторен вместе с соответствующими значениями атрибутов из Y.

Например, в отношении со схемой {emp, dept, mgr}, для каждого сотрудника указан его отдел и менеджер. В таком отношении имеется избыточность, потому что у всех сотрудников одного отдела менеджер один и тот же.

Для устранения таких аномалий исходное отношение заменяется на его проекции таким образом, что зависимости, являющиеся транзитивными, оказываются в разных отношениях. В нашем примере такими отношениями могут быть {emp, dept} и {dept, mgr}. Эквисоединение этих проекций восстанавливает исходное отношение, поэтому при создании проекций потери информации не происходит. Неформально устранение транзитивных зависимостей переводит отношение в *третью нормальную форму* (3NF).

Существует два неэквивалентных определения третьей нормальной формы. Использованный выше вариант принято обозначать 3NF. Говорят, что отношение находится в *нормальной форме Бойса-Кодда* (BCNF), если для любой нетривиальной функциональной зависимости $X \to Y$ между атрибутами этого отношения множество X содержит некоторый ключ этого отношения. Различие между 3NF и BCNF для нас несущественно.

Построение схемы реляционной базы данных (т. е. набора схем отношений) можно начинать не с определения отношений, а с определения перечня атрибутов и функциональных зависимостей. Известны алгоритмы, обладающие полиномиальной сложностью, которые на основе этой информации строят схему базы данных, состоящую из отношений в третьей нормальной форме (или в BCNF, в зависимости от алгоритма).

Практическое значение нормальных форм и нормализации состоит в том, что они дают критерии, по которым можно оценивать качество логической структуры базы данных. Подчеркнем, что речь идет именно о логической структуре базы данных, на основе которой строятся представления структуры данных для приложений. При этом структуры хранения могут отличаться от логической структуры, поскольку при их проектировании учитываются и другие критерии.

Непосредственное хранение логической схемы может привести к необходимости частого выполнения вычислительно сложных операций соединения во многих запросах. Для того чтобы исключить излишние вычисления, в подобных случаях целесообразно организовать хранение ненормализованных отношений (на уровне схемы хранения, а не на логическом уровне).

В некоторых случаях нормализацией называют проектные решения, никакого отношения к нормализации не имеющие. Типичным примером является замена значений атрибутов (например, строковых) на суррогатные идентификаторы с последующим вынесением строковых значений в отдельное отношение. Такое проектное решение может иметь некоторые основания, однако связь с нормализацией состоит только в том, что оно вводит искусственные транзитивные зависимости. Как указывалось выше, в реляционной модели данных все значения являются константами, они идентифицируют сами себя, и поэтому в рамках этой модели никакой необходимости в дополнительных суррогатных идентификаторах нет. Отметим, что зачастую именно такие проектные решения приводят к излишнему усложнению многих запросов.

Другие зависимости и нормальные формы

В теоретической реляционной модели, кроме функциональных зависимостей, рассматривается целый ряд других классов зависимостей и связанных с ними нормальных форм.

Так, *многозначные зависимости* определяют соответствие между группами значений атрибутов. Устранение нежелательных многозначных зависимостей приводит к четвертой нормальной форме.

Мы не будем детально рассматривать эти типы зависимостей и соответствующие нормальные формы, потому что их практическое значение невелико.

2.2.6. Практические варианты реляционной модели данных

Большинство используемых в настоящее время систем управления базами данных основано на реляционной модели данных, однако все эти реализации обладают существенными отличиями от этой теоретической модели.

Для того чтобы не отпугнуть потребителей, не имеющих соответствующей подготовки, математические термины заменяются другими: вместо отношений говорят о таблицах, атрибуты называются колонками, а кортежи — строками таблиц. (В некоторых публикациях на русском языке строки таблиц принято называть «рядами».)

Далее мы рассмотрим более существенные отличия практических вариантов реляционной модели данных от теоретической.

Неопределенные значения

Одним из важнейших отличий практических реализаций является возможность использования неопределенных значений атрибутов, обычно обозначаемых ключевым словом NULL. Предполагается, что неопределенные значения можно задавать в тех случаях, когда значение атрибута не известно, не определено или не имеет смысла в сочетании со значениями других атрибутов той же строки.

Использование неопределенных значений зачастую упрощает проектирование схемы базы данных за счет некоторого усложнения кода приложений. Так, при отображении в базу данных иерархии классов все объекты могут быть отображены в одну таблицу; при этом атрибуты, имеющиеся только у объектов подкласса, получают неопределенные значения для объектов суперкласса. В этом случае принадлежность объекта к определенному классу должна устанавливаться в коде приложения, чтобы предотвратить некорректное использование атрибутов, имеющих неопределенные значения.

Довольно часто неопределенные значения используются в тех случаях, когда по каким-либо причинам значение не является обязательным. Например, для пассажира, не имеющего карты постоянного клиента, значение соответствующего атрибута может быть неопределенным.

В теоретической реляционной модели данных неопределенные значения не допускаются, потому что их использование существенно усложняет определение основных операций и приводит к появлению неустранимых парадоксов.

Например, операции над атрибутами, значениями которых являются неопределенные значения, в результате дают неопределенные значения, однако операция фильтрации интерпретирует неопределенное значение логического предиката как ложное. Более детально нелогичности и парадоксы, связанные с использованием NULL, обсуждаются в главе 4.

При проектировании баз данных хорошей практикой считается запрет на использование неопределенных значений для всех атрибутов, кроме тех, для которых неопределенные значения необходимы. Запрет на использование неопределенных значений является ограничением целостности.

Дубликаты

В отличие от теоретической реляционной модели, практические реализации допускают появление идентичных строк как в хранимых таблицах, так и в результатах выполнения запросов. Использование дубликатов разрешается стандартом потому, что в ранних системах устранение дубликатов или проверка их отсутствия в хранимых таблицах оказывались вычислительно сложными и могли приводить к существенному снижению производительности системы.

Конечно, для удаления дубликатов аналогичные вычисления требуются и в современных условиях, однако их влияние на общую производительность систем стало менее существенным вследствие увеличения мощности вычислительных систем на несколько порядков, и, с другой стороны, усложнение приложений повысило значение логической корректности.

Наличие дубликатов в хранимых таблицах чаще всего является следствием ошибки разработчиков приложения.

Дополнительные операции

Как возможность использования неопределенных значений, так и допущение дубликатов приводят к изменению семантики и алгебраических свойств теоретических реляционных операций в промышленных реализациях СУБД.

Так, допущение дубликатов приводит к необходимости различать теоретико-множественные операции, исключающие дубликаты из результата и оставляющие их. Соответственно, каждая из операций объединения, пересечения и разности существует в двух вариантах, и появляется дополнительная операция устранения дубликатов. Подчеркнем еще раз, что алгебраические свойства операций, допускающих дубликаты, отличаются от свойств обычных реляционных операций, поэтому нецелесообразно исключать последние из алгебры.

Практические реализации предусматривают также операции внешнего соединения, результаты которых могут содержать неопределенные значения. Например, операция левого внешнего соединения включает в результат те кортежи первого аргумента, для которых не нашлось пары во втором аргументе, дополняя эти кортежи неопределенными значениями для атрибутов второго операнда. Подробнее операции внешнего соединения рассмотрены в главе 4. Здесь мы только обратим внимание на то, что алгебраические свойства этих операций отличаются от свойств обычной операции соединения. Например, левое внешнее соединение некоммутативно.

2.3. Средства концептуального моделирования

Проектирование информационных систем и лежащих в их основе баз данных является довольно сложной задачей. Описание всех используемых в системе или взаимодействующих с ней типов объектов реального мира может насчитывать сотни и тысячи единиц, соотношения и взаимосвязи между ними зачастую оказываются нетривиальными и требуют глубоких знаний процессов, происходящих в реальном мире.

Чтобы упростить взаимопонимание между специалистами предметной области и разработчиками информационных систем, используются разнообразные языки моделирования, среди которых широкой известностью пользуются унифицированный язык объектного моделирования UML и модель данных концептуального уровня «сущность-связь» (Entity-Relationship, ER). Различие между этими инструментами моделирования в том, что UML позволяет описывать высокоуровневую объектную модель всей системы, в том числе поведение, т. е. описывать функционирование разрабатываемой системы, в то время как ER в основном описывает свойства данных, используемых в системе. Существуют и другие средства моделирования, описывающие отдельные стороны ее функционирования, например языки моделирования бизнес-процессов.

В этом разделе обсуждается главным образом модель данных «сущность-связь», так как ее развитые возможности позволяют описывать самые разнообразные свойства данных, далеко не всегда легко представляемые средствами более общих языков моделирования, в том числе UML. В рамках этого курса будут рассмотрены только основные особенности и возможности модели «сущность-связь»; для более детального освоения модели и процесса проектирования следует обратиться к специальной литературе и документации по системам проектирования баз данных с использованием этой модели.

2.3.1. Модель данных «сущность-связь»

Основные понятия

Одним из основных понятий модели является понятие *сущности*. По определению, сущность представляет собой описание некоторого объекта реального мира, который может быть четко отделен от других объектов (возможно, также представленных сущностями в модели), и его описание однозначно связано с этим реальным объектом. Такое, на вид размытое и нечеткое, определение на самом деле задает важное свойство модели: сущности должны быть различимы. Другими словами, с самого начала в модели постулируется наличие некоторого способа идентификации, позволяющего сопоставить объекты реального мира с их представлениями в базе данных. То есть предполагается идентификация сущностей по их естественным признакам (а не по суррогатным идентификаторам).

Кроме этого, требуется, чтобы описания сущностей в модели были связаны с описываемыми объектами реального мира. Как отмечено выше, из этого следует, что сущности могут быть изменяемыми, т. е. могут иметь дополнительные свойства (называемые атрибутами сущности), значения которых могут изменяться. Поскольку при этом привязка сущности к объекту реального мира изменяться не должна, сущности в модели должны иметь некоторый неизменяемый идентификатор. На практике в качестве такого идентификатора можно использовать набор из нескольких свойств реального объекта, которые позволяют его определить.

В модели «сущность-связь» предполагается, что все атрибуты сущностей имеют скалярные значения (т. е. эти значения не структурируются в рамках данной модели) и для идентификации атрибутов используются имена.

Совокупность сущностей, имеющих совпадающие (по именам) наборы атрибутов, называется *множеством сущностей*. Понятие множества сущностей является аналогом понятия класса в некоторых объектных моделях. В некоторых источниках множества сущностей называются сущностями, а их элементы называются экземплярами сущностей.

Может показаться, что такое формальное (на основе списка атрибутов) определение множеств сущностей может привести к тому, что в одном множестве окажутся сущности, представляющие реальные объекты совершенно различных типов. Однако в рамках понятий, имеющихся в модели, различить типы объектов, имеющих одинаковые наборы атрибутов, невозможно. Если различие между типами объектов существенно для решаемых задач, то оно должно быть отражено различием в списке атрибутов. Проиллюстрируем это упрощенными примерами.

Пусть в системе хранится информация о производителе и модели для автомобилей и самолетов. Это объекты разного типа, и, очевидно, можно определить по значению атрибута «производитель», что Ford — автомобиль, а Airbus — самолет. Однако, если производитель — SAAB, подобный вывод сделать будет сложнее. В действительности такой набор атрибутов описывает не автомобили или самолеты, а любые транспортные средства, поэтому различия между конкретными реальными объектами несущественны. Но если включить, например, атрибут «максимальная высота полета», то различие становится видимым на уровне модели данных.

Рассмотрим другой пример. Пусть в информационной системе необходимо хранить информацию о поставщиках и потребителях продукции предприятия. По-видимому, поставщики и потребители будут характеризоваться одним и тем же набором атрибутов, описывающим (юридические или физические) лица. Немного более тщательное проектирование показывает, что одно и то же лицо может быть как поставщиком, так и потребителем, поэтому объединение всех партнеров в одном множестве сущностей вполне оправдано.

Другим ключевым понятием модели является понятие *связи*. По определению, связь представляет собой упорядоченную последовательность сущностей, имеет свою идентификацию и может иметь свои собственные атрибуты. Связи между сущностями из одних и тех же множеств, имеющие идентификацию одного типа и совпадающие (по именам) наборы атрибутов, составляют множество связей.

Например, сущности студент и дисциплина могут быть связаны связью экзамен. При этом упорядоченным набором сущностей будет (студент, дисциплина). Эта

же пара будет идентифицировать связь, а дополнительными атрибутами связи могут быть оценка и дата ее получения.

Структуры данных, описываемые в рамках модели ER, принято представлять двумерными диаграммами. В классической модели ER сущности изображаются прямоугольниками, связи — ромбами, атрибуты — овалами. Такие диаграммы, однако, получаются слишком громоздкими, поэтому описания атрибутов обычно включаются в тот элемент диаграммы, к которому они относятся, или вовсе не включаются, в особенности на начальных фазах проектирования.

Простейший пример диаграммы «сущность-связь» с использованием ромба для обозначения связи показан на рис. 2.3.1.

Рис. 2.3.1. Простейшая диаграмма «сущность-связь»

Ограничения целостности

Наиболее распространенным видом связей являются бинарные, т. е. такие, которые связывают две сущности. Для них можно определить ограничения, показывающие, как именно сущности могут быть взаимосвязаны. Используются следующие символы:

1 — в связи может участвовать и должна участвовать одна сущность;

0 — в связи может участвовать не больше, чем одна сущность;

m, n — в связи может участвовать нуль или несколько сущностей из одного множества.

Ограничения целостности описываются парой таких символов, разделенных двоеточием.

Например, если каждый сотрудник некоторого предприятия обязательно является сотрудником одного отдела, такое ограничение будет выражено записью 1 : n (читается «один ко многим»). При этом n обозначает, что несколько сотрудников могут быть связаны с одним отделом, а 1 — что каждый сотрудник обязательно связан с каким-нибудь отделом и только с одним.

Точно так же связь между аэропортом отправления и рейсом подчиняется ограничению 1 : n, потому что каждый рейс обязательно отправляется из какого-нибудь одного аэропорта. С другой стороны, связь между пассажиром и постоянным клиентом подчиняется ограничению 0 : n, потому что постоянный клиент может быть пассажиром нескольких рейсов (все-таки он — постоянный клиент и, скорее всего, пользуется услугами авиаперевозчика неоднократно), но пассажир не обязательно должен быть связан с постоянным клиентом.

Наиболее сложным типом бинарных связей являются связи типа m : n (читается «многие ко многим»). Например, сотрудник может участвовать в нескольких проектах, и в каждом проекте может участвовать несколько сотрудников.

Наследование

Объекты реального мира могут одновременно входить в состав нескольких множеств. Так, любой человек может быть (а может не быть) работником какого-либо предприятия. Подобные ситуации представляются в модели «сущность-связь» с помощью понятия наследования, которое представляется связью *is a*. Эта связь всегда является бинарной связью 1 : 0, и дополнительно требуется, чтобы идентификаторы связанных сущностей совпадали. Например, если каким-либо способом в базе данных найден студент, то во множестве персон обязательно должна быть сущность с таким же идентификатором и обязательно связанная с найденной сущностью во множестве студентов. Неформально можно это выразить утверждением: *студент является человеком*. Заметим, что во многих объектных моделях данных такое требование отсутствует.

Модель «сущность-связь» предписывает также, что при наследовании атрибуты более широкого множества не дублируются в сущностях более узкого. Например, если сущность персона имеет атрибут дата рождения, то такого атрибута не должно быть у сущности студент, так как его можно получить из сущности более широкого множества.

Это правило, на первый взгляд, определяет требования к организации хранения, что несколько странно для модели концептуального уровня. Однако смысл данного ограничения состоит в том, что значения атрибутов могут наследоваться, но не могут изменяться при наследовании. Например, рост и вес человека будут одинаковыми, независимо от того, рассматриваем мы этого человека как студента или нет.

Условие на тождественность идентификаторов сущностей накладывает ограничение на множественное наследование: оно возможно только в том случае,

если имеется общее множество, из которого наследуют оба наследуемых множества (правило «ромба»). Например, множество работающих студентов может наследовать свойства студентов и свойства сотрудников, потому что и студенты, и сотрудники являются специализациями более общей сущности «человек» и, следовательно, все четыре множества могут иметь общий идентификатор.

Далее в главе 8 мы увидим, что реализация наследования в системе PostgreSQL отличается от наследования в модели данных «сущность-связь».

Отображение в реляционную модель

В модели «сущность-связь» не предусмотрены операции манипулирования данными. Для того чтобы описание данных можно было использовать, необходимо на его основе построить описание данных в другой модели.

Отображение в реляционную модель данных строится следующим образом:

1. Для каждого множества сущностей строится отношение, атрибутами которого становятся идентификатор и все атрибуты, имеющиеся у сущностей, входящих в это множество.
2. Определяются функциональные зависимости атрибутов от идентификатора для каждого построенного отношения.
3. Для каждого множества связей строится отношение, атрибутами которого становятся идентификатор и все атрибуты связи. Напомним, что идентификатор связи содержит все идентификаторы связываемых сущностей, поэтому в отношении идентификатору связи будет соответствовать несколько атрибутов.
4. Определяются функциональные зависимости атрибутов, полученных из атрибутов связи, от атрибутов, полученных из идентификатора связи.

Если другие функциональные зависимости не определяются, то полученные отношения будут в 3NF. Если же дополнительные функциональные зависимости удалось обнаружить, то полученная схема не будет нормализованной и, возможно, понадобится дополнительная нормализация. Заметим, однако, что наличие таких функциональных зависимостей может указывать на ошибки в проектировании модели «сущность-связь» (возможно, некоторые сущности на самом деле являются агрегатами, состоящими из более мелких сущностей).

Рассмотрим пример.

При перевозке грузов с каждым заказом на перевозку связаны отправитель, получатель, плательщик и, возможно, еще несколько юридических или физических лиц. Поскольку каждое из этих лиц может выступать как отправителем, так и получателем (для разных грузов), все сущности, которые могут выступать в одном из этих качеств, объединяются в одно множество сущностей, которое будет называться party. Конечно, эти сущности имеют много разнообразных атрибутов, но пока мы ограничимся только идентификатором и именем.

Из атрибутов заказа (order) пока включим только вес. И наконец, атрибутом связи между заказом и партией будет, кроме идентификаторов связываемых сущностей, роль партии в этой связи (отправитель, получатель и т. п.).

Диаграмма для этого примера показана на рис. 2.3.2.

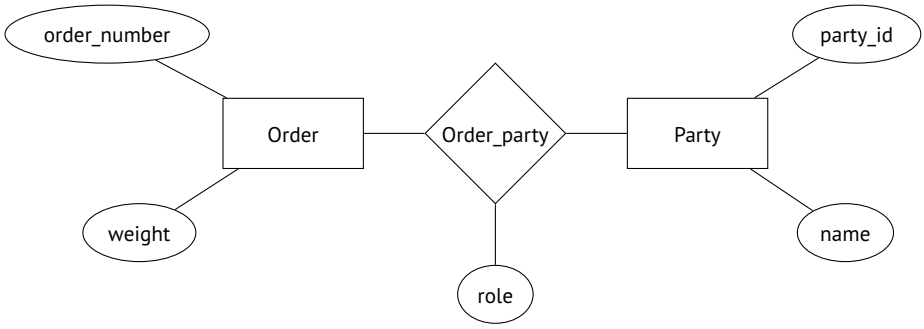


Рис. 2.3.2. Связь груза с партиями

В результате отображения будет получена схема, показанная на рис. 2.3.3.

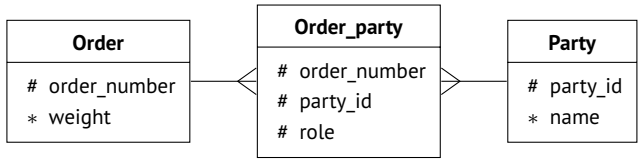


Рис. 2.3.3. Представление связи m : n

При этом ключами в отношениях order и party будут order_number и party_id соответственно, а ключом отношения order_party, представляющего связь между партиями и заказами, будет совокупность всех трех его атрибутов, поскольку

одна и та же партия может выступать по отношению к грузу в нескольких ролях одновременно: например, отправитель может быть плательщиком.

Множество связей, на которые наложено ограничение 1 : n, можно отобразить более простым способом. Дело в том, что ограничение такого вида определяет функциональную зависимость идентификатора одной сущности от идентификатора другой. Действительно, пусть множества сущностей R_1 и R_n с ключами k_1 и k_n связаны связью E, которая подчиняется ограничению 1 : n. Ключом для связи будет пара ключей связываемых отношений, однако ключ k_n однозначно определяет ключ k_1 в силу ограничения целостности.

Поэтому атрибуты связи могут зависеть только от идентификатора второй сущности k_n и могут быть включены в отношение, представляющее при отображении множество сущностей R_n вместе с идентификатором из первого множества R_1, и для представления связи отдельная таблица не нужна.

В практически используемых инструментах для построения диаграмм модели данных «сущность-связь» не предусмотрена возможность определения связей m : n, поэтому для представления таких связей необходимо определять дополнительные сущности и две связи типа 1 : n.

Ключ другого отношения, функционально зависящий от ключа рассматриваемого отношения и включенный в состав его атрибутов, называется *внешним ключом* (foreign key).

2.3.2. Концептуальные объектные модели

Современные варианты модели «сущность-связь» содержат большое количество разнообразных возможностей для описания структур данных и особенностей их взаимосвязей, однако не позволяют описывать какие-либо операции над этими структурами данных. По-видимому, эта особенность модели связана с тем, что в период ее создания предполагалось, что операции обработки данных, выполняемые в приложении, должны быть отделены от схемы базы данных и поэтому их следует проектировать отдельно другими средствами.

Зачастую, однако, эта особенность модели рассматривается как недостаток, который преодолевается в рамках объектно-ориентированного подхода к проектированию систем. В настоящее время доминирующим средством объектно-ориентированного проектирования является унифицированный язык моделирования (Unified Modeling Language, UML).

В процессе моделирования с использованием UML одновременно создаются диаграмма классов приложения и соответствующая ей модель базы данных. Хотя формально никаких ограничений на структуры данных этот язык сам по себе не накладывает, однако на практике зачастую получаются схемы баз данных, несколько уступающие по качеству схемам, получаемым при использовании модели «сущность-связь». Это вызвано в первую очередь необходимостью компромиссов для упрощения отображения схемы базы данных в структуры данных приложения (и обратно). Некоторые дополнительные ограничения на структуры базы данных возникают при использовании систем, реализующих объектно-реляционные отображения (ORM), или специализированных систем генерации приложений (frameworks).

2.4. Объектные и объектно-реляционные модели данных

В связи с расширением спектра областей применения систем управления базами данных предположения, в которых разрабатывались ранние реляционные СУБД, оказались слишком ограничительными. Поэтому начиная с середины 80-х годов исследуются различные расширения модели, позволяющие более непосредственно отобразить специализированные структуры хранения на логическом уровне. В рамках этого направления рассматривались, например, отношения не в первой нормальной форме (Non-First-Normal-Form, NFNF, NF^2, nested relations) и другие варианты структурирования данных, допускающие более сложные типы атрибутов, чем скаляры.

В дальнейшем это направление привело к созданию языков программирования с постоянным хранением данных (persistent programming languages) и объектных моделей баз данных. И в том, и в другом случае в базе данных размещаются объекты, определяемые в рамках объектной модели языка программирования. Этот подход позволил добиться очень высокой эффективности доступа в режиме навигации: в лучших системах время доступа к постоянно хранимым объектам только в 3–4 раза превышало время доступа к оперативной памяти.

Ожидалось, что объектные базы данных вытеснят все остальные классы моделей данных, однако по ряду причин этого не произошло:

- база данных может использоваться только клиентами, написанными на одном языке программирования;
- не удается создать высокоуровневый декларативный язык запросов;

- не удается обеспечить высокую производительность при массовой выборке данных.

В итоге системы, основанные на чисто объектных моделях данных, заняли относительно небольшой сегмент в многообразии применений СУБД, но некоторые объектные средства стали составной частью систем управления базами данных общего назначения. В настоящее время все высокопроизводительные системы, в том числе PostgreSQL, реализуют объектные расширения, и поэтому их принято называть объектно-реляционными.

Наиболее важными видами объектных расширений можно считать:

- возможность определения пользовательских типов данных, в том числе структурных;
- использование коллекций объектов.

Средства определения типов данных, по сути, заложены в концепции абстрактного домена, поэтому можно сказать, что возможность определения пользовательских типов данных скорее снимает ограничения ранних реализаций, чем расширяет теоретическую реляционную модель данных, по крайней мере если ограничиваться скалярными типами. В дополнение к скалярным пользовательским типам обычно объектные расширения включают возможности создания структурных типов. Примерами структурных типов могут быть геометрические объекты (точки, прямые и т. д.).

Коллекцией называется набор объектов определенного типа. Различают следующие разновидности коллекций:

set — набор объектов, не содержащий дубликатов, т. е. аналог реляционного отношения;

bag — неупорядоченный набор объектов, в котором могут быть дубликаты;

list — упорядоченный список объектов;

array — набор объектов с доступом по индексу или индексам.

Любая коллекция может быть значением атрибута. Существуют функции, преобразующие коллекции в виртуальные таблицы, а также позволяющие записывать результаты выполнения запросов в качестве значений коллекций.

Возможности определения и использования коллекций в системе PostgreSQL обсуждаются далее в главе 8.

2.5. Другие модели данных

2.5.1. Слабоструктурированные модели данных

В некоторых классах приложений отделение описания структуры данных (т. е. схемы) от самих данных оказывается нежелательным. Например, при передаче документов по сети целесообразно определение структуры документа пересылать вместе с документом. Поскольку в подобных случаях значительная часть данных представляет собой текст на естественном языке, такие данные принято называть *слабоструктурированными* (semi-structured; некоторые авторы упрямо называют такие данные полуструктурированными). Наиболее широко используемым форматом для представления слабоструктурированных данных является XML, довольно часто употребляется также JSON, популярность которого стремительно растет в последние годы.

Хотя ни XML, ни JSON не предполагалось использовать для хранения данных, оба этих формата можно рассматривать как модели данных. Широко распространено мнение о том, что эти модели предоставляют возможности для более гибкого описания и представления данных, чем реляционная модель, однако на самом деле схемы XML (XSD) предоставляют средства для описания не менее жестких ограничений целостности, чем реляционная модель данных.

В объектно-реляционных базах слабоструктурированные данные могут храниться как значения атрибутов отношений (таблиц). В системе PostgreSQL для этого используются типы данных xml, json и jsonb, более детально обсуждаемые в главе 8. Как и для типов коллекций, имеются встроенные функции, позволяющие формировать значения слабоструктурированных типов из табличных данных и, наоборот, извлекать элементы слабоструктурированных значений в виде коллекций. Это дает возможность сочетать реляционные и слабоструктурированные языки запросов (в частности, XPath и XQuery, а также SQL/JSON path, введенный в стандарте SQL 2016).

2.5.2. Модели для представления знаний

В системах представления знаний (таких как семантические сети и онтологии), а также для представления графов часто используется тернарная модель данных, в которой элементарной структурой является тройка вида *(объект, атрибут, значение)*. При этом для записи каждого объекта необходимо столько строк, сколько атрибутов имеет этот объект.

В таком представлении схема (перечень атрибутов) неотделима от данных, что обеспечивает возможность хранения любых объектов без какого-либо предварительного описания их структуры. Это дает очень большую гибкость представления, которая привлекает многих разработчиков.

С другой стороны, применение тернарного представления существенно усложняет функции приложения и очень существенно влияет на его производительность.

2.5.3. Ключ-значение

В последнее десятилетие получили довольно широкую известность системы, предназначенные для хранения пар «ключ-значение». При этом предполагается, что поиск данных возможен только по (первичному) ключу, а интерпретация значения выполняется в приложении. Привлекательной стороной таких систем является простота начального запуска, однако практически все функции, обычно выполняемые системами управления базами данных, в том числе

- сложные структуры данных,
- взаимосвязи между объектами данных,
- ограничения целостности,
- высокоуровневые декларативные запросы,
- поиск по значениям неключевых атрибутов

и другие, должны реализовываться в коде приложения, что существенно усложняет разработку приложений или приводит к снижению качества.

Многие системы, первоначально позиционировавшиеся как системы этого типа, эволюционируют в направлении включения более сложных возможностей, приближающих их к более развитым СУБД, вплоть до реализации полноценных декларативных языков запросов.

2.5.4. Устаревшие модели данных

Во многих учебниках по базам данных до сих пор рассматриваются ранние модели данных, использовавшиеся в 70–80-е годы. Наиболее известной из этих моделей является сетевая, язык описания данных для которой разрабатывался комитетом CODASYL, а наиболее широко применявшейся была иерархическая модель данных, реализованная в системе IMS компании IBM. Обе эти модели предоставляют возможности навигационного доступа к отдельным объектам и возможности описания взаимосвязей между ними, однако в иерархической модели представление для клиентской программы всегда является деревом (хотя в самой базе данных возможно описание и хранение любых видов взаимосвязей).

Возможности сетевой модели данных полностью перекрываются объектными и объектно-реляционными моделями данных, а возможности иерархической — средствами XML и JSON.

2.6. Примеры проектирования схемы в модели «сущность-связь»

Варианты модели данных «сущность-связь», используемые на практике и реализованные во многих инструментах проектирования баз данных, значительно отличаются от представления модели, излагаемого в учебниках (в том числе от варианта, кратко представленного выше).

Наиболее важное отличие состоит в том, что исключаются любые связи между более чем двумя сущностями, а также связи типа «многие ко многим». Поскольку для бинарных связей типа 1 : n или 0 : n атрибуты связи можно перенести в одну из связываемых сущностей (выше показано, как это делать, при описании отображения в реляционную модель), и поскольку нет необходимости в явной идентификации связей, обозначения связей ромбами становятся ненужными. Связи обозначаются линиями, соединяющими сущности (прямоугольники).

Для описания в базе данных ситуаций, в которых требуются связи «многие ко многим», необходимо вводить дополнительные сущности, выполняющие роль таких связей.

На рис. 2.3.3 показан фрагмент подобной диаграммы, на котором множественная связь заменена на сущность. Аналогично множественная связь экзамен между множествами курсов и студентов может быть заменена на сущность.

На концах линии, представляющей бинарную связь, указывается кратность связи. Существуют различные системы нотации для кратностей, при этом многие инструментальные средства предоставляют возможность выбора нотации.

Развитые реализации модели «сущность-связь» предоставляют целый ряд различных дополнительных понятий, позволяющих более детально и более точно описывать взаимосвязи между сущностями.

Одним из таких понятий является понятие *слабой сущности*. В отличие от обычных (сильных) сущностей, слабая сущность может существовать в базе данных, только если она связана с некоторой *сильной сущностью*. Например, в базе данных предприятия может содержаться информация о детях сотрудников, необходимая для расчета каких-либо льгот или выплат, предусмотренных законодательством, или, может быть, просто для рассылки новогодних подарков. Однако эти сущности (дети сотрудников) не имеют смысла для предприятия, если сотрудники увольняются. Возможно, конечно, что такая информация должна сохраняться в качестве архивной вместе с информацией о лицах, ранее работавших на предприятии, однако это совсем другой аспект проектирования базы данных.

Рассмотрим фрагмент схемы базы данных, показанный на рис. 2.6.1 и предназначенный для хранения информации, имеющейся на авиабилетах. Конечно, этот фрагмент значительно проще, чем реальные схемы, пригодные для данной цели. Многие важные стороны процесса перевозки пассажиров в этом варианте учесть невозможно, однако даже такая упрощенная схема дает возможность показать использование различных типов связей.

Основной сущностью в этой схеме является *бронирование* (bookings). В момент приема заказа создается уникальный номер бронирования book_ref, который никогда не изменяется и поэтому может служить в качестве идентификатора.

В каждое бронирование можно включить несколько пассажиров, для каждого из которых создается отдельный *билет* (tickets). Пассажир не выделен в отдельную сущность; его идентификационная информация (номер документа и имя) записывается в билет. Наша схема не дает возможности узнать, являются ли разные пассажиры одним физическим лицом.

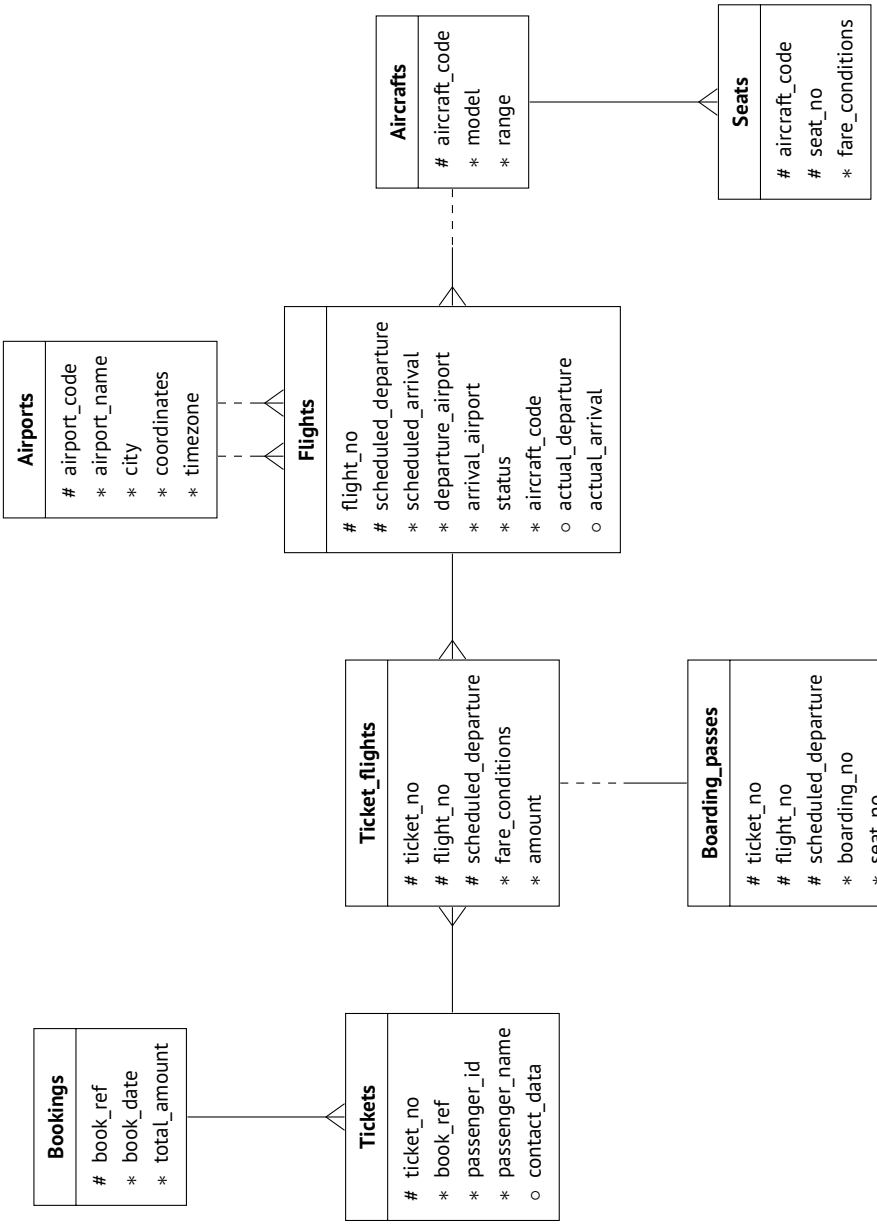


Рис. 2.6.1. Фрагмент схемы базы данных для хранения авиабилетов

Билет может содержать несколько *перелетов* (ticket_flights), соответствующих прямому и обратному рейсам, а кроме того, перемещение в каждом направлении может включать пересадки. Каждый перелет связан с бронированием и с *рейсом* (flights), поэтому в качестве идентификатора перелета используется комбинация из идентификатора бронирования и идентификатора рейса.

Рейс идентифицируется своим номером и датой вылета по расписанию. Он связан с двумя *аэропортами* (airports) отправления и прибытия. Каждый аэропорт имеет уникальный трехбуквенный код (например, SVO или LED).

При регистрации на рейс каждому пассажиру выдается *посадочный талон* (boarding_passes), содержащий номер места в качестве атрибута (в дополнение к необходимой идентификации).

Количество *мест* (seats) в кабине зависит от модели *самолета* (aircrafts), совершающего рейс. Предполагается, что каждая модель имеет только одну компоновку салона.

Заметим, что в этом примере использованы только естественные ключи, т. е. все значения, использованные для идентификации, применяются в реальных бизнес-процессах и никак не зависят от информационной системы. Такое проектное решение обеспечивается тем, что в реальности фактически применяются неизменяемые идентификаторы, обеспечивающие уникальность, поэтому введение дополнительных суррогатных ключей привело бы к некоторой избыточности данных. С другой стороны, это приводит к появлению составных первичных ключей: в нашем примере ключи перелета и посадочного талона содержат по три атрибута.

Отметим также, что на самом деле наши таблицы не нормализованы. Так, в таблице рейсов аэропорты отправления и прибытия зависят только от номера рейса, но не зависят от даты и времени вылета, т. е. в этой таблице имеется зависимость от неполного ключа, и поэтому она не находится во второй нормальной форме. Конечно, этот недостаток модели легко устранить, выделив еще одну сущность — *маршрут,* который идентифицируется номером рейса. Однако особенности использования этой таблицы и ее небольшие размеры делают влияние аномалий ненормализованных схем не очень существенным. Поскольку таблица невелика по размеру, избыточность тоже невелика, коды аэропортов практически никогда не изменяются, поэтому аномалии обновления также не могут повлиять на работу системы.

На рис. 2.6.2 показан вариант схемы базы данных авиабилетов, в котором во всех таблицах использованы суррогатные ключи.

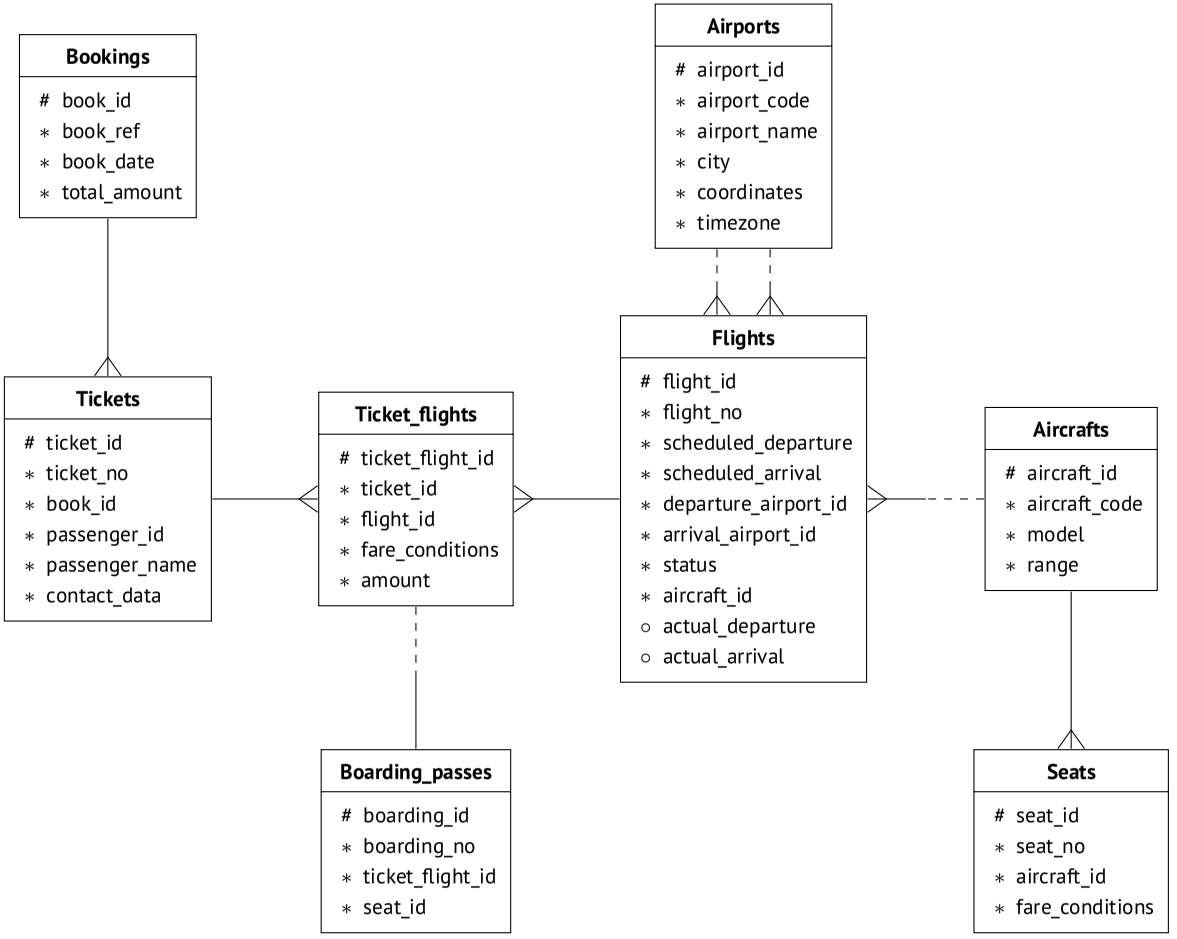


Рис. 2.6.2. Фрагмент схемы базы данных для хранения авиабилетов с суррогатными ключами

Можно заметить, что эта схема обладает некоторыми недостатками, по сравнению со схемой, показанной на рис. 2.6.1. Так, при извлечении данных из таблицы рейсов для получения кодов аэропортов, которые понятны любому специалисту без какой-либо дополнительной расшифровки, необходим доступ к таблице аэропортов. Зависимость от неполного ключа превратилась в транзитивную зависимость (идентификаторы аэропортов отправления и прибытия зависят от номера рейса, однако для нормализации потребуется ввести еще один суррогат для маршрута). Поскольку коды аэропортов и номера бронирования являются в действительности уникальными и неизменяемыми, использование суррогатов в этих таблицах создает транзитивные зависимости. Устранение этих зависимостей привело бы к появлению сущностей, которым не соответствуют никакие объекты реального мира.

Необходимо, однако, заметить, что в подобных случаях транзитивные зависимости не могут вызвать отрицательные эффекты, связанные с аномалиями (кроме избыточности), потому что такие атрибуты, как номер бронирования, обладают уникальностью и неизменяемостью.

При проектировании схемы базы данных необходимо учитывать ограничения, накладываемые на структуру базы данных средой разработки приложений. При этом неизбежны компромиссы, которые приводят к ухудшению качества схемы базы данных. Одним из часто встречающихся компромиссов такого рода является использование искусственных (суррогатных) ключей для всех таблиц базы данных. Это может быть необходимо, если среда разработки приложения не допускает использование никаких идентификаторов объектов, кроме целочисленных. Повсеместное применение суррогатов может приводить к появлению фактических дубликатов (различающихся только искусственными идентификаторами) и, как следствие, к трудно обнаруживаемым ошибкам в работе приложения.

Сторонники тотального использования суррогатных ключей указывают на следующие преимущества этого подхода к проектированию схемы:

- гарантируется уникальность идентификаторов объектов, так как они генерируются в системе;
- гарантируется неизменяемость идентификаторов, нет необходимости в тщательном анализе предметной области для выявления неизменяемых естественных идентификаторов;
- исключается необходимость использования составных ключей.

На рис. 2.6.3 показана схема демонстрационной базы данных, которая будет использоваться в большинстве примеров и упражнений в этом курсе. При проектировании этой схемы были приняты компромиссные решения: в большинстве отношений использованы естественные ключи, но для некоторых оказалось целесообразно определить суррогатные.

Кроме хранимых отношений, эта схема содержит представление для маршрутов, которое можно использовать в запросах наравне с хранимыми отношениями и которое содержит данные, вычисляемые на основе данных из хранимых отношений.

2.7. Библиографические комментарии

Понятия, связанные с навигационной обработкой данных, и основы сетевой модели данных собраны в статье [1], автор которой был позже удостоен премии Тьюринга (русский перевод в [22]). Опыт использования иерархической системы управления базами данных IMS/360 обобщен в [10].

Основы реляционной модели данных представлены широкой аудитории в [8], наиболее часто цитируемой статье в компьютерной литературе. Несколько более сложный вариант этой модели данных представлен в [9].

Различные подходы к нормализации реляционных схем обсуждаются и сопоставляются в [11]; работа [2] описывает алгоритмы синтеза реляционных схем по функциональным зависимостям. Обстоятельная книга [14] (русский перевод [19]) содержит изложение теории реляционных баз данных. Все статьи и книга, перечисленные в этом разделе, требуют достаточно серьезной математической подготовки.

Модель данных «сущность-связь» впервые представлена в [7]. За этой статьей последовало огромное количество исследований и практических разработок методов и инструментов проектирования схем реляционных баз данных.

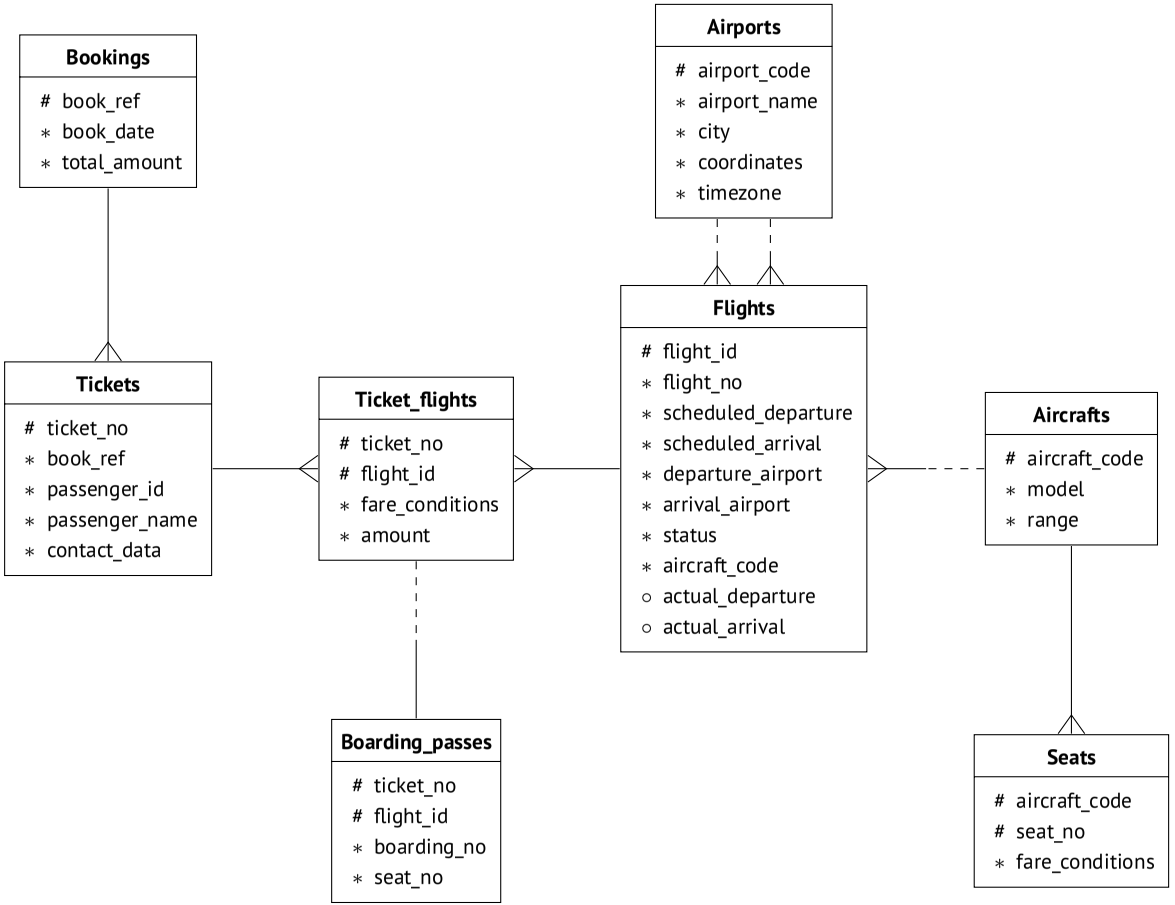


Рис. 2.6.3. Схема демонстрационной базы данных

2.8. Упражнения

Упражнение 2.1. Докажите, что операции UNION, INTERSECT, PROD, JOIN коммутативны.

Упражнение 2.2. Докажите, что операции UNION, INTERSECT, PROD, JOIN ассоциативны.

Упражнение 2.3. Докажите, что выполняются дистрибутивные законы для PROD и JOIN относительно UNION и INTERSECT, а также для UNION относительно INTERSECT, INTERSECT относительно UNION.

Упражнение 2.4. В схеме с курсами и студентами предусмотрите возможность ведения занятий по курсу несколькими преподавателями. Экзамен может сдаваться не тому преподавателю, который вел занятия.

Упражнение 2.5. Укажите отклонения от третьей нормальной формы в демонстрационной базе данных.

Упражнение 2.6. В схемах с авиаперевозками аэропорт может иметь несколько терминалов, названия терминалов уникальны внутри аэропорта. Каждый рейс отправляется с определенного терминала аэропорта отправления и прибывает на определенный терминал аэропорта прибытия. Внесите дополнения в схему, позволяющие хранить эту информацию. Во многих аэропортах имеется только один терминал, это не должно быть особым случаем при выполнении запросов.

Упражнение 2.7. Авиаперевозчики могут регистрировать постоянных клиентов. Если при бронировании указывается карточка постоянного клиента, информация о нем может быть скопирована в бронирование, но может быть и изменена. Добавьте в схему демонстрационной базы возможность хранения данных о постоянных клиентах.

Упражнение 2.8. Багаж пассажира может опаздывать на рейс при пересадке и доставляться отдельно от пассажира другим рейсом. Добавьте отношения для представления информации о багаже.

Упражнение 2.9. Аэропорт может быть основным аэропортом для нескольких близлежащих населенных пунктов. Внесите изменения в схему, позволяющие описать такие ситуации.

Упражнение 2.10. Создайте схему базы, которую можно использовать для хранения исторических данных о погоде (температура, влажность, скорость ветра, время суток и т. п.).

Упражнение 2.11. Создайте схему базы с информацией о наличии автомобилей в автомобильных салонах. Необходимо учитывать марку автомобиля, модель, год выпуска, адрес салона и т. п.

Глава 3

Знакомимся с базой данных

3.1. Установка базы данных

Для того чтобы продолжить освоение материала, представленного в этой книге, необходимо установить систему управления базами данных PostgreSQL (или получить доступ к уже установленной). Система PostgreSQL может работать под управлением любой распространенной операционной системы практически на любых компьютерах.

Действия, необходимые для установки, зависят от операционной системы и детально описаны на сайте PostgreSQL https://www.postgresql.org/download.

Также можно воспользоваться специально подготовленным образом операционной системы Linux с предустановленной СУБД PostgreSQL. Образ доступен по адресу https://edu.postgrespro.ru/DBTECH-student.ova и может быть импортирован в систему виртуализации, такую как VirtualBox или VMWare.

Почти все примеры и упражнения в первой части этой книги используют *демонстрационную базу данных* системы PostgreSQL. В подготовленный образ эта база уже включена, а в случае самостоятельной установки нужно дополнительно скачать один из доступных вариантов демобазы и загрузить его в свою систему. Как это сделать, детально описано на странице сайта компании Postgres Professional https://postgrespro.ru/education/demodb. Там же вы найдете и полное описание схемы этой базы данных.

3.2. Подключение к серверу базы данных

Напомним, что базой данных (БД) называется совокупность данных, доступная для приложения через сервер базы данных. В этой главе мы будем предполагать, что приложение выступает в роли клиента и взаимодействует непосредственно с сервером базы данных. В более сложных (многослойных) архитектурах эту роль выполняют компоненты, непосредственно взаимодействующие

с СУБД (например, часть системы, работающая на сервере приложений), но в этой главе рассматривается только взаимодействие с сервером базы данных непосредственно.

Для того чтобы выполнять какие-либо действия над данными, находящимися в БД, приложение должно установить соединение с сервером базы данных. Различные клиенты (приложения) могут устанавливать соединение по-разному, однако обычно для этого требуется следующая информация:

- сетевой адрес сервера базы данных (может быть именем или IP-адресом);
- номер порта, через который производится соединение;
- имя базы данных;
- имя пользователя базы данных.

Значения этих параметров могут быть записаны в конфигурационных файлах клиента и поэтому не обязательно указываются явно, однако они необходимы в любом случае.

Сетевой адрес сервера и порт определяют, куда передаются сообщения, адресованные серверу базы данных. Большинство СУБД имеет зафиксированный предпочтительный номер порта, который используется, если администратор не изменит его при создании сервера. Для СУБД PostgreSQL это порт 5432. Использование другого номера порта необходимо, если на одном компьютере одновременно работает несколько серверов баз данных. Конечно, сервер и клиент базы данных тоже могут работать на одном и том же компьютере, однако для способа установки соединения это не имеет значения, потому что и в этом случае используется сетевой протокол (исключение составляют локальные подключения через сокеты домена Unix, поддерживаемые некоторыми операционными системами).

Один сервер баз данных может обрабатывать запросы к нескольким базам данных. Совокупность баз данных, обслуживаемых одним сервером, в системе PostgreSQL называется *кластером*. (Термин «кластер» используется также во многих других контекстах и может обозначать совокупность компьютеров, совокупность близких точек в методах анализа данных, способ размещения коллекций данных на носителях и имеет ряд других значений, непосредственно с компьютерами не связанных.)

Некоторые объекты в системе PostgreSQL, например пользователи, определены для всего кластера, однако в рамках одного соединения клиентская программа может работать только с одной базой данных. Имя этой базы указывается при установлении соединения. В простых конфигурациях (например, для серверов баз данных, запускаемых на компьютере клиента) часто достаточно одной базы данных. Как мы увидим далее, ничто не мешает хранить данные нескольких независимых приложений в одной базе. Это упрощает совместное использование данных разных приложений в рамках одного сеанса. В то же время практика создания отдельных баз данных для каждого небольшого приложения широко распространена.

Понятие *пользователь* (user) имеет много различных значений. Нам будет нужно различать пользователя операционных систем на компьютерах сервера и клиента, пользователя базы данных и пользователя приложения. Эти понятия часто существенно отличаются. Например, если приложение работает как веб-приложение (такое как интернет-магазин), то пользователем приложения становится любой покупатель, однако такой пользователь не будет зарегистрирован как пользователь базы данных.

В некоторых случаях может быть удобно использовать одинаковые имена для обозначения разных объектов, например пользователи операционной системы или домена могут быть зарегистрированы как пользователи базы данных. Но в любом случае эти сущности остаются различными.

Понятие пользователя базы данных важно для реализации разграничения доступа и защиты данных на уровне БД. Права доступа как к базе данных в целом, так и к отдельным объектам определяются именно для пользователей БД. Управление доступом к объектам базы данных более детально обсуждается в главе 5.

3.3. Простой клиент: psql

Для того чтобы работать с базой данных, необходима программа, выполняющая функции клиента. В примерах этой книги в качестве такой программы будет использоваться `psql`. Эта программа входит в состав любого комплекта PostgreSQL в любой операционной системе. Для запуска программы `psql` нужно с командной строки операционной системы ввести команду

```
psql -d имя_бд -U имя_пользователя -h сервер -p порт
```

Возможно, в ответ на подсказку системы потребуется ввести пароль. В некоторых комплектах эту программу можно запускать и другим способом.

Часто для соединения с локальным сервером базы данных (т. е. работающим на том же компьютере, что и psql) не требуется указывать вообще никаких параметров. Например, при использовании образа ОС с предустановленной системой PostgreSQL команда подключения к демобазе (которая называется demo) выглядит как

```
psql -d demo
```

Если не указать и имя базы данных, то в установленном соединении будет использоваться БД, имеющая имя postgres.

Программа psql предоставляет интерфейс типа командной строки. Существуют другие клиентские программы, предоставляющие более развитый экранный интерфейс, который дает некоторые преимущества для быстрой ориентации в структуре базы данных, однако для использования сколько-нибудь сложных конструкций SQL в любом случае необходимо текстовое представление.

В командной строке psql можно вводить операторы SQL, которые необходимо завершать точкой с запятой «;», и команды программы psql, начинающиеся с символа обратной косой черты «\». Список команд с краткими описаниями того, что они делают, можно получить по команде \?.

Язык SQL, предназначенный для работы с базой данных и предоставляющий мощные средства для выполнения любых операций в базе данных, обсуждается в главе 4. Поэтому здесь мы рассмотрим только некоторые из команд psql, которые позволяют бегло ознакомиться с тем, что находится в базе данных.

Каждый объект базы данных создается в некоторой схеме. Для небольших баз данных можно использовать всего одну схему, которую можно не указывать при ссылках на объекты базы данных при выполнении операций, однако такая практика считается недопустимой в серьезных промышленных разработках. Для баз данных сложной структуры схемы позволяют объединить объекты (например, относящиеся к одному приложению) и дают ряд дополнительных возможностей, в основном связанных с использованием имен объектов. Схемы можно также применять для разграничения доступа. Такие возможности обсуждаются в главе 5.

Основным объектом логического уровня в базах данных, построенных на основе реляционной модели данных, являются двумерные *таблицы*. При этом

строки (кортежи) описывают значения различных атрибутов одной сущности, а *колонки* — значения одного атрибута, принадлежащие разным сущностям.

Кроме таблиц, в базе данных могут быть определены другие объекты логического уровня: типы данных, домены, представления (view), процедуры, триггеры и др. Многие из таких объектов не содержат данных.

В программе psql имеется большой набор команд, которые позволяют вывести информацию об объектах базы данных. Обычно в качестве параметра таких команд можно указать имя или шаблон, которому должны удовлетворять имена объектов, которые будут обработаны такой командой.

Приведем в качестве примера результат выполнения команды \dt, которая выводит список таблиц в схеме bookings демонстрационной базы данных:

```
demo=# \dt bookings.*
              List of relations
  Schema  |      Name       | Type  |  Owner
----------+-----------------+-------+---------
 bookings | aircrafts_data  | table | student
 bookings | airports_data   | table | student
 bookings | boarding_passes | table | student
 bookings | bookings        | table | student
 bookings | flights         | table | student
 bookings | seats           | table | student
 bookings | ticket_flights  | table | student
 bookings | tickets         | table | student
(8 rows)
```

Более детальную информацию об объектах базы данных можно получить с помощью команды \d:

```
demo=# \d bookings.aircrafts_data
                Table "bookings.aircrafts_data"
    Column     |     Type     | Collation | Nullable | Default
---------------+--------------+-----------+----------+---------
 aircraft_code | character(3) |           | not null |
 model         | jsonb        |           | not null |
 range         | integer      |           | not null |
Indexes:
    "aircrafts_pkey" PRIMARY KEY, btree (aircraft_code)
Check constraints:
    "aircrafts_range_check" CHECK (range > 0)
Referenced by:
    TABLE "flights" CONSTRAINT "flights_aircraft_code_fkey"
      FOREIGN KEY (aircraft_code) REFERENCES
        aircrafts_data(aircraft_code)
    TABLE "seats" CONSTRAINT "seats_aircraft_code_fkey"
      FOREIGN KEY (aircraft_code) REFERENCES
        aircrafts_data(aircraft_code) ON DELETE CASCADE
```

Для таблиц команда \d выводит список столбцов (атрибутов) вместе с их типами и ограничением NOT NULL, а также другие ограничения целостности, определенные для этой таблицы.

Существуют другие программы-клиенты, большинство из которых использует графические средства для отображения содержимого базы данных на экране: деревья для представления структуры вложенных объектов, решетки (таблицы) и т. п. В качестве примера такой программы можно назвать pgAdmin.

Тем не менее какая-либо обработка данных, извлеченных из БД, предполагает использование программы-клиента, написанного для решения конкретных прикладных задач. Разработка приложений и организация их взаимодействия с базой данных рассматривается в главе 7.

3.4. Итоги главы

Начиная с этой главы для эффективного освоения материала книги необходимо установить СУБД PostgreSQL и демонстрационную базу данных. В начале главы указывается, каким образом получить детальные инструкции по установке. Далее показано, каким образом можно познакомиться с логическими структурами, имеющимися в базе данных, используя программу psql, которая входит в комплект системы PostgreSQL и необходима для выполнения почти всех упражнений в следующих главах этой книги.

3.5. Упражнения

Упражнение 3.1. Установите соединение с демонстрационной базой данных, используя psql.

Упражнение 3.2. С помощью команды \d (и других) получите информацию об объектах демонстрационной базы данных.

Упражнение 3.3. Установите программу pgAdmin (https://www.pgadmin.org). Найдите с ее помощью объекты демонстрационной базы данных.

Глава 4

Введение в SQL

4.1. Назначение языка SQL

Одним из основных требований к системам управления базами данных является наличие высокоуровневых средств выполнения запросов. В системах, реализующих реляционную модель данных, в качестве такого средства используется язык SQL. Фактически этот язык содержит полный набор операций, необходимых для выполнения любых действий с базой данных.

Стандарты SQL предусматривают подразделение средств языка на несколько категорий, отличающихся по их обязательности, для того чтобы реализация удовлетворяла требованиям стандарта:

- средства манипулирования данными (Data Manipulation Language, DML) обеспечивают выполнение поиска, извлечения, добавления, изменения и удаления данных, определенных в описании логической структуры базы данных, но не позволяют изменять эту структуру;
- средства определения данных (Data Definition Language, DDL) обеспечивают создание, модификацию и удаление элементов описания структуры базы данных, как логической, так и структуры хранения;
- расширения, предусмотренные стандартом, но не входящие в основное ядро языка;
- дополнительные средства, обеспечивающие описание особенностей, специфических для конкретной СУБД. Чаще всего это — определения конкретных структур хранений или их параметров.

В системе PostgreSQL имеется довольно много расширений, реализующих дополнительную функциональность системы (не предусмотренную стандартом SQL). Такие расширения, конечно, относятся к логическому уровню.

Далее в этой главе кратко описываются средства языка SQL, минимально необходимые для того, чтобы начать работу с базой данных. Более сложные конструкции, а также их использование обсуждаются в следующих главах. Ни эта глава, ни последующие не заменяют документацию или стандарт. Приводимые здесь сведения неформально описывают назначение отдельных конструкций языка, их взаимосвязи, соотношение с другими моделями баз данных, а также некоторые особенности их использования. Изложение иллюстрируется примерами, однако для всестороннего освоения языка SQL и системы PostgreSQL, конечно, необходимо изучить документацию и выполнить достаточное количество упражнений.

4.2. Быстрый старт

В этом разделе приводятся сведения о различных элементах SQL, необходимые для того, чтобы начать практическую работу с базой данных, в том числе для выполнения упражнений.

4.2.1. Простые типы данных

В теоретической реляционной модели данных значения атрибутов являются элементами доменов. Напомним, что домены могут описывать свойства некоторых конкретных типов значений, например длин, весов, денежных сумм, имен и т. п. Однако в ранних вариантах стандарта SQL не были предусмотрены средства учета специфики доменов и допускалось использование только таких типов, как числа, текстовые строки, моменты времени и некоторые другие, не привязанные к конкретным доменам.

В системе PostgreSQL имеются средства для определения пользовательских типов данных, однако в этой главе будут использоваться только типы, аналогичные типам ранних версий стандарта SQL. Определение доменов обсуждается позже.

В языке SQL предусмотрены следующие числовые типы данных.

Целые числа (int, integer) обычно представляют собой двоичные числа, для которых определены алгебраические операции и операции сравнения.

Вещественные числа (real, double precision) также обычно хранятся во внутреннем представлении, естественном для компьютера, на котором работает СУБД, и могут иметь различную точность. Конечно, для них тоже определены все обычные операции и отношения.

Десятичные числа (decimal, numeric) обрабатываются по правилам десятичной арифметики с фиксированным положением десятичной точки (отделяющей дробную часть от целой). Для этих типов можно указывать максимальное количество цифр и количество цифр после десятичной точки.

Последняя из перечисленных разновидностей числовых типов требует дополнительных пояснений. Результаты обычных арифметических операций в десятичной арифметике могут отличаться от результатов, получаемых при использовании других числовых типов. Это связано с особенностями правил округления. Для десятичных чисел, имеющих дробную часть, результаты зачастую оказываются неожиданными для программистов, привыкших работать только с целыми или вещественными числами (с плавающей точкой). Десятичные числа полезны для хранения денежных величин. Во многих странах, в которых основная денежная единица состоит из 100 более мелких единиц, законодательство требует, чтобы все вычисления выполнялись с сохранением ровно трех знаков после десятичной точки (т. е. вычисления должны выполняться с точностью до 0.1 цента или копейки, если такие правила применяются).

Отметим, однако, что промежуточные результаты в выражениях, содержащих десятичные числа, могут вычисляться с очень большой точностью, поэтому, для того чтобы получить правильное округление, необходимо принимать специальные меры.

Простой пример, иллюстрирующий особенности десятичной арифметики: процент, вычисленный от суммы нескольких значений, может не совпадать с суммой процентов, вычисленных для каждого значения отдельно.

В этом примере, который показывает особенности использования десятичной арифметики, используются операторы, действие которых объясняется ниже. Необходимость в использовании таких операторов связана с тем, что для промежуточных вычислений PostgreSQL автоматически выбирает тип данных, по возможности исключающий ошибки. Округление выполняется только при записи результатов. Поэтому в примере создается и заполняется таблица, затем вычисляются значения для колонки процентов. Далее операторы выборки данных показывают содержимое полученной таблицы и различия при подсчете суммы до и после вычисления процента.

```
demo=# CREATE TABLE fixed_precision (
    amount  numeric(7,2),
    percent numeric(7,2)
);
CREATE TABLE
demo=# INSERT INTO fixed_precision (amount) VALUES (2.13), (3.14);
INSERT 0 2
demo=# UPDATE fixed_precision SET percent = amount * 0.03;
UPDATE 2
demo=# SELECT * FROM fixed_precision;
 amount | percent
--------+---------
   2.13 |    0.06
   3.14 |    0.09
(2 rows)
demo=# SELECT sum(amount) * 0.03, sum(percent) FROM fixed_precision;
 ?column? | sum
----------+------
   0.1581 | 0.15
(1 row)
```

Другие примеры, иллюстрирующие особенности использования числовых типов, содержатся в упражнениях к этой главе.

Для хранения символьных данных можно использовать **строки фиксированной длины** (char) и **строки переменной длины** (varchar, text). Тип varchar требует указания максимальной длины, а text — нет и пригоден для хранения строк большого размера. В последних версиях PostgreSQL рекомендуется использовать text во всех случаях, когда ограничение максимальной длины не требуется.

Для времени предусмотрены типы **дат** (date), **времени суток** (time) и **отметок времени** (timestamp), которые включают и дату, и время. Для типов, содержащих время (т. е. time и timestamp), имеются разновидности, включающие (или не включающие) информацию о часовом поясе. В системе PostgreSQL эти типы определены так, как предусмотрено стандартом SQL, однако, как отмечено в документаци, полезность указания часового пояса без одновременного указания даты сомнительна, поэтому обычно часовой пояс указывают только для типа timestamp.

Логический тип (boolean), определенный в системе PostgreSQL, как и следовало ожидать, хранит булевы значения (истина или ложь).

4.2.2. Основные конструкции и синтаксис

Любое обращение клиентской программы к серверу базы данных, реализующему язык SQL, должно быть оформлено как *оператор SQL* (SQL statement). В языке SQL предусмотрено небольшое количество видов операторов, которые будут рассмотрены ниже. Каждый оператор начинается с ключевого слова, определяющего вид этого оператора. За этим ключевым словом обычно следуют *предложения* (clause), многие из которых также начинаются определенным ключевым словом, определяющим назначение этого предложения. Кроме ключевых слов, операторы могут содержать константы, имена объектов базы данных и позиции, в которые должны быть подставлены значения переменных, передаваемых программой-клиентом, а также встроенные функции языка.

Например, оператор SELECT, приведенный ниже, возвращает текущую дату, используя функцию current_date:

```
demo=# SELECT current_date;
 current_date
--------------
 2016-03-13
(1 row)
```

В языке SQL прописные и строчные буквы эквивалентны, за исключением символьных констант и идентификаторов, заключенных в двойные кавычки. Во многих учебниках ключевые слова принято записывать прописными буквами, чтобы отличать их от нетерминальных символов в синтаксических формулах. Такой же стиль принят и в документации PostgreSQL.

4.2.3. Описание данных: отношения

Подмножество языка SQL, предназначенное для описания данных (SQL DDL), включает операторы CREATE, ALTER и DROP, а также GRANT и REVOKE, используемые для управления разграничением доступа и рассматриваемые в главе 5.

За ключевым словом, определяющим оператор, следует другое ключевое слово, определяющее тип объекта, к которому применяется оператор, например CREATE TABLE является оператором создания таблиц. Далее обычно следует имя объекта базы данных и затем предложения, уточняющие, что именно делается при выполнении оператора. В этом разделе рассматривается только работа с таблицами на уровне логической структуры, а управление структурами хранения обсуждается далее в других разделах курса.

В системе PostgreSQL используется термин «отношение», однако он означает не отношения в смысле теоретической реляционной модели, а понятие, включающее таблицы и представления в смысле модели данных SQL, а также некоторые другие виды объектов базы данных.

Простейшая форма оператора CREATE TABLE содержит имя создаваемой таблицы, за которым следует список определений атрибутов (колонок), разделяемых запятыми. Определение каждого атрибута содержит его наименование, тип и может содержать некоторые дополнительные указания, в частности ограничения целостности и значение, присваиваемое этому атрибуту при добавлении строки в таблицу, если оно не указано явно.

Например, таблица дисциплин может быть создана с помощью следующего оператора:

```
demo=# CREATE TABLE courses (
    course_no varchar(30),
    title     text,
    credits   integer
);
CREATE TABLE
```

Последняя строка представляет собой ответ системы, подтверждающий выполнение оператора, и не является частью оператора.

Кроме определений атрибутов, список в команде CREATE TABLE может содержать ограничения целостности. Определение таблицы courses, приведенное выше, содержит не всю необходимую информацию. Более полный вариант оператора создания таблицы содержит определение первичного ключа и требование отсутствия неопределенных значений для атрибута, содержащего название курса:

```
demo=# CREATE TABLE courses (
    course_no varchar(30),
    title     text NOT NULL,
    credits   integer,
    CONSTRAINT course_pkey PRIMARY KEY (course_no)
);
CREATE TABLE
```

Заметим, что имена таблиц должны быть различны, поэтому выполнить оператор создания таблицы можно, только если таблицы с таким именем еще не существует.

Оператор DROP удаляет объект базы даных, заданный параметрами этого оператора. Например, для удаления таблицы courses можно было бы использовать следующий оператор:

```
demo=# DROP TABLE courses;
DROP TABLE
```

Оператор ALTER изменяет структуру или свойства существующего объекта. В частности, оператор ALTER TABLE можно использовать для добавления, переименования или удаления отдельных атрибутов таблицы, а также для добавления или удаления ограничений целостности.

В языке SQL предусмотрены следующие ограничения целостности.

NOT NULL запрещает появление неопределенных значений атрибута.

UNIQUE задает группу атрибутов, которые образуют возможный ключ отношения, т. е. для любых двух строк таблицы значения хотя бы одного из атрибутов, входящих в возможный ключ, должны различаться. Очень часто такое ограничение состоит всего из одного атрибута.

PRIMARY KEY задает атрибут или группу атрибутов, которые используются как первичный ключ отношения. Обычно такое ограничение определяется для атрибутов, составляющих идентификатор сущности, и поэтому обычно эти атрибуты считаются неизменяемыми, однако язык SQL допускает изменение значений атрибутов первичного ключа. Очень часто первичный ключ состоит только из одного атрибута, хотя язык SQL позволяет создавать составные первичные ключи.

Важное отличие уникального ключа от первичного состоит в том, что для атрибутов, входящих в уникальный ключ, могут допускаться неопределенные значения (NULL), а для атрибутов первичного ключа — нет.

FOREIGN KEY задает группу атрибутов, значения которых должны совпадать со значениями атрибутов первичного или уникального ключа другого отношения, указанных в этом ограничении. Таким образом, это ограничение задает бинарные связи между сущностями.

CHECK задает произвольное условие на значения одного или нескольких атрибутов в одной строке таблицы.

Невозможность выполнения операторов DDL вызывает индикацию ошибки: например, попытка создать таблицу с именем, совпадающим с именем существующей таблицы в той же схеме, или попытка удаления несуществующего объекта. В системе PostgreSQL имеются условные формы операторов языка описания данных, проверяющие возможность выполнения.

Например, оператор

```
DROP TABLE IF EXISTS old_table;
```

удалит таблицу old_table, если таблица с таким именем существовала, и ничего не сделает в противоположном случае. Для оператора CREATE условие записывается как IF NOT EXISTS, поскольку выполнение этого оператора возможно, если объект не существует.

В программе psql можно получить список доступных таблиц, используя команду \d или \dt. Команда \d с указанием имени таблицы выводит ее описание:

```
demo=# \d courses
                    Table "education.courses"
  Column   |         Type          | Collation | Nullable | Default
-----------+-----------------------+-----------+----------+---------
 course_no | character varying(30) |           | not null |
 title     | text                  |           | not null |
 credits   | integer               |           |          |
Indexes:
    "courses_pkey" PRIMARY KEY, btree (course_no)
Referenced by:
    TABLE "exams" CONSTRAINT "exams_course_no_fkey"
      FOREIGN KEY (course_no) REFERENCES courses(course_no)
```

(В выводе этой команды через точку перед именем таблицы находится указание схемы. О том, что это такое, говорится в разделе 4.5.)

Отметим, что хотя по форме операторы определения данных выглядят как выполняемые и изменения схемы можно делать когда угодно, эти операторы следует использовать как статические. Другими словами, все таблицы, которые нужны приложению, должны быть созданы до начала работы приложения, и в дальнейшем изменения схемы производятся относительно редко (например, при изменении версии приложения или появлении новых функций группы приложений, работающих с базой данных).

Это не совсем формальное требование мотивируется тем, что работа системы управления базами данных существенно зависит от статистических характеристик накопленных данных. Слишком частое изменение схемы базы данных

может помешать выбору оптимальных алгоритмов для их обработки и, следовательно, приведет к неэффективной работе СУБД. Кроме этого, операции изменения схемы блокируют доступ к изменяемым объектам на все время выполнения операции, что может помешать нормальной работе других пользователей с этим объектом. Наконец, код приложения может зависеть от схемы базы данных.

Технически, однако, ничто не препятствует динамическому выполнению операторов DDL, а в некоторых приложениях это может быть необходимо. В частности, вре́менные объекты, которые не должны сохраняться после окончания сеанса работы с базой данных, обычно создаются операторами DDL, выполняемыми в рамках того же сеанса.

4.2.4. Заполнение таблиц

Добавление новых данных производится оператором INSERT, содержащим два предложения. Первое из них (INTO) указывает, куда помещаются новые данные, и содержит имя таблицы, за которым может следовать список атрибутов, заключенный в круглые скобки. Второе предложение (VALUES) задает значения, которые добавляются в базу данных. В этом разделе мы используем только одну форму оператора INSERT, которая добавляет в базу данных одну строку. Значения атрибутов перечисляются в круглых скобках вслед за ключевым словом VALUES.

Следующие операторы заносят в базу данных строки созданной ранее таблицы courses.

```
demo=# INSERT INTO courses (course_no, title, credits)
    VALUES ('CS301', 'Базы данных', 5);
INSERT 0 1
demo=# INSERT INTO courses (course_no,  credits, title)
    VALUES ('CS305', 10, 'Анализ данных');
INSERT 0 1
```

Ответ системы (второе число) показывает количество строк в таблице, которые были изменены, в данном случае — добавлены.

Если в предложении INTO указан список атрибутов, то и значения в предложении VALUES должны идти в том же порядке. В этом случае не важно, в каком порядке расположены атрибуты в определении таблицы.

Если в предложении INTO перечислены не все атрибуты, имеющиеся в таблице, то остальные получают значения по умолчанию, указанные в описании таблицы, или неопределенные значения (NULL), если значения в описании таблицы не заданы.

В случае если список атрибутов в предложении INTO не указан, порядок значений в предложении VALUES должен совпадать с порядком в определении таблицы. Использование такой формы оператора INSERT делает код приложения зависимым от схемы базы данных: любое добавление или изменение порядка атрибутов потребует модификации кода приложения.

С другой стороны, если в коде приложений используется формат оператора INSERT, содержащий список атрибутов, то при добавлении новых атрибутов необходимо либо разрешать неопределенные значения, либо определять значения, присваиваемые по умолчанию, если приложение их не задает.

Если при выполнении оператора INSERT нарушаются ограничения целостности, то его выполнение заканчивается с ошибкой. В системе PostgreSQL в предложении ON CONFLICT можно указать действия, которые будут выполняться вместо индикации ошибки при нарушении ограничений уникальности (в том числе уникальности первичного ключа). Например, при попытке повторной вставки записи, в которой значение первичного или уникального ключа совпадает с уже имеющимся в таблице, можно вместо вставки выполнить изменение значений других атрибутов.

Обычно СУБД предоставляют средства для массовой загрузки данных. Это можно делать с помощью другой формы оператора INSERT, позволяющей включить данные, выбираемые из других таблиц (в этой форме вместо предложения VALUES записывается оператор SELECT).

В PostgreSQL предложение VALUES может содержать данные для нескольких строк таблицы. Кроме этого, в системе PostgreSQL имеется оператор COPY (не входящий в стандарт SQL), который может копировать данные из внешних файлов, находящихся в файловой системе того компьютера, на котором выполняется сервер базы данных, или наоборот — из базы данных во внешние файлы. Оператор может работать с несколькими форматами данных, в том числе с форматом CSV (comma-separated values), который часто используется для передачи данных между системами и инструментами анализа данных.

Массовую загрузку можно выполнить и с помощью команды \copy программы psql. Эта команда может записать внешний (по отношению к базе данных, т. е. находящийся в файловой системе компьютера, на котором запущен psql) файл,

содержащий данные в одном из нескольких форматов, или, наоборот, записать содержимое таблицы в файл. Такого рода возможности, однако, не являются частью SQL, и на самом деле в реализации этой команды используются операторы SQL для работы с базой данных, а чтение или запись файлов происходит в программе-клиенте.

4.2.5. Чтение данных

Любая операция чтения из базы данных на языке SQL задается оператором SELECT. Результатом выполнения этого оператора всегда является некоторая таблица, содержимое которой может передаваться в программу-клиент, запросившую выполнение оператора, или использоваться в любой другой конструкции языка SQL, в которой может находиться таблица.

После ключевого слова SELECT в операторе размещается несколько предложений, описывающих, какой именно результат должен быть извлечен из базы данных. Подчеркнем, что оператор описывает именно результат, но не алгоритм его вычисления, даже если запись выглядит как последовательность действий. СУБД, в том числе система PostgreSQL, может изменять порядок выполнения отдельных операций, с тем чтобы выполнить все вычисление наиболее эффективным способом.

В отличие от хранимых таблиц, создаваемых оператором CREATE, не требуется заранее описывать схему таблицы, которая получается в результате выполнения оператора SELECT. Первое предложение в операторе SELECT, следующее за этим ключевым словом, содержит список выражений, вычисление которых задает значения атрибутов результата, возможно, с указанием имен атрибутов результата. Если других предложений в операторе SELECT нет, то эти выражения могут содержать константы и функции с параметрами-константами, а результирующая таблица будет содержать ровно одну строку.

Например, оператор

```
demo=# SELECT current_date AS today, current_time AS right_now;
   today    |     right_now
------------+--------------------
 2016-03-27 | 20:53:48.109582+03
```

вырабатывает таблицу с атрибутами today и right_now, содержащую строку с текущими значениями даты и времени.

Предложение FROM оператора SELECT указывает список источников данных, из которых выбираются данные для результата. Источниками данных могут быть таблицы базы данных, представления или любые конструкции SQL, вырабатывающие таблицы (такие конструкции называются табличными выражениями). Пока мы будем предполагать, что все источники данных являются таблицами базы данных.

Если предложение FROM присутствует, то в выражениях, определяющих вычисляемые значения, можно использовать имена атрибутов таблиц, перечисленных в списке. Количество выводимых строк зависит от способа комбинирования таблиц и от других предложений. В простейшем случае, когда указана только одна таблица, для каждой строки таблицы будет выведена одна строка в таблицу-результат.

Например, оператор

```
demo=# SELECT title AS course_name, credits FROM courses;
  course_name  | credits
---------------+---------
 Базы данных   |       5
 Анализ данных |      10
(2 rows)
```

выводит таблицу, содержащую для каждой строки таблицы courses наименование курса и количество зачетных единиц. Поскольку имя второго атрибута не указано, используется имя колонки таблицы.

Вместо списка выражений можно указать символ «*». В этом случае выводятся все колонки таблицы, указанной в предложении FROM. Таким способом можно получить копию хранимой таблицы:

```
demo=# SELECT * FROM courses;
 course_no |     title     | credits
-----------+---------------+---------
 CS301     | Базы данных   |       5
 CS305     | Анализ данных |      10
(2 rows)
```

Отметим, что использование «звездочки» приводит к зависимости приложения от схемы базы данных: приложение должно быть готово принять столько колонок, сколько их имеется в хранимой таблице, независимо от того, нужны значения этих колонок для работы приложения или нет. «Звездочка» может также приводить к снижению эффективности работы сервера базы данных, поскольку в этом случае просматриваются все атрибуты. По этим причинам

не рекомендуется применять такой способ в приложениях, однако это удобно в интерактивных ad-hoc-запросах.

Предложение WHERE задает условия, которым должны удовлетворять строки входных таблиц, для того чтобы эти строки использовались при выполнении оператора. Эти условия могут быть выражены как:

- бинарные отношения, определенные на доменах атрибутов, связывающие значения атрибутов с константами или значения разных атрибутов или выражений (например, для числовых доменов определены бинарные отношения $=, \neq, >, \geqslant, <, \leqslant$);
- логические выражения, получаемые из более простых выражений при помощи операций конъюнкции (AND), дизъюнкции (OR) и отрицания (NOT) и круглых скобок;
- функции, вырабатывающие логические значения.

Например, оператор

```
demo=# SELECT title, credits
FROM courses
WHERE credits > 8;
     title     | credits
---------------+---------
 Анализ данных |      10
(1 row)
```

вырабатывает список курсов, по которым можно получить более 8 зачетных единиц. Отметим, что атрибуты, используемые в условиях, не обязательно должны включаться в число атрибутов результата.

4.2.6. Модификация данных

Оператор UPDATE изменяет значения атрибутов в хранимой таблице. За ключевым словом UPDATE следуют имя таблицы и предложение SET, содержащее список имен изменяемых атрибутов из этой таблицы с указанием выражений, задающих для них новые значения. Далее следует предложение WHERE (такое же, как в операторе SELECT), которое определяет, какие строки будут обрабатываться данным оператором. Если предложение WHERE не указано, изменению подвергаются все строки таблицы.

Оператор UPDATE часто используется для изменения только одной строки таблицы, хотя SQL допускает указание условий, которым удовлетворяет несколько строк. Отметим также, что для изменения значения не требуется его предварительное считывание в приложение оператором SELECT: выражение, вычисляющее новое значение атрибута, может содержать имя этого атрибута, при этом для вычисления нового значения будет использоваться старое. Новое значение может задаваться подзапросом, эта возможность более детально обсуждается ниже.

Например, оператор

```
demo=# UPDATE courses
SET credits = credits + 1
WHERE course_no = 'CS305';
UPDATE 1
```

увеличивает на единицу значение одного атрибута в одной строке.

Оператор DELETE удаляет строки из хранимой таблицы, имя которой указывается в предложении FROM, а строки, подлежащие удалению, задаются предложением WHERE. Как и для операторов SELECT и UPDATE, отсутствие предложения WHERE означает, что должны быть обработаны (т. е. удалены) все строки таблицы. Так же как и оператор UPDATE, чаще всего оператор DELETE используется для удаления только одной строки.

Так, оператор

```
demo=# DELETE FROM courses
WHERE course_no = 'CS305';
DELETE 1
```

удаляет одну строку таблицы (так как условие задает значение первичного ключа).

4.3. Запросы

В этом разделе показывается, каким образом можно выразить на языке SQL основные операции реляционной алгебры и как строить более сложные запросы. Для примеров, кроме уже показанной таблицы курсов, понадобятся также таблицы студентов и экзаменов.

Если используется готовый образ системы с PostgreSQL (как описано в главе 3), то эти таблицы уже есть в базе данных demo; если нет — их можно создать следующими командами:

```
CREATE TABLE students (
    stud_id    integer PRIMARY KEY,
    name       text NOT NULL,
    start_year integer NOT NULL
);
CREATE TABLE exams (
    stud_id    integer     REFERENCES students(stud_id),
    course_no  varchar(30) REFERENCES courses(course_no),
    exam_date  date,
    grade      integer NOT NULL,
    PRIMARY KEY(stud_id,course_no,exam_date)
);
INSERT INTO students
    VALUES (1451, 'Анна', 2014),
           (1432, 'Виктор', 2014),
           (1556, 'Нина', 2015);
INSERT INTO exams
    VALUES (1451, 'CS301', '2016-05-25', 5),
           (1556, 'CS301', '2017-05-23', 5),
           (1451, 'CS305', '2016-05-25', 5),
           (1432, 'CS305', '2016-05-25', 4);
```

4.3.1. Фильтрация и проекция

Операция фильтрации, по-видимому, является наиболее часто используемой, так как именно эта операция позволяет выбрать те данные, которые нужны приложению в данный момент. Фактически запись операции фильтрации на языке SQL уже приведена выше: условие фильтрации задается предложением WHERE, при этом предложение FROM должно содержать ровно одно табличное выражение (т. е. ту таблицу, данные из которой фильтруются), а условие, записанное в предложении WHERE, содержит только константы и атрибуты этой таблицы.

Операция проекции включает в результат только указанные атрибуты исходного отношения. При этом разные строки исходной таблицы могут оказаться совпадающими по значениям оставшихся атрибутов, поэтому в реляционной модели такие строки должны быть представлены одной строкой результата. Поскольку в модели данных SQL дубликаты допускаются, для их устранения после ключевого слова SELECT указывается еще одно ключевое слово DISTINCT.

Например, если таблица студентов содержит следующие данные:

```
demo=# SELECT * FROM students;
 stud_id |  name  | start_year
---------+--------+------------
    1451 | Анна   |       2014
    1432 | Виктор |       2014
    1556 | Нина   |       2015
(3 rows)
```

то операция проекции, извлекающая все различные годы поступления, может выглядеть так:

```
demo=# SELECT DISTINCT start_year FROM students;
 start_year
------------
       2015
       2014
(2 rows)
```

Ключевое слово DISTINCT является альтернативой для слова ALL, которое предполагается по умолчанию и поэтому обычно не указывается.

В некоторых случаях указание DISTINCT может существенно увеличивать время выполнения операторов SQL, поэтому не следует его указывать без необходимости. Например, если среди выбираемых атрибутов присутствуют все атрибуты первичного ключа, то строки результата будут различны.

Следует также заметить, что иногда кажущаяся необходимость в использовании ключевого слова DISTINCT может быть следствием ошибки в другой части программы или в данных. Дубликаты могут появляться, если программист забыл указать некоторые из условий фильтрации. Если используются генерируемые (суррогатные) первичные ключи, ошибки в коде приложения могут приводить к повторной вставке строк, описывающих сущности, уже представленные в таблице. Необдуманное использование DISTINCT в подобных случаях существенно затрудняет поиск ошибки.

4.3.2. Произведение и соединение

Операция прямого произведения задается на языке SQL указанием нескольких таблиц в предложении FROM, как показано на рис. 4.3.1.

```
demo=# SELECT * FROM courses, exams;
 course_no |     title      | credits | stud_id | course_no | exam_date  | grade
-----------+----------------+---------+---------+-----------+------------+-------
 CS301     | Базы данных    |       5 |    1451 | CS301     | 2016-05-25 |     5
 CS305     | Анализ данных  |      10 |    1451 | CS301     | 2016-05-25 |     5
 CS301     | Базы данных    |       5 |    1556 | CS301     | 2017-05-23 |     5
 CS305     | Анализ данных  |      10 |    1556 | CS301     | 2017-05-23 |     5
 CS301     | Базы данных    |       5 |    1451 | CS305     | 2016-05-25 |     5
 CS305     | Анализ данных  |      10 |    1451 | CS305     | 2016-05-25 |     5
 CS301     | Базы данных    |       5 |    1432 | CS305     | 2016-05-25 |     4
 CS305     | Анализ данных  |      10 |    1432 | CS305     | 2016-05-25 |     4
(8 rows)
```

Рис. 4.3.1. Прямое произведение в SQL

Как отмечено в главе 2, операция прямого произведения сама по себе не очень полезна, так как, по существу, ее результат не содержит дополнительной информации. Однако в том случае, если предложение FROM содержит несколько таблиц, в предложении WHERE можно задавать условия, связывающие атрибуты из разных таблиц, и, таким образом, задавать операцию соединения, как показано на рис. 4.3.2.

Поскольку имена атрибутов должны быть уникальными только в пределах одной таблицы, при совпадении имен атрибутов из разных таблиц необходимо указание, из какой таблицы следует выбирать значение атрибута. В приведенном примере атрибут course_no задан в условии с указанием имен таблиц.

В предложении WHERE можно перемешивать в произвольном порядке условия, задающие операцию фильтрации (которые содержат атрибуты только одной таблицы) и условия, задающие операцию соединения (которые связывают атрибуты разных таблиц).

В языке SQL существует альтернативная форма записи операции соединения с использованием ключевого слова JOIN, показанная на рис. 4.3.3.

Система обрабатывает оба варианта записи операции соединения одинаково. Ни результат, ни алгоритм выполнения не зависят от способа записи, поэтому можно использовать такой способ, который представляется более наглядным. Формально, однако, различие существенно: при использовании ключевого слова JOIN предложение FROM содержит одно табличное выражение, а не две таблицы.

При этом в предложении WHERE можно указать условия фильтрации:

```
demo=# SELECT students.name, exams.grade
FROM students
    JOIN exams ON students.stud_id = exams.stud_id
WHERE course_no = 'CS305';
  name  | grade
--------+-------
 Анна   |     5
 Виктор |     4
(2 rows)
```

Этот пример показывает, что операция соединения не включает в результат данные из строк исходных таблиц, для которых не нашлось пары в другой таблице: условие фильтрации ограничивает только выбор дисциплины, но при этом исключаются и студенты, которые указанную дисциплину не сдавали. Такое поведение вполне соответствует реляционной теории, однако во многих

```
demo=# SELECT * FROM courses, exams
WHERE courses.course_no = exams.course_no;
 course_no |     title      | credits | stud_id | course_no | exam_date  | grade
-----------+----------------+---------+---------+-----------+------------+-------
 CS301     | Базы данных    |       5 |    1451 | CS301     | 2016-05-25 |     5
 CS301     | Базы данных    |       5 |    1556 | CS301     | 2017-05-23 |     5
 CS305     | Анализ данных  |      10 |    1451 | CS305     | 2016-05-25 |     5
 CS305     | Анализ данных  |      10 |    1432 | CS305     | 2016-05-25 |     4
(4 rows)
```

Рис. 4.3.2. Условия соединения в предложении WHERE

```
demo=# SELECT *
FROM courses
    JOIN exams ON courses.course_no = exams.course_no;
 course_no |     title      | credits | stud_id | course_no | exam_date  | grade
-----------+----------------+---------+---------+-----------+------------+-------
 CS301     | Базы данных    |       5 |    1451 | CS301     | 2016-05-25 |     5
 CS301     | Базы данных    |       5 |    1556 | CS301     | 2017-05-23 |     5
 CS305     | Анализ данных  |      10 |    1451 | CS305     | 2016-05-25 |     5
 CS305     | Анализ данных  |      10 |    1432 | CS305     | 2016-05-25 |     4
(4 rows)
```

Рис. 4.3.3. Условия соединения с явным оператором JOIN

случаях бывает необходимо отображать информацию из строк, не попадающих в результат соединения.

В языке SQL предусмотрены варианты операции соединения, которые не являются в строгом смысле реляционными, но решают указанную задачу и поэтому часто используются на практике. Операция левого внешнего соединения (LEFT OUTER JOIN) возвращает, кроме обычного результата соединения, строки из левого операнда, для которых не нашлось парной строки в правом операнде. При этом вместо значений атрибутов правого операнда возвращаются неопределенные значения.

```
demo=# SELECT students.name, exams.grade
FROM students
    LEFT JOIN exams ON students.stud_id = exams.stud_id
                   AND course_no = 'CS305';
  name   | grade
--------+-------
 Анна    |     5
 Виктор |     4
 Нина    |
(3 rows)
```

Аналогично определяются правое (RIGHT) соединение, включающее строки второго операнда, для которых не нашлось пары в первом, и полное (FULL) внешнее соединение, которое включает непарные строки из обоих операндов.

Отметим, что в последнем примере условие фильтрации помещено в предложении ON вместе с условием соединения, а не в предложении WHERE. Это необходимо потому, что условия, записанные в предложении WHERE, вычисляются так, как будто они проверяются после вычисления табличного выражения в предложении FROM. Строки, для которых не нашлось пары (в нашем примере строка «Нина»), содержат неопределенные значения атрибута course_no, сравнение которого с любым значением дает результат NULL, и поэтому такие строки будут исключены из результата. Другими словами, внешнее соединение выполнится, как внутреннее:

```
demo=# SELECT students.name, exams.grade
FROM students
    LEFT JOIN exams ON students.stud_id = exams.stud_id
WHERE course_no = 'CS305';
  name   | grade
--------+-------
 Анна    |     5
 Виктор |     4
(2 rows)
```

Особенности использования условий при наличии неопределенных значений, конечно, никак не связаны ни с операцией внешнего соединения, ни с отношением равенства. Для того чтобы исключить неожиданности при использовании в предикатах фильтрации колонок, допускающих неопределенные значения, следует включать дополнительно проверку значений с помощью встроенных предикатов IS NULL или IS NOT NULL.

4.3.3. Псевдонимы для таблиц

Использование имен таблиц для уточнения принадлежности атрибутов может оказаться невозможным или недостаточным. В предложении FROM могут задаваться не хранимые таблицы, а табличные выражения, которые не имеют своих имен, как таблицы. Кроме этого, одна и та же таблица может использоваться в одном запросе в нескольких разных ролях. Например, в схеме демонстрационной БД авиабилетов, представленной на рис. 2.6.3, для того чтобы вывести наименования пунктов отправления и прибытия, необходимо выполнить соединение с таблицей аэропортов дважды в одном запросе.

В подобных случаях применяется аппарат *псевдонимов* (aliases): в предложении FROM после любого табличного выражения можно указать идентификатор, который будет использоваться в других частях запроса как псевдоним этого табличного выражения.

Использование псевдонимов для указания ролей таблиц показано в следующем примере из демонстрационной базы данных.

```
demo=# SELECT f.flight_no,
    f.departure_airport AS d_airport,
    dep.city AS d_city,
    f.arrival_airport AS a_airport,
    arr.city AS a_city
FROM flights f
    JOIN airports dep ON f.departure_airport = dep.airport_code
    JOIN airports arr ON f.arrival_airport = arr.airport_code
WHERE f.status = 'Departed'
  AND f.scheduled_arrival < bookings.now();
 flight_no | d_airport |   d_city    | a_airport |     a_city
-----------+-----------+-------------+-----------+----------------
 PG0496    | SVO       | Москва      | JOK       | Йошкар-Ола
 PG0574    | OVB       | Новосибирск | HMA       | Ханты-Мансийск
 PG0304    | SGC       | Сургут      | SVX       | Екатеринбург
(3 rows)
```

Зачастую псевдонимы используются для сокращения записи. Например, один из предыдущих запросов, возвращающий оценки, полученные студентами по некоторой дисциплине, можно переписать следующим образом:

```
demo=# SELECT s.name, e.grade
FROM students s
    LEFT JOIN exams e ON s.stud_id = e.stud_id
                      AND e.course_no = 'CS305';
  name  | grade
--------+-------
 Анна   |     5
 Виктор |     4
 Нина   |
(3 rows)
```

Однако использование псевдонимов для указания роли таблицы в запросе, конечно, намного важнее.

4.3.4. Вложенные подзапросы

Любой оператор SQL, вычисляющий какой-либо результат, оформляет этот результат в виде таблицы. Такие вычисленные таблицы можно не только читать в прикладной программе, но и использовать в других запросах. Оператор SQL, вырабатывающий таблицу, после заключения в круглые скобки становится табличным выражением и может использоваться в качестве таблицы в другом операторе. Конечно, такую таблицу нельзя модифицировать: табличные выражения используются только для выборки данных из них.

Табличные выражения, возвращающие одну строку и один атрибут (т. е. содержащие только одно скалярное значение), в языке SQL считаются совпадающими с этими скалярными выражениями. Такие табличные выражения можно использовать в другом запросе в любом месте, в котором требуется скалярное значение.

Подзапросы в списке выбираемых значений

Подзапросы, возвращающие одно скалярное значение, можно использовать в списке выражений после ключевого слова SELECT.

```
demo=# SELECT
    f.flight_no,
    f.departure_airport AS d_airport,
    ( SELECT city
      FROM airports
      WHERE airport_code = f.departure_airport
    ) AS d_city,
    f.arrival_airport AS a_airport,
    ( SELECT city
      FROM airports
      WHERE airport_code = f.arrival_airport
    ) AS a_city
FROM flights f
WHERE f.status = 'Departed'
  AND f.scheduled_arrival < bookings.now();
 flight_no | d_airport |   d_city    | a_airport |     a_city
-----------+-----------+-------------+-----------+----------------
 PG0496    | SVO       | Москва      | JOK       | Йошкар-Ола
 PG0574    | OVB       | Новосибирск | HMA       | Ханты-Мансийск
 PG0304    | SGC       | Сургут      | SVX       | Екатеринбург
(3 rows)
```

Если вложенный подзапрос в списке SELECT возвращает ровно одну строку, то этот оператор эквивалентен оператору, приведенному в предыдущем подразделе, и возвращает точно такой же результат.

Если вложенный запрос для какой-либо строки основного запроса возвращает больше, чем одну строку, выполнение всего оператора прекращается с индикацией ошибки.

Если же вложенный подзапрос не возвращает ни одной строки, в системе PostgreSQL в качестве значения подзапроса используется NULL, однако в других системах это может считаться ошибкой.

Форма оператора с вложенными подзапросами может использоваться, например, для улучшения читаемости кода. Это может быть целесообразно, если из таблицы выбирается только один атрибут.

Отметим, что каждый вложенный подзапрос превращается в операцию соединения. В приведенном выше запросе это не приводит к потере эффективности, потому что хотя соединения выполняются с одной и той же таблицей, но роли этой таблицы и, главное, предикаты соединения различаются. Однако в следующем примере вложенные подзапросы извлекают значения из одной и той же строки, поэтому использование такой формы записи может привести к потере эффективности, по сравнению с использованием явного соединения.

```
demo=# SELECT
    ( SELECT courses.course_no FROM courses
      WHERE courses.course_no  = exams.course_no
    ) AS course_no,
    ( SELECT courses.title FROM courses
      WHERE courses.course_no  = exams.course_no
    ) AS title,
    exams.exam_date,
    exams.stud_id,
    exams.grade
FROM exams;
 course_no |     title     | exam_date  | stud_id | grade
-----------+---------------+------------+---------+-------
 CS301     | Базы данных   | 2016-05-25 |    1451 |     5
 CS301     | Базы данных   | 2017-05-23 |    1556 |     5
 CS305     | Анализ данных | 2016-05-25 |    1451 |     5
 CS305     | Анализ данных | 2016-05-25 |    1432 |     4
(4 rows)
```

В этом примере выполняются два соединения с таблицей courses, хотя для получения результата достаточно выполнить соединение один раз.

Подзапросы в условии фильтрации

Вложенные подзапросы, возвращающие скалярные значения, можно использовать в предложении WHERE, для того чтобы задать условия на атрибуты другой таблицы, например:

```
demo=# SELECT stud_id, grade, course_no
FROM exams
WHERE (SELECT start_year
       FROM students
       WHERE students.stud_id = exams.stud_id) > 2014;
 stud_id | grade | course_no
---------+-------+-----------
    1556 |     5 | CS301
(1 row)
```

Как и для подзапросов в списке выбираемых значений, вложенные подзапросы представляют собой неявные операции соединения.

В системе PostgreSQL можно также использовать условия, которые сравнивают несколько скалярных выражений с подзапросом, возвращающим одну строку, содержащую столько же колонок. В этом случае скалярные выражения должны быть заключены в скобки.

Предикаты, использующие подзапросы

В языке SQL имеются средства, позволяющие формулировать условия на подзапросы, возвращающие произвольное количество строк. Бинарное отношение IN возвращает истинное значение тогда и только тогда, когда скалярный левый операнд этого отношения содержится в таблице, возвращаемой вторым операндом (заключенным в скобки).

Например, следующий запрос вычисляет список студентов, получивших оценки по указанному курсу:

```
demo=# SELECT name, start_year
FROM students
WHERE stud_id IN (
    SELECT stud_id FROM exams WHERE course_no = 'CS305'
);
  name  | start_year
--------+------------
 Виктор |       2014
 Анна   |       2014
(2 rows)
```

Как и в случае подзапросов, возвращающих одну строку, в подзапросе можно выбирать несколько колонок и, соответственно, сопоставлять их значения с несколькими скалярными значениями, заключенными в круглые скобки.

Так же, как и в предыдущих примерах, при использовании подзапроса выполняется неявная операция соединения. Отличие, однако, состоит в том, что будет выведена только одна строка результата для каждой строки из первой таблицы, для которой нашлась парная во второй, даже если парных строк несколько. В реляционной алгебре такая операция называется полусоединением.

Бинарное отношение NOT IN работает так же, как и отношение IN, однако возвращает противоположный результат (т. е. ложное значение, если скаляр содержится в таблице).

Следующий запрос возвращает список студентов, получивших только отличные оценки:

```
demo=# SELECT name, start_year
FROM students
WHERE stud_id NOT IN (SELECT stud_id FROM exams WHERE grade < 5);
 name | start_year
------+------------
 Анна |       2014
 Нина |       2015
(2 rows)
```

Неявное соединение в этом случае соответствует реляционной операции антисоединения (выбираются строки, для которых не нашлось пары во второй таблице).

Те же самые условия можно выразить и с помощью предиката EXISTS, который возвращает истинное значение, если подзапрос, заданный его операндом, возвращает непустую таблицу. Этот предикат, как и любой другой, можно использовать вместе с операцией логического отрицания NOT.

Список студентов, имеющих только отличные оценки, можно получить и таким способом:

```
demo=# SELECT name, start_year
FROM students
WHERE NOT EXISTS (
    SELECT stud_id
    FROM exams
    WHERE grade < 5 AND exams.stud_id = students.stud_id
);
 name | start_year
------+------------
 Анна |       2014
 Нина |       2015
(2 rows)
```

Предикат EXISTS можно рассматривать как реализацию квантора существования ($\exists$), а его отрицание можно использовать как квантор всеобщности ($\forall$).

Подзапросы как источники данных

Вложенные подзапросы можно использовать, наряду с другими конструкциями, в предложении FROM — как в качестве отдельных таблиц в списке, так и в качестве аргументов для операций JOIN.

4.3.5. Упорядочивание результата

В реляционной теории результат любого запроса является множеством и поэтому не имеет никакого упорядочения. Однако для практического использования часто бывает важно получать строки результата в определенном порядке. В языке SQL такое упорядочивание результата можно задать с помощью предложения ORDER BY, в котором указывается список значений, по которым требуется выполнить сортировку. При этом для каждого ключа можно указать

направление сортировки (по возрастанию — ASC, по убыванию — DESC). Если задано несколько ключей, то их значения используются лексикографически, т. е. значение второго ключа используется, если значения первого совпадают, и так далее.

```
demo=# SELECT *
FROM exams
ORDER BY grade, stud_id, course_no;
 stud_id | course_no | exam_date  | grade
---------+-----------+------------+-------
    1432 | CS305     | 2016-05-25 |     4
    1451 | CS301     | 2016-05-25 |     5
    1451 | CS305     | 2016-05-25 |     5
    1556 | CS301     | 2017-05-23 |     5
(4 rows)
```

Обычно упорядочивание делается по атрибутам результата, однако можно использовать и другие выражения.

4.3.6. Агрегирование и группировка

Средства агрегирования позволяют разместить в одной строке результата значения, вычисленные на основе данных из всех строк исходных таблиц или групп строк.

При агрегировании в качестве значений атрибутов можно использовать только выражения, содержащие вызовы агрегирующих функций или выражения, по которым производится группировка строк. Стандартом SQL предусмотрены следующие агрегирующие функции:

count вычисляет количество значений выражений, указанных в качестве аргумента;

avg вычисляет среднее значение атрибута (или выражения, зависящего от атрибутов исходных отношений);

sum суммирует значения атрибута или выражения;

min находит минимальное значение атрибута или выражения;

max находит максимальное значение атрибута или выражения.

В системе PostgreSQL существуют и другие функции агрегирования, а также можно определить собственные.

Следующий пример показывает, как вычисляются некоторые из агрегирующих функций по всем строкам таблицы:

```
demo=# SELECT count(*), count(DISTINCT stud_id), avg(grade)
FROM exams;
 count | count |        avg
-------+-------+--------------------
     4 |     3 | 4.7500000000000000
(1 row)
```

Функция count допускает в качестве параметра символ «*», при этом функция возвращает количество агрегируемых строк. Вариант со словом DISTINCT подсчитывает количество различных значений указанного выражения, а если вместо выражения указана «звездочка» — количество различных строк.

Для того чтобы вычислить агрегирующие функции по отдельности для каждой группы строк, необходимо задать значения, по которым будет определяться принадлежность строки к определенной группе. Обычно в качестве значений используются атрибуты, однако можно использовать любые выражения, зависящие от атрибутов исходной таблицы или таблиц. Список выражений, по которым производится группировка (ключей группировки), задается в предложении GROUP BY.

Например, следующий запрос выдает агрегированную информацию о курсах:

```
demo=# SELECT courses.title, count(*), avg(exams.grade)
FROM exams
    JOIN courses ON exams.course_no = courses.course_no
GROUP BY courses.title;
     title     | count |        avg
---------------+-------+--------------------
 Базы данных   |     2 | 5.0000000000000000
 Анализ данных |     2 | 4.5000000000000000
(2 rows)
```

Отметим, что совокупность ключей группировки составляет ключ отношения, которое получается в результате выполнения запроса.

Спецификация агрегирования в языке SQL имеет некоторые нелогичности. Так, при вычислении всех функций, кроме count, не учитываются неопределенные значения. В результате деление суммы на количество может давать значения, отличающиеся от значения avg. Кроме этого, агрегирование без указания

GROUP BY всегда возвращает ровно одну строку результата, даже если исходная таблица пуста (или нет строк, удовлетворяющих критериям фильтрации):

```
demo=# SELECT count(*) FROM exams WHERE grade > 200;
 count
-------
     0
(1 row)
```

При добавлении GROUP BY получается пустой результат:

```
demo=# SELECT count(*) FROM exams WHERE grade > 200
GROUP BY course_no;
 count
-------
(0 rows)
```

В запросах, использующих группировку, может возникнуть необходимость в фильтрации на основе значений, полученных в результате агрегирования. Такие условия фильтрации можно задать в предложении HAVING. Его отличие от предложения WHERE состоит в том, что в предложении WHERE можно использовать атрибуты исходных таблиц, а в предложении HAVING — атрибуты результата, т. е. агрегированные значения.

```
demo=# SELECT stud_id
FROM exams
WHERE grade = 5
GROUP BY stud_id
HAVING count(*) > 1;
 stud_id
---------
    1451
(1 row)
```

Некоторые условия фильтрации, например ограничения на значения ключа группировки, можно проверять как в предложении WHERE, так и в HAVING. Результаты будут одинаковыми.

4.3.7. Теоретико-множественные операции

Использование теоретико-множественных операций (объединение, пересечение, разность) проиллюстрируем на примерах выборки из самой маленькой таблицы в демонстрационной базе данных: aircrafts. Во всех примерах будем использовать два подмножества строк из этой таблицы. Конечно, для

теоретико-множественных операций не требуется, чтобы данные выбирались из одной таблицы, требуется только, чтобы совпадали типы всех выбираемых колонок.

Операция объединения записывается следующим образом:

```
demo=# SELECT * FROM aircrafts WHERE range > 4500
UNION
SELECT * FROM aircrafts WHERE range < 7500;
 aircraft_code |        model         | range
---------------+----------------------+-------
 CR2           | Бомбардье CRJ-200    |  2700
 CN1           | Сессна 208 Караван   |  1200
 733           | Боинг 737-300        |  4200
 319           | Аэробус A319-100     |  6700
 321           | Аэробус A321-200     |  5600
 320           | Аэробус A320-200     |  5700
 773           | Боинг 777-300        | 11100
 763           | Боинг 767-300        |  7900
 SU9           | Сухой Суперджет-100  |  3000
(9 rows)
```

Конечно, условия в подзапросах таковы, что объединение совпадает со всей таблицей aircrafts, и запрос извлекает содержимое этой таблицы далеко не лучшим образом. Заметим, что в этом случае результат, эквивалентный объединению, можно получить, записав условия фильтрации в одном предложении WHERE и соединив их логической операцией OR. Но этот прием невозможно применить, если объединяемые множества выбираются из разных таблиц.

В SQL имеется вариант операции объединения, который не удаляет дублирующие строки из результата:

```
demo=# SELECT * FROM aircrafts WHERE range > 4500
UNION ALL
SELECT * FROM aircrafts WHERE range < 7500;
 aircraft_code |        model         | range
---------------+----------------------+-------
 773           | Боинг 777-300        | 11100
 763           | Боинг 767-300        |  7900
 320           | Аэробус A320-200     |  5700
 321           | Аэробус A321-200     |  5600
 319           | Аэробус A319-100     |  6700
 SU9           | Сухой Суперджет-100  |  3000
 320           | Аэробус A320-200     |  5700
 321           | Аэробус A321-200     |  5600
 319           | Аэробус A319-100     |  6700
 733           | Боинг 737-300        |  4200
 CN1           | Сессна 208 Караван   |  1200
 CR2           | Бомбардье CRJ-200    |  2700
(12 rows)
```

Получить такой результат с помощью каких-либо комбинаций логических операций невозможно, потому что проверка условий WHERE никак не может привести к появлению дубликатов.

Следующий запрос вычисляет пересечение тех же множеств. Очевидно, он возвращает те самые строки, которые были дубликатами в предыдущем запросе:

```
demo=# SELECT * FROM aircrafts WHERE range > 4500
INTERSECT
SELECT * FROM aircrafts WHERE range < 7500;
 aircraft_code |      model       | range
---------------+------------------+-------
 321           | Аэробус A321-200 |  5600
 320           | Аэробус A320-200 |  5700
 319           | Аэробус A319-100 |  6700
(3 rows)
```

Как и для объединения, в данном случае такой же результат можно получить, записывая условия фильтрации в одном предложении WHERE, но логическая операция в этом случае будет AND. Как и для объединения, такая замена возможна, только если пересекаемые множества выбираются из одной таблицы (точнее, должны совпадать предложения FROM).

Операция разности вырабатывает строки, которые входят в первое множество, но не входят во второе:

```
demo=# SELECT * FROM aircrafts WHERE range > 4500
EXCEPT
SELECT * FROM aircrafts WHERE range < 7500;
 aircraft_code |     model     | range
---------------+---------------+-------
 773           | Боинг 777-300 | 11100
 763           | Боинг 767-300 |  7900
(2 rows)
```

Эквивалентное логическое выражение должно содержать операции AND NOT.

4.3.8. Вывод результатов после модификации данных

Операторы обновления обычно не вырабатывают никаких результатов, кроме изменений в базе данных и кода, характеризующего успешность выполнения операции (в том числе количество строк, обработанных оператором). В некоторых случаях необходима более детальная информация об измененных строках.

Предложение RETURNING задает список тех значений, которые будут выведены после выполнения операции модификации. При этом для каждой строки, обработанной оператором, выводится одна строка в результирующей таблице. Список выражений, специфицирующих атрибуты результата, строится точно так же, как список выражений в предложении SELECT, в том числе можно указывать «звездочку», для того чтобы вывести все атрибуты. Предложение RETURNING можно использовать во всех операторах модификации.

```
demo=# UPDATE courses
SET credits = 12
WHERE course_no = 'CS305'
RETURNING *;
 course_no |     title     | credits
-----------+---------------+---------
 CS305     | Анализ данных |      12
(1 row)
UPDATE 1
```

Для операций INSERT и UPDATE для формирования вывода используются значения, получившиеся после выполнения модификации, а для оператора DELETE — значения, которые были в удаленных из таблицы строках.

4.3.9. Последовательности

Одним из методов идентификации объектов, рассмотренных в главе 2, является применение суррогатных ключей. Значения таких ключей генерируются информационной системой таким образом, что каждое новое значение отличается от всех ранее полученных. Это гарантирует однозначность идентификации.

Генерация уникальных значений реализуется в SQL с помощью механизма *последовательностей* (sequence). В системе PostgreSQL последовательности рассматриваются как особый вид отношений, содержащих одну колонку и одну строку. Как и другие объекты базы даных, последовательности создаются с помощью оператора CREATE:

```
demo=# CREATE SEQUENCE my_seq;
CREATE SEQUENCE
```

Для последовательностей можно задавать начальное значение, шаг, конечное значение, а также поведение системы в случае, если достигнуто последнее значение. При исчерпании заданного диапазона может возбуждаться ошибка (при

каждой попытке получения нового значения) или может происходить переход к начальному значению. В последнем случае, конечно, уникальность не обеспечивается. Такие циклические последовательности не годятся для идентификации, но могут быть полезны в других случаях.

Для получения нового уникального значения последовательности вызывается функция nextval:

```
demo=# SELECT nextval('my_seq');
 nextval
---------
       1
(1 row)
demo=# SELECT nextval('my_seq');
 nextval
---------
       2
(1 row)
```

Подчеркнем, что применение функции nextval может гарантировать уникальность, но не гарантирует никаких других свойств получаемых значений. В частности, возможны пропуски некоторых значений. Это может происходить в результате обрыва транзакций (обсуждается более подробно в главе 6) или при одновременной работе нескольких приложений, выбирающих уникальные значения из одной последовательности.

Извлечь последнее сгенерированное в текущем сеансе значение последовательности можно обычным оператором SELECT, используя функцию currval:

```
demo=# SELECT currval('my_seq');
 currval
---------
       2
(1 row)
```

Функциональность последовательностей можно реализовать, обновляя значение обычного атрибута в любой постоянно хранимой таблице, но подобная реализация неизбежно окажется значительно менее эффективной, чем применение последовательности. Это связано с тем, что при работе с последовательностями не нужны и не используются средства изоляции транзакций, которые необходимы при работе с обычными таблицами.

В системе PostgreSQL в качестве типа данных колонки таблицы можно указать serial. В этом случае будет создана последовательность и функция nextval будет задана как значение по умолчанию для этой колонки. Если при вставке строк

значение колонки не указывается, будет выбрано следующее значение из последовательности. Такие колонки можно использовать в качестве первичных ключей, содержащих суррогатные значения.

Начиная с версии 10 в PostgreSQL реализована предусмотренная стандартом SQL конструкция GENERATED AS IDENTITY, обеспечивающая такие же функции, т. е. генерацию уникальных значений на основе последовательности. Использование этой конструкции предпочтительнее, чем serial, поскольку в этом случае последовательность управляется не как самостоятельный объект базы данных, а как неотъемлемая часть определения колонки.

Изменение параметров последовательности выполняется оператором ALTER, а удаление из базы данных — оператором DROP. Последовательности, созданные системой для колонок типа serial, удаляются из базы данных автоматически при удалении таблицы.

4.3.10. Представления

В соответствии с трехуровневой моделью описания данных приложение должно иметь возможность использовать структуры данных, отличающиеся от хранимых в базе данных (внешняя схема может отличаться от концептуальной). В языке SQL такая возможность реализуется с помощью аппарата *представлений* (view). Можно сказать, что представление является виртуальной таблицей: имя представления можно использовать в запросах в тех же местах, где можно использовать имена таблиц, однако содержимое представления специфицируется запросом (оператором SELECT) и обычно не хранится в базе данных.

Представления можно использовать в операторах выборки данных и при вычислении условий в любых операторах, однако в общем случае обновление представлений возможно только для ограниченного класса запросов, определяющих представления. Смысл этих ограничений в том, что по строке представления должна однозначно вычисляться строка хранимой таблицы, к которой нужно применить обновление. Это условие является необходимым, но СУБД может накладывать дополнительные ограничения, облегчающие проверку этого условия.

В системе PostgreSQL существует два механизма, позволяющих существенно расширить класс запросов, которые допускают операции обновления: это механизм правил, позволяющий переопределить способ выполнения любых операций на конкретном объекте, и механизм триггеров INSTEAD OF, которые опи-

сывают действия, выполняемые при попытке обновления тех объектов, на которых такие триггеры определены.

Другое ограничение на запросы, используемые в определениях представлений, состоит в том, что все выбираемые колонки (атрибуты результата) должны быть именованы. Иными словами, если выбираемое значение задается выражением, то такое выражение должно иметь псевдоним в определении запроса.

Кроме изменения структуры и формата данных, представления часто используются для того, чтобы записать сложные или часто встречающиеся подзапросы. Во втором случае представления можно рассматривать как некоторый аналог процедур или функций в императивных языках программирования.

Следующее представление использует таблицы демонстрационной базы данных и содержит список всех запланированных на ближайшую неделю рейсов с указанием даты вылета и кодов аэропортов отправления и прибытия:

```
demo=# CREATE VIEW upcoming_flights AS
    SELECT f.flight_id,
        f.flight_no,
        f.scheduled_departure::date AS d_date,
        dep.airport_code AS d_airport,
        arr.airport_code AS a_airport
    FROM flights f
        JOIN airports dep ON dep.airport_code = f.departure_airport
        JOIN airports arr ON arr.airport_code = f.arrival_airport
    WHERE f.scheduled_departure BETWEEN bookings.now()
      AND bookings.now() + INTERVAL '7 days';
CREATE VIEW
```

В этом представлении формат даты вылета отличается от хранимого в базе данных (в формате timestamp) времени отправления, а колонки с кодами аэропортов извлекаются из другой таблицы, соединение с которой необходимо выполнить дважды, потому что условия соединения различаются.

Когда представления используются в запросах в качестве таблиц, запросы выполняются так, как если бы данные, содержащиеся в представлениях, были бы записаны в хранимых таблицах. Однако не следует думать, что представления полностью вычисляются при выполнении каждого запроса, который их использует (хотя некоторые учебники по базам данных утверждают, что такое вычисление выполняется). Дело в том, что для вычисления результатов запроса могут быть нужны не все строки представления, а только небольшая их часть. Во многих случаях система управления базами данных в состоянии определить, какие именно строки представления необходимы для вычисления результата, и не пытается вычислить остальные.

В следующем запросе выбираются рейсы, которые запланированы на ближайшую неделю и которые направляются в один аэропорт. Для вычисления результата нет необходимости рассматривать строки, в которых аэропорт прибытия отличается от указанного:

```
demo=# SELECT * FROM upcoming_flights
WHERE a_airport = 'DYR';
 flight_id | flight_no |   d_date   | d_airport | a_airport
-----------+-----------+------------+-----------+-----------
      1706 | PG0315    | 2017-08-21 | DME       | DYR
      3865 | PG0450    | 2017-08-15 | VKO       | DYR
      6489 | PG0255    | 2017-08-19 | SVO       | DYR
     26493 | PG0088    | 2017-08-19 | KHV       | DYR
(4 rows)
```

В некоторых случаях использование представлений может привести к неэффективным запросам. Например, если из представления выбираются только даты вылета и номера рейсов, то прямое использование хранимой таблицы будет значительно более эффективным, потому что соединение с таблицей аэропортов не потребуется. В системе PostgreSQL при подготовке запросов к выполнению лишние операции соединения могут быть исключены, если данные из этих таблиц не используются для формирования результата запроса. Однако такое преобразование выполняется не всегда, а многие другие системы не делают его никогда.

4.4. Структуры хранения

При создании базы данных для ее хранения должно быть выделено некоторое пространство на устройстве энергонезависимой памяти с произвольным доступом. Это пространство называется *табличным пространством* (tablespace), и обычно такое пространство выделяется в виде файлов операционной системы, однако некоторые СУБД могут работать и непосредственно с устройствами, минуя интерфейс файловой системы.

В PostgreSQL память для табличных пространств не выделяется, а запрашивается отдельно для каждого хранимого объекта. Тем не менее понятие табличного пространства используется и определяет, где именно будут храниться объекты базы данных, в терминах каталогов (директорий, папок) файловой системы.

Управление размещением баз данных является одной из обязанностей администратора данных. Для баз данных небольшого размера обычно используют

пространство, которое выделяется системой при отсутствии каких-либо указаний (т. е. используется конфигурация, предусмотренная при установке программных компонент СУБД).

Все объекты базы данных, для которых требуется память, хранятся в табличных пространствах. При создании объекта можно указать, в каком табличном пространстве следует его размещать, но пока мы не будем обсуждать эту возможность.

Подчеркнем, что табличные пространства — структура уровня хранения. Их можно использовать, например, для изменения характеристик производительности (размещать некоторые объекты на устройствах большей производительности, если эти объекты часто используются). В некоторых случаях управление размещением важно для процедур создания резервных копий. Не следует применять табличные пространства для логического структурирования базы данных.

В языке SQL отсутствует четкая граница между описанием логической структуры данных и определением схемы хранения. Во многих операторах, создающих или модифицирующих структуры данных логического уровня, например таблицы, можно указывать параметры, влияющие на ее размещение (в табличном пространстве), и некоторые особенности структуры хранения.

Это относится и к материализованным представлениям, которые, как и обычные представления, специфицируются запросом, но результат выполнения которого записывается на диск как таблица, и поэтому для таких представлений возможно указание таких же параметров хранения, как для таблиц. Кроме этого, для материализованных представлений задаются параметры, определяющие способ обновления его содержимого.

Индексы являются структурой, которая полностью относится к схеме хранения, так как по определению индексы прозрачны для приложения (и, следовательно, не могут быть видны в логической структуре БД). В системе PostgreSQL реализовано большое количество различных типов индексов и предусмотрены разнообразные средства для расширения этого набора.

Здесь мы приведем только оператор SQL, создающий наиболее распространенный тип индекса: упорядоченный индекс по значениям одной или нескольких колонок (при этом в случае нескольких колонок применяется лексикографическое упорядочение). Такие индексы позволяют существенно ускорить выполнение фильтрации по точному значению входящих в индекс атрибутов или по диапазону значений и могут быть полезны для некоторых других операций.

Например, индекс, ускоряющий поиск пассажиров по имени, может быть создан командой:

```
demo=# CREATE INDEX tickets_passenger_name_idx
    ON tickets(passenger_name);
CREATE INDEX
```

Некоторые индексы могут создаваться для поддержки ограничений целостности. Для уничтожения как индексов, так и других объектов используется оператор DROP.

Планирование и создание индексов требует учета большого количества разнообразных факторов, в том числе особенностей СУБД, аппаратуры, на которой работает сервер баз данных, и, главное, характера выполняемых приложением запросов. Далеко не во всех случаях индексы приводят к сокращению количества ресурсов, необходимых для выполнения запросов, а в некоторых случаях наличие индексов приводит к заметному снижению производительности системы. Отметим также, что в PostgreSQL создание индексов является транзакционной операцией и может вызвать задержки в нормальной работе системы.

Покажем, каким образом индексы могут влиять на время отклика системы. Для этого понадобятся две ранее не упомянутые возможности.

Команда `\timing on` программы psql включает вывод времени выполнения каждого оператора:

```
demo=# \timing on
Timing is on.
```

Кроме этого, нам понадобится оператор EXPLAIN, который показывает, какие операции и в каком порядке будут использоваться при выполнении запроса. В этой части курса мы не будем подробно обсуждать всю информацию, которую можно получить с помощью EXPLAIN. Нам понадобятся только названия выполняемых операций, для того чтобы понять, чем отличаются разные варианты плана выполнения запроса.

Следующий запрос выводит загруженность рейсов, следующих в определенный день из Внуково в Пулково. Поскольку он потребуется несколько раз, создадим для него представление.

```
demo=# CREATE VIEW svo_led_utilization AS
    SELECT f.flight_no,
        f.scheduled_departure,
        count(tf.ticket_no) passengers
    FROM flights f
        JOIN ticket_flights tf ON tf.flight_id = f.flight_id
    WHERE f.departure_airport = 'SVO'
      AND f.arrival_airport = 'LED'
      AND f.scheduled_departure
          BETWEEN bookings.now() - INTERVAL '1 day'
              AND bookings.now()
GROUP BY f.flight_no, f.scheduled_departure;
CREATE VIEW
Time: 8,940 ms
```

Выполним запрос:

```
demo=# SELECT * FROM svo_led_utilization;
 flight_no |  scheduled_departure   | passengers
-----------+------------------------+------------
 PG0469    | 2017-08-15 12:35:00+03 |         40
 PG0470    | 2017-08-15 10:20:00+03 |        169
 PG0472    | 2017-08-14 18:30:00+03 |        121
(3 rows)
Time: 831.227 ms
```

Время выполнения составило около секунды, это очень большое время для демонстрационной базы данных. Причина состоит в том, что потребовалось выполнить полный просмотр (Seq Scan) таблиц, участвующих в запросе. Это можно увидеть, используя команду EXPLAIN (результат выполнения которой немного сокращен):

```
demo=# EXPLAIN (costs off) SELECT * FROM svo_led_utilization;
                      QUERY PLAN
-------------------------------------------------------
 GroupAggregate
   Group Key: f.flight_no, f.scheduled_departure
   -> Sort
        Sort Key: f.flight_no, f.scheduled_departure
        -> Hash Join
             Hash Cond: (tf.flight_id = f.flight_id)
             -> Seq Scan on ticket_flights tf
             -> Hash
                  -> Seq Scan on flights f
                       Filter: ...
(10 rows)
```

Попробуем построить индексы, которые могли бы ускорить выполнение этого запроса. Поскольку в запросе есть условие на точное значение аэропорта отправления, построим индекс для этого атрибута.

```
demo=# CREATE INDEX flights_departure_airport_idx
    ON flights(departure_airport);
CREATE INDEX
Time: 247.384 ms
```

Можно убедиться в том, что новый план выполнения того же запроса использует только что созданный индекс (Index Scan), как видно на рис. 4.4.1.

Тем не менее время выполнения запроса почти не изменилось:

```
demo=# SELECT * FROM svo_led_utilization;
 flight_no |  scheduled_departure   | passengers
-----------+-----------------------+------------
 PG0469    | 2017-08-15 12:35:00+03 |         40
 PG0470    | 2017-08-15 10:20:00+03 |        169
 PG0472    | 2017-08-14 18:30:00+03 |        121
(3 rows)
Time: 637.549 ms
```

Это можно объяснить тем, что индекс построен на таблице рейсов, которая по размеру значительно меньше, чем другая таблица, используемая в запросе. Фактически для выполнения запроса нужны только те строки таблицы ticket_flights, которые связаны с рейсами, удовлетворяющими критериям фильтрации. Поэтому следует попробовать создать еще один индекс на атрибут flight_id этой таблицы, который является внешним ключом, связывающим с таблицей flights. Важно, что именно этот атрибут использован в условии операции соединения.

```
demo=# CREATE INDEX ticket_flights_flight_id_idx
    ON ticket_flights(flight_id);
CREATE INDEX
Time: 3718.205 ms
```

Попробуем выполнить тот же самый запрос еще раз:

```
demo=# SELECT * FROM svo_led_utilization;
 flight_no |  scheduled_departure   | passengers
-----------+-----------------------+------------
 PG0469    | 2017-08-15 12:35:00+03 |         40
 PG0470    | 2017-08-15 10:20:00+03 |        169
 PG0472    | 2017-08-14 18:30:00+03 |        121
(3 rows)
Time: 4.268 ms
```

```
demo=# EXPLAIN (costs off) SELECT * FROM svo_led_utilization;
                                  QUERY PLAN
------------------------------------------------------------------------------
 GroupAggregate
   Group Key: f.flight_no, f.scheduled_departure
   -> Sort
        Sort Key: f.flight_no, f.scheduled_departure
        -> Hash Join
             Hash Cond: (tf.flight_id = f.flight_id)
             -> Seq Scan on ticket_flights tf
             -> Hash
                  -> Bitmap Heap Scan on flights f
                       Recheck Cond: (departure_airport = 'SVO'::bpchar)
                       Filter: ...
                       -> Bitmap Index Scan on flights_departure_airport_idx
                            Index Cond: (departure_airport = 'SVO'::bpchar)
```

Рис. 4.4.1. План выполнения запроса после создания индекса

Время выполнения сократилось примерно в 200 раз. На базе данных большего размера это соотношение было бы еще более значительным. Причина ускорения состоит в том, что теперь при выполнении запроса используются оба индекса, как видно на рис. 4.4.2.

Ускорение запроса на два порядка выглядит неплохо, но намного важнее то, что при увеличении размера базы данных время выполнения запроса без использования индексов будет расти линейно (пропорционально размеру большей таблицы), а при использовании индексов — пропорционально логарифму размера той же таблицы (но пропорционально количеству строк в результате выполнения запроса).

База данных, на которой выполнялись эти измерения, была размещена на устройстве SSD. Если бы вместо этого использовались вращающиеся диски, то различие во времени выполнения было бы еще более значительным. Заметим, что при выполнении того же запроса на другом компьютере абсолютные значения времени могут получиться совсем другими, однако важно не абсолютное время выполнения, а соотношение времени выполнения разных планов.

Подчеркнем, что задача выбора индексов далеко не так проста, как может показаться на основе этого примера, а выбор индексов — только один из инструментов, которые можно использовать для управления производительностью системы и эффективностью выполнения отдельных запросов.

4.5. Логическая организация данных

Логически взаимосвязанные объекты базы данных группируются в *схему*. Понятие схемы является чисто логическим, оно никак не связано с размещением или структурами, используемыми для хранения данных. Например, схема может содержать объекты, необходимые для работы одного приложения, но любое приложение может работать с объектами, находящимися в разных схемах. Внутри схемы имена объектов должны быть уникальны, полное имя объекта включает имя схемы и идентификатор (имя) объекта, разделенные точкой.

Схемы можно использовать для разграничения доступа, как описано в главе 5. Обычно в базе данных PostgreSQL существует схема public, доступ к которой имеют все пользователи, а доступ к другим схемам зависит от предоставленных привилегий.

```
demo=# EXPLAIN (costs off) SELECT * FROM svo_led_utilization;
                                QUERY PLAN
-------------------------------------------------------------------------
 GroupAggregate
   Group Key: f.flight_no, f.scheduled_departure
   -> Sort
        Sort Key: f.flight_no, f.scheduled_departure
        -> Nested Loop
             -> Bitmap Heap Scan on flights f
                  Recheck Cond: (departure_airport = 'SVO'::bpchar)
                  Filter: ...
                  -> Bitmap Index Scan on flights_dep_airport_idx
                       Index Cond: (departure_airport = 'SVO'::bpchar)
             -> Bitmap Heap Scan on ticket_flights tf
                  Recheck Cond: (flight_id = f.flight_id)
                  -> Bitmap Index Scan on ticket_flights_flight_id_idx
                       Index Cond: (flight_id = f.flight_id)
(14 rows)
```

Рис. 4.4.2. План выполнения запроса с двумя индексами

Для создания схемы используют оператор CREATE SCHEMA. Например, схема для демонстрационного примера создается оператором:

```
CREATE SCHEMA bookings;
```

Если при обращении к объекту схема не указана, то для поиска объекта используется значение переменной search_path. Значение этой переменной можно установить командой SET, а получить текущее значение — командой SHOW.

```
demo=# SHOW search_path;
   search_path
-----------------
 "$user", public
(1 row)
```

Такое определение пути поиска дает возможность не указывать имя схемы для объектов, размещенных в схеме текущего пользователя (если она существует) и в схеме public. При подключении к демобазе значение этой конфигурационной переменной автоматически изменяется так, чтобы не требовалось указывать имя схемы bookings для демонстрационных таблиц. Аналогичное изменение можно выполнить и вручную командой:

```
demo=# SET search_path TO bookings,"$user",public;
SET
```

При такой установке пути поиска следующие два оператора будут эквивалентны, потому что таблица airports находится в схеме bookings:

```
SELECT * FROM bookings.airports;
SELECT * FROM airports;
```

Этот же способ поиска применяется и для представлений, функций, типов и других видов объектов базы данных. Более того, такой же способ применяется и для системных объектов PostgreSQL.

Поскольку имя объекта должно быть уникально только в пределах схемы, возможно создание одинаково названных объектов в разных схемах. Поэтому выполнение запросов, в которых явно не указано имя схемы, будет зависеть от текущего значения пути поиска. Это, с одной стороны, дает возможность использовать идентичные запросы для обработки различных наборов данных за счет размещения их в разных схемах, с другой — создает условия для возникновения ошибок. Поэтому при проектировании базы данных и приложений необходимо установить и соблюдать правила, регламентирующие использование имен схем и размещенных в них объектов.

4.6. Итоги главы

В этой главе показаны основные конструкции языка SQL, способы выражения базовых операций реляционной алгебры в декларативном стиле, а также способы конструирования сложных декларативных запросов. На основе этого материала можно программировать значительную долю запросов, необходимость которых возникает в практических приложениях, использующих СУБД.

Не затрагивались более сложные конструкции SQL, процедурные средства программирования баз данных, средства расширения функциональности СУБД, поддержка расширенной системы типов данных и вопросы проектирования.

Более детальное изложение языка SQL можно найти в учебнике [18]. Описание основных конструкций SQL, ориентированное на систему PostgreSQL, содержится в [20]. Индексные структуры и их влияние на время выполнения запросов подробно обсуждаются в книге [16].

4.7. Упражнения

Предполагается, что все упражнения к этой главе выполняются над демонстрационной базой данных, с которой вы уже познакомились в главе 3.

Основные конструкции и синтаксис

Упражнение 4.1. Запустите клиент psql с демонстрационной базой данных и попробуйте выполнить запрос

```
SELECT 0
```

Что произошло? Какого синтаксического элемента не хватает?

Упражнение 4.2. Выберите все модели самолетов и соответствующие им диапазоны дальности полетов.

Упражнение 4.3. Найдите все самолеты с максимальной дальностью полета:

1) либо больше 10 000 км, либо меньше 4 000 км;
2) больше 6 000 км, а название не заканчивается на «100».

Обратите внимание на порядок следования предложений WHERE и FROM.

Упражнение 4.4. Определите номера и времена отправления всех рейсов, прибывших в аэропорт назначения не вовремя.

Упражнение 4.5. Подсчитайте количество отмененных рейсов из аэропорта Пулково (LED), и вылет, и прибытие которых было назначено на четверг.

Соединения

Упражнение 4.6. Выведите имена пассажиров, купивших билеты эконом-класса за сумму, превышающую 70 000 рублей.

Упражнение 4.7. Напечатанный посадочный талон должен содержать фамилию и имя пассажира, коды аэропортов вылета и прилета, дату и время вылета и прилета по расписанию, номер места в салоне самолета. Напишите запрос, выводящий всю необходимую информацию для полученных посадочных талонов на рейсы, которые еще не вылетели.

Упражнение 4.8. Некоторые пассажиры, вылетающие сегодняшним рейсом («сегодня» определяется функцией bookings.now), еще не прошли регистрацию, т. е. не получили посадочного талона. Выведите имена этих пассажиров и номера рейсов.

Упражнение 4.9. Выведите номера мест, оставшихся свободными в рейсах из Анапы (AAQ) в Шереметьево (SVO), вместе с номером рейса и его датой.

Агрегирование и группировка

Упражнение 4.10. Напишите запрос, возвращающий среднюю стоимость авиабилета из Воронежа (VOZ) в Санкт-Петербург (LED). Поэкспериментируйте с другими агрегирующими функциями (sum, max). Какие еще агрегирующие функции бывают?

Упражнение 4.11. Напишите запрос, возвращающий среднюю стоимость авиабилета в каждом из классов перевозки. Модифицируйте его таким образом, чтобы было видно, какому классу какое значение соответствует.

Упражнение 4.12. Выведите все модели самолетов вместе с общим количеством мест в салоне.

Упражнение 4.13. Напишите запрос, возвращающий список аэропортов, в которых было принято более 500 рейсов.

Модификация данных

Упражнение 4.14. Авиакомпания провела модернизацию салонов всех имеющихся самолетов «Сессна» (код CN1), в результате которой был добавлен седьмой ряд кресел. Измените соответствующую таблицу, чтобы отразить этот факт.

Упражнение 4.15. В результате еще одной модернизации в самолетах «Аэробус A319» (код 319) ряды кресел с шестого по восьмой были переведены в разряд бизнес-класса. Измените таблицу одним запросом и получите измененные данные с помощью предложения RETURNING.

Упражнение 4.16. Создайте новое бронирование текущей датой. В качестве номера бронирования можно взять любую последовательность из шести символов, начинающуюся на символ подчеркивания. Общая сумма должна составлять 30 000 рублей.

Создайте электронный билет, связанный с бронированием, на ваше имя.

Назначьте электронному билету два рейса: один из Москвы (VKO) во Владивосток (VVO) через неделю, другой — обратно через две недели. Оба рейса выполняются эконом-классом, стоимость каждого должна составлять 15 000 рублей.

Описание данных: отношения

Упражнение 4.17. Авиакомпания хочет предоставить пассажирам возможность повышения класса обслуживания уже после покупки билета при регистрации на рейс. За это взимается отдельная плата. Добавьте в демонстрационную базу данных возможность хранения таких операций.

Упражнение 4.18. Авиакомпания начинает выдавать пассажирам карточки постоянных клиентов. Вместо того чтобы каждый раз вводить имя, номер документа и контактную информацию, постоянный клиент может указать номер своей карты, к которой привязана вся необходимая информация. При этом клиенту может предоставляться скидка.

Измените существующую схему данных так, чтобы иметь возможность хранить информацию о постоянных клиентах.

Упражнение 4.19. Постоянные клиенты могут бесплатно провозить с собой животных. Добавьте в ранее созданную таблицу постоянных клиентов информацию о перевозке домашних животных.

Вложенные подзапросы

Упражнение 4.20. Найдите модели самолетов «дальнего следования», максимальная продолжительность рейсов которых составила более 6 часов.

Упражнение 4.21. Подсчитайте количество рейсов, которые хотя бы раз были задержаны более чем на 4 часа.

Упражнение 4.22. Для составления рейтинга аэропортов учитывается суточная пропускная способность, т. е. среднее количество вылетевших из него и прилетевших в него за сутки пассажиров. Выведите 10 аэропортов с наибольшей суточной пропускной способностью, упорядоченных по убыванию данной величины.

Псевдонимы для таблиц

Упражнение 4.23. С целью оценки интенсивности работы обслуживающего персонала аэропорта Шереметьево (SVO) вычислите, сколько раз вылеты следовали друг за другом с перерывом менее пяти минут.

Представления

Упражнение 4.24. Количество рейсов, принятых конкретным аэропортом за каждый день, — довольно востребованный запрос. Напишите представление данного запроса для аэропорта города Барнаул (BAX).

Глава 5

Управление доступом в базах данных

5.1. Модели защиты и разграничения доступа

Обеспечение защиты данных от несанкционированного использования с самого начала рассматривалось как одно из основных требований к системам управления базами данных, однако вопросы защиты и разграничения доступа к функциям системы важны практически в любой информационной системе. Поэтому соответствующие понятия и модели можно рассматривать в более широком контексте.

Построение любой модели защиты начинается с понятия *принципала* (английский термин — principal). Принципал обладает очень большими полномочиями (например, может принимать на работу или увольнять). Для нас, однако, важно только то, что принципал имеет право разрешать доступ к информационной системе другими лицам, которые в результате становятся *пользователями* этой информационной системы.

Только пользователи имеют возможность работы с системой. Для того чтобы разграничить возможности разных пользователей, вводятся понятия *объекта* и *действия*. Действия могут быть как связаны с объектом (например, методы объекта) или с классом объектов, так и не связаны. На этом уровне абстракции связь между действиями и объектами не имеет значения.

Обычно от имени принципала действует администратор, который для доступа к программной системе (например, операционной системе или системе управления базами данных) использует особый идентификатор пользователя, который называется *суперпользователем*. Как правило, суперпользователь создается при установке программной системы. В дальнейшем нам не понадобится различать понятия принципала, администратора, действующего от его имени, и суперпользователя, хотя эти понятия не совпадают. Например, суперпользователь операционной системы не обязательно совпадает с суперпользователем базы данных, хотя управление и СУБД, и операционной системой может быть поручено одному человеку.

Суперпользователь может выполнять любые действия и имеет неограниченный доступ ко всем объектам. Любой другой пользователь может выполнять только те действия и только над теми объектами, на которые ему предоставлено право.

Особое значение имеет действие, в результате которого возникает новый объект. Конечно, суперпользователь имеет право создания любых объектов, в том числе регистрировать новых пользователей. Кроме этого, он может предоставить право создания объектов (возможно, ограниченного класса) другим пользователям. Любой пользователь, создавший объект, становится его *владельцем*. Владелец любого объекта имеет право выполнять любые действия, связанные с этим объектом, и может предоставлять (возможно, ограниченные) права доступа к своим объектам и действиям над ними другим пользователям. Права доступа к объектам и использования действий называются *привилегиями*.

Для того чтобы упростить управление передачей привилегий пользователям, вводится понятие *роли*. Каждой роли передаются привилегии, необходимые для выполнения всех операций, связанных с этой ролью. Например, в системе интернет-магазина могут быть роли посетителя, покупателя, продавца и администратора.

Каждый пользователь получает право (привилегию) выполнять некоторую роль или несколько ролей. Кроме этого, конечно, пользователю могут передаваться и отдельные привилегии, не обязательно предоставленные какой-либо роли. Подобные модели защиты и разграничения доступа называют моделями, основанными на ролях (RBAC, Role Based Access Control). Кроме отдельных привилегий, роли могут получать другие роли в качестве привилегий. Передача роли в качестве привилегии эквивалентна передаче всех привилегий, имеющихся у этой роли.

Кроме RBAC, существуют другие модели, в частности основанные на значениях атрибутов объектов (ABAC, Attribute Based Access Control). Подобные модели необходимы, например, для того, чтобы пассажир мог получить доступ к своим бронированиям (в демонстрационной базе данных), но не к бронированиям других пассажиров. В этой главе мы не будем обсуждать механизмы системы PostgreSQL, необходимые для реализации разграничения доступа такого типа.

В сложных информационных системах применяется многоуровневая схема разграничения и контроля доступа: контроль может выполняться на уровне вычислительной сети, операционной системы, приложения и системы управления базами данных. При этом на разных уровнях обычно используются разные модели и, как правило, различные понятия пользователя.

На любом уровне работа всегда происходит от имени какого-нибудь пользователя, и при установлении соединения система должна его *аутентифицировать,* т. е. проверить, является ли пользователь тем, за кого себя выдает. Наиболее широко известна аутентификация при помощи имени пользователя и пароля, однако существуют и более сложные схемы, обеспечивающие более высокую степень защиты.

Обычно принято считать, что конфигурация средств контроля доступа относится к компетенции администратора баз данных, а не пользователей готовых приложений или разработчиков. Тем не менее как пользователям, так и разработчикам приложений полезно иметь представление о моделях разграничения доступа, которые могут использоваться на уровне базы данных. В этой главе приводятся основные сведения о средствах и моделях разграничения доступа.

5.2. Пользователи и роли в СУБД

Посмотрим, каким образом модели разграничения доступа реализуются в системах управления базами данных.

В системе PostgreSQL определены понятия *роли* и *привилегии* Для каждой роли задается набор *атрибутов*, определяющих ее свойства, и предоставляются привилегии для работы с объектами базы данных.

Роли применяются для представления различных функций.

Набор привилегий, которые часто должны предоставляться вместе, например, необходимы для использования какого-либо приложения. Такие роли часто бывают связаны с группами пользователей, которым необходимы одинаковые права доступа к данным.

Пользователь базы данных. Если роль имеет атрибут LOGIN, от ее имени можно создавать сеансы работы с сервером баз данных.

Важная особенность системы PostgreSQL состоит в том, что роли создаются на уровне кластера баз данных, а возможности работы от имени роли (пользователя) с каждой из баз данных, входящей в кластер, определяются соответствующими привилегиями. Атрибуты роли задают важные свойства роли, в том числе возможность управления объектами, которые определены на уровне кластера: в частности, ролями и базами данных.

В ранних версиях PostgreSQL использовались понятия пользователя и группы, однако начиная с версии 8.1 осталось только понятие роли, обобщающее эти два понятия.

Помимо аутентификации, СУБД обеспечивает разграничение доступа: каждой роли позволяется работать только с теми объектами данных и выполнять только те действия, на выполнение которых этой роли предоставлены привилегии. Например, объектами являются таблицы и хранимые процедуры. Пример действий — чтение, добавление, модификация или удаление данных. Перед выполнением любого действия СУБД выполняет *авторизацию*, т. е. проверку права роли на выполнение этого действия. Соединение с базой данных также является операцией, требующей авторизации.

Системы управления базами данных (в том числе и PostgreSQL) поддерживают понятие суперпользователя. Суперпользователь может совершать любые действия, ограничения доступа на него не распространяются. Наличие прав суперпользователя определяется атрибутом роли.

Создание, модификация и удаление ролей и пользователей выполняется с помощью операторов SQL. Оператор CREATE ROLE создает, ALTER ROLE изменяет, а оператор DROP ROLE — уничтожает роль. Во многих системах имеются аналогичные операторы для регистрации пользователей: CREATE USER и т. д. В системе PostgreSQL такие операторы тоже можно использовать, однако они, по существу, не отличаются от операторов для ролей.

В качестве действий, выполнение которых регулируется привилегиями, обычно используются операторы SQL. Так, существуют привилегии на создание объектов (CREATE), вставку, изменение и удаление (INSERT, UPDATE, DELETE соответственно). Та роль (пользователь), от имени которой был создан некоторый объект, становится владельцем этого объекта и имеет все права на доступ, модификацию и уничтожение этого объекта. Владелец может передавать права на доступ к своим объектам другим ролям (и пользователям).

Для некоторых типов объектов некоторые из действий не имеют смысла; в таких случаях, естественно, невозможно предоставление соответствующих привилегий (операция DELETE не имеет смысла для последовательностей). Некоторые привилегии не связаны непосредственно с операторами SQL (для хранимых функций существует привилегия EXECUTE на выполнение этих функций).

Например, оператор

```
demo=# CREATE ROLE reader;
CREATE ROLE
```

создает роль reader (не предоставляя этой роли никаких привилегий). Свойства ролей можно получить с помощью команды \du программы psql:

```
demo=# \du reader
            List of roles
 Role name |  Attributes  | Member of
-----------+--------------+-----------
 reader    | Cannot login | {}
```

Для того чтобы роль reader могла использоваться для создания сеансов, можно выполнить команду

```
demo=# ALTER ROLE reader LOGIN;
ALTER ROLE
```

Конечно, атрибут LOGIN можно указывать и при создании роли.

5.3. Объекты и привилегии

Перечень возможных привилегий не определен в стандарте SQL, поэтому в разных СУБД возможные виды привилегий могут различаться. Среди привилегий, которые могут быть предоставлены ролям, есть привилегии на создание объектов базы данных. В системе PostgreSQL имеются привилегии на создание схемы и на создание объектов внутри схемы. В других системах можно обнаружить отдельные привилегии для создания объектов каждого типа (таблиц, представлений и т. д.).

Пользователь, создавший объект, становится его владельцем. Владелец объектов базы данных имеет право выполнять любые операции над этими объектами и может предоставлять некоторые привилегии для работы с ними другим пользователям.

Хотя обычно владельцем объекта становится именно тот пользователь, который его создал, можно заменять владельца уже существующего объекта. Такая возможность важна, для того чтобы создавать объекты, владельцами которых будут пользователи, не имеющие права создавать объекты такого типа.

Предоставление привилегий осуществляется оператором GRANT, простейшая форма которого имеет вид:

```
GRANT привилегия ON объект_данных TO пользователь;
```

При этом можно указывать предложение WITH GRANT OPTION, чтобы получатель привилегии мог передавать эту привилегию другим пользователям. Любой пользователь, которому привилегия была предоставлена с правом передачи, может предоставлять эту привилегию так же, как привилегии на объекты, которыми он владеет.

Перечень привилегий, которые могут быть предоставлены для работы с некоторым объектом базы данных, зависит от типа этого объекта и, как правило, соответствует набору операторов, которые можно выполнять над этим объектом. Например привилегии для таблиц включают SELECT, UPDATE, INSERT, DELETE, TRUNCATE, REFERENCES и TRIGGER. Последние две привилегии позволяют создавать ограничения ссылочной целостности (FOREIGN KEY) и триггеры соответственно. В качестве списка привилегий в операторе GRANT можно указывать ALL PRIVILEGES: в этом случае передаются все привилегии, определенные для этого типа объекта, которые имеет пользователь, от имени которого выполняется оператор GRANT.

Если в качестве роли в операторе GRANT указано public, то привилегии предоставляются всем пользователям (ролям), как уже существующим, так и тем, которые, возможно, будут созданы в будущем.

Например, оператор

```
demo=# GRANT SELECT ON TABLE airports TO public;
GRANT
```

предоставляет право выборки данных из таблицы (или представления) airports всем пользователям (ролям) сервера базы данных.

Если при конфигурации базы данных не определено иначе, то все объекты, определенные в схеме public, становятся доступными для роли public.

Если объект базы данных является таблицей или представлением, то можно указать список колонок, на которые распространяются предоставляемые привилегии.

Предоставленные пользователю привилегии могут быть отозваны с помощью оператора

```
REVOKE привилегия ON объект_данных FROM пользователь;
```

Операторы GRANT и REVOKE позволяют предоставлять привилегии сразу на все объекты некоторого типа.

Например, оператор

```
demo=# GRANT SELECT ON ALL TABLES IN SCHEMA bookings TO reader;
GRANT
```

предоставляет привилегию чтения данных из всех таблиц схемы bookings пользователю (роли) reader. Заметим, однако, что доступ не будет предоставлен для таблиц, созданных в этой схеме после выполнения данной команды.

Некоторые привилегии не связаны с конкретным объектом; например, такими привилегиями являются роли. Для них указание объекта в операторах GRANT и REVOKE не требуется. Так, оператор

```
demo=# GRANT reader TO writer;
GRANT
```

предоставит роли writer все привилегии роли reader (как уже имеющиеся, так и те, которые могут быть ей предоставлены в будущем). Заметим, что в этой форме оператора GRANT нельзя использовать public в качестве получателя привилегий, т. е. роли не могут быть сделаны общедоступными.

5.4. Итоги главы

В этой главе рассмотрены основные понятия модели разграничения доступа на основе ролей (Role Based Access Control, RBAC) и показано, каким образом эта модель реализуется в системе привилегий PostgreSQL.

5.5. Упражнения

При выполнении упражнений этой главы особенно важно понимать, от имени какого пользователя СУБД выполняются команды. Текущего пользователя можно узнать командой

```
SELECT session_user;
```

Чтобы ориентироваться было проще, можно изменить приглашение psql таким образом, чтобы оно включало не только имя текущей базы данных, но и имя пользователя. Это выполняется следующими командами (которые могут быть помещены в файл ~/.psqlrc, чтобы их не приходилось вводить каждый раз заново):

```
demo=# \set PROMPT1 %n@%/%R%#
student@demo=# \set PROMPT2 :PROMPT1
```

Напомним, что переключение на другого пользователя в программе psql выполняется командой \c.

Упражнение 5.1. Создайте роль для доступа на чтение к демонстрационной базе данных без права создания сеансов работы с сервером БД.

Упражнение 5.2. Создайте пользователя сервера БД и предоставьте ему привилегию использования роли, созданной в предыдущем упражнении. Проверьте, что этот пользователь может выполнять любые запросы на выборку из таблиц демонстрационной базы данных, но не может их обновлять.

Упражнение 5.3. Заберите у пользователя привилегию, выданную в предыдущем упражнении. Убедитесь, что этот пользователь не сможет осуществлять выборку из таблиц демонстрационной базы данных.

Упражнение 5.4. Постройте пример, показывающий, что для доступа к таблицам схемы необходимо также предоставить право использования (USAGE) этой схемы.

Глава 6

Транзакции и согласованность базы данных

Средства управления транзакциями в системах управления базами данных выполняют функции, связанные с реализацией согласованности, т. е. обеспечивают корректное состояние данных. Во время нормальной работы системы за поддержку согласованности отвечает диспетчер транзакций, работающий на основе некоторого протокола. Каждый протокол гарантирует выполнение некоторого критерия корректности. Выбор протокола и, следовательно, поведение диспетчера зависят от требований приложения и от конфигурации системы.

Отметим, что использование диспетчеров и протоколов необходимо для обеспечения корректности (согласованности) только при одновременной работе с базой данных нескольких сеансов. При этом не имеет значения, от имени каких пользователей и какие приложения используют эти сеансы, важно только то, что операции с базой данных выполняются в разных сеансах независимо друг от друга. Управление транзакциями часто связывают с организацией многопользовательского доступа к базе данных, хотя в этом контексте не имеет значения, созданы сеансы от имени одного пользователя или от имени разных.

Другая, не менее важная функция средств управления транзакциями — восстановление корректного состояния после отказов (т. е. обеспечение отказоустойчивости): после любых отказов база данных должна быть приведена в корректное состояние. Важно отметить, что для многих классов современных приложений отказоустойчивость баз данных значительно важнее, чем согласованность. Поэтому довольно часто используются протоколы управления транзакциями, обеспечивающие соблюдение только ослабленных критериев корректности, но гарантирующие максимально возможное сохранение результатов работы приложений.

В этой главе обсуждаются требования к средствам поддержки согласованности, возможные последствия неконтролируемого выполнения, а также управление транзакциями в приложениях.

6.1. Определение и основные требования к транзакциям

Транзакцией называется конечное множество операций над базой данных, выполняемое приложением, которое переводит базу данных из одного согласованного состояния в другое согласованное состояние, при условии что транзакция выполнена полностью и без помех со стороны других приложений. В этой главе мы будем предполагать, что множество операций, составляющее транзакцию, упорядочено.

Прежде всего необходимо еще раз подчеркнуть различие между понятиями *целостности* (integrity) и *согласованности* (consistency): целостность определяется с помощью ограничений целостности, и СУБД предотвращает любые попытки нарушения этих ограничений. В то же время согласованность может временно нарушаться и восстанавливается только в конце выполнения транзакции.

Неформально требования к транзакционным системам принято описывать в терминах свойств ACID — по первым буквам английских названий: *атомарность* (atomicity), *согласованность* (consistency), *изоляция* (isolation), *долговечность* (durability). Все эти свойства требуют дополнительных пояснений.

Атомарность означает, что любая транзакция должна быть либо выполнена полностью, либо, если по каким-либо причинам завершение транзакции невозможно, ее частичное выполнение не должно оставлять никаких следов ни в базе данных, ни в результатах работы приложений. Это требование накладывает ограничения как на СУБД (которая должна устранять последствия неполного выполнения транзакций), так и на приложения, которые не должны выводить какие-либо результаты, зависящие от не полностью выполненных транзакций.

Требование **согласованности** включено в наше определение транзакций. Еще раз отметим, что согласованность определяется семантикой приложения и не может быть в полной мере проверена СУБД.

Требование **изоляции** означает, что СУБД должна обеспечить выполнение без помех со стороны других транзакций — ограничение, входящее в определение транзакции. Нарушение изоляции транзакций может приводить к появлению некорректных результатов и состояний базы данных. Подобные ситуации называются аномалиями конкурентного выполнения и обсуждаются ниже в этой главе. Для предотвращения всех возможных аномалий необходима полная изоляция транзакций, однако требование изоляции может вступать в противоре-

чие с требованием высокой пропускной способности, поэтому довольно часто используются ослабленные условия изоляции.

Свойство **долговечности** предъявляет очень сильные и трудно реализуемые требования к СУБД. Оно означает, что никакие изменения, выполненные завершенными транзакциями, не могут быть потеряны, что бы ни происходило с сервером базы данных или вычислительной системой, на которой этот сервер работает. Таким образом, это требование, по существу, определяет отказоустойчивость базы данных. Конечно, это требование вовсе не означает, что данные, записанные транзакцией, не могут быть изменены другими транзакциями. Возможно, следующая транзакция изменит их через миллисекунды, однако она в своей работе будет учитывать результаты работы завершенных транзакций.

Необходимо также отметить, что современные технологии позволяют обеспечить любую степень отказоустойчивости базы данных, однако это может привести к существенному увеличению стоимости системы как на этапе разработки, так и во время эксплуатации. Выбор уровня защищенности от отказов обычно требует компромиссов, учитывающих реальные требования прикладной системы.

Из свойства атомарности следует, что любая транзакция может завершиться одним из двух способов.

- Нормальное завершение транзакции называется *фиксацией* (commit). Операция фиксации выполняется приложением для того, чтобы сообщить СУБД, что все операции транзакции выполнены.
- Невозможность полного выполнения приводит к необходимости *обрыва* транзакции (abort).

Обрыв может выполняться как по инициативе приложения, так и по инициативе СУБД. Способ реализации обрыва зависит от внутренней организации СУБД, однако в большинстве систем операции выполняются в предположении, что транзакция будет зафиксирована, а в случае обрыва внесенные изменения устраняются из базы данных. Операция устранения изменений называется *откатом* (rollback).

6.2. Аномалии конкурентного выполнения

В соответствии с определением транзакций существует просто реализуемый способ обеспечения согласованности: строго последовательное выполнение транзакций. Очевидно, при этом каждая транзакция выполняется без помех со стороны других транзакций. Единственное, что все-таки нужно сделать в этом случае — это гарантировать откат транзакций, которые не могут завершиться полностью.

Такой метод выполнения транзакций, однако, привел бы к очень большим задержкам при работе системы, поскольку почти полностью исключил бы возможность одновременного выполнения каких-либо операций. Высокая производительность систем управления базами данных достигается, кроме всего прочего, тем, что большое количество транзакций выполняется одновременно.

Заметим, что для одновременного выполнения транзакций вовсе не обязательно, чтобы какие-либо компоненты СУБД или вычислительной системы работали параллельно. Даже в очень простой системе, которая выполняет все операции в рамках одного процесса, возможно чередование операций разных транзакций, поэтому управление транзакциями оказывается необходимым. Такая организация работы дает возможность завершать короткие транзакции быстрее, чем при строго последовательном выполнении, поскольку их операции будут выполняться между операциями других, возможно, более длинных транзакций. Мы будем поэтому говорить не о параллельном, а о конкурентном выполнении транзакций, объединяя, таким образом, чередование операций разных транзакций и при последовательном выполнении операций, и при псевдопараллельном (многопроцессном) выполнении, и при действительно параллельном выполнении на соответствующих конфигурациях оборудования.

Ситуации, в которых конкурентное выполнение корректных транзакций приводит к некорректным результатам, принято называть аномалиями. Для иллюстрации аномалий достаточно предполагать, что операции выполняются последовательно (однако, конечно, операции разных транзакций чередуются). Кроме этого, достаточно использовать только простые операции чтения и записи, обрабатывающие один элемент данных каждая. Сейчас для нас не имеет значения, что представляет собой элемент данных: это могут быть физический блок на устройстве хранения, содержащий страницу базы данных, строка таблицы, пара ключ-значение или какой-нибудь еще объект в базе данных. Важно только то, что разные элементы данных не имеют общих частей,

т. е. изменение значения одного элемента никак не влияет на значения других элементов. Для того чтобы облегчить восприятие, при чтении этой главы можно считать, что элементы данных — это строки таблиц. Операция чтения элемента x, выполняемая в транзакции i, обозначается $r_i(x)$, операция записи — $w_i(x)$, а сама транзакция обозначается t_i.

Запись последовательности выполнения всех операций нескольких транзакций называется *историей*, а любой начальный отрезок истории называется *расписанием*.

Рассмотрим простую транзакцию, которая читает числовое значение из базы данных, уменьшает его на 100 и записывает обратно. Если такая транзакция выполняется дважды для разных пользователей, но с одним и тем же числовым значением, в результате это значение, очевидно, уменьшится на 200. Корректное (последовательное) выполнение этих двух транзакций может быть представлено следующим расписанием:

$$r_1(x)\ w_1(x)\ r_2(x)\ w_2(x).$$

Однако если операции двух транзакций выполняются в следующем порядке:

$$r_1(x)\ r_2(x)\ w_1(x)\ w_2(x),$$

то в результате будет записано значение, учитывающее только работу последней записавшей транзакции из двух, выполнявших изменение, т. е. значение x будет уменьшено только на 100, потому что вторая транзакция читает x до того, как первая записывает результаты своей работы. Такая аномалия называется **потерянным обновлением**.

Условие согласованности может связывать значения различных элементов данных. Например, предположим, что сумма значений x и y должна оставаться постоянной в любом согласованном состоянии базы данных. Пусть первая транзакция считывает оба элемента данных, затем вычитает ненулевое значение из x и прибавляет его к y, а вторая транзакция читает оба этих объекта. Тогда следующая последовательность выполнения операций

$$r_1(x)\ r_1(y)\ r_2(x)\ w_1(x)\ w_1(y)\ r_2(y)$$

приводит к некорректному результату: вторая транзакция считывает несогласованные значения объектов x и y. Аномалии такого типа называются аномалиями **несогласованного чтения**.

Еще один вид аномалий связан с операциями завершения транзакций, т. е. фиксации (*c*) и обрыва (*a*). Рассмотрим следующую последовательность:

$$r_1(x)\ w_1(x)\ r_2(x)\ a_1\ c_2.$$

Поскольку первая транзакция обрывается, ее результаты не должны оставлять следов, однако они уже были прочитаны второй транзакцией. Такая ситуация называется аномалией **грязного чтения**. Заметим, что устранить эту аномалию можно двумя способами: задержать выполнение операции второй транзакции до завершения первой (не важно, фиксации или обрыва) или оборвать вторую транзакцию, когда обрывается первая. Такие обрывы, которые могут быть нужны для сохранения согласованности, называются *каскадными обрывами*.

Известно еще несколько видов аномалий конкурентного выполнения.

Грязная запись — вторая транзакция записывает новое значение до фиксации первой: $w_1(x)\ w_2(x)\ c_1$.

Нечеткое (неповторяющееся) чтение — повторное чтение элемента данных дает другой результат, поскольку значение элемента было изменено другой транзакцией: $r_1(x)\ w_2(x)\ c_2\ r_1(x)$.

Фантомное чтение — повторный поиск данных по предикату возвращает результат, отличающийся от первого, потому что другая транзакция добавила, удалила или изменила записи таким образом, что они стали удовлетворять (или перестали удовлетворять) предикату.

Несогласованная запись — в отличие от последовательного выполнения транзакций, новые значения *x* и *y* не учитывают значений, записанных другой транзакцией: $r_1(x)\ r_2(y)\ w_1(y)\ w_2(x)\ c_1\ c_2$.

Аномалия только читающей транзакции — в приведенном ниже расписании t_3 возвращает некорректные значения, т. к. учитывает изменение элемента *y* первой транзакцией, но не учитывает изменения *x* второй транзакцией, хотя t_2 использует значение, записанное до выполнения t_1: $r_2(x)\ w_1(y)\ c_1\ r_3(x)\ r_3(y)\ c_3\ w_2(x)\ c_2$.

В дальнейшем нам понадобятся различные критерии корректности. Идеальная корректность расписания достигается, если все результаты работы всех операций всех транзакций, как записываемые в базу данных, так и передаваемые в приложение, совпадают с результатами некоторого последовательного выполнения тех же транзакций. Расписание, выполнение которого эквивалентно последовательному, называется *сериализуемым*. Заметим, что разные последовательные расписания не обязательно эквивалентны между собой, т. е.

могут давать разные результаты, но любое из них будет корректным. Более слабые критерии корректности определяются в терминах запрещаемых аномалий: чем больше аномалий предотвращается, тем выше степень корректности.

6.3. Восстановимость

Требование отказоустойчивости накладывает дополнительные ограничения на то, в каком порядке могут выполняться операции разных транзакций.

Отказы могут приводить к необходимости обрыва транзакций. В большинстве СУБД обрывы реализуются с помощью операции *отката* (rollback). Выполнение этой операции реализуется так, как будто для каждой операции записи $w(x)$, которая была выполнена в обрываемой транзакции, выполняется обратная операция $w^{-1}(x)$, в порядке, обратном тому порядку, в котором выполнялись (прямые) операции записи. По определению, обратная операция записывает в объект x то значение, которое этот объект имел непосредственно перед выполнением прямой операции записи.

Во многих высокопроизводительных системах, в том числе в PostgreSQL, применяются многоверсионные протоколы управления транзакциями, в которых содержимое данных перед обновлением сохраняется в базе данных. Поскольку предыдущее состояние данных фактически сохраняется в блоках данных, записывать его заново при выполнении операции обратной записи не требуется. Для того чтобы сделать вновь записанные версии недействительными, достаточно пометить транзакцию как оборванную.

Такое восстановление состояния базы данных можно выполнять только в том случае, если данные, измененные обрываемой транзакцией, не были изменены другой транзакцией. Например, в следующей последовательности операций восстановление невозможно:

$$w_1(x)\ w_2(x)\ c_2\ a_1.$$

Дело в том, что при выполнении $w_1^{-1}(x)$ с целью отката первой транзакции было бы восстановлено значение x перед выполнением первой операции записи, и при этом было бы потеряно значение, записанное второй транзакцией. Но вторая транзакция была зафиксирована, и поэтому ее результаты не должны быть потеряны. Такие расписания называются невосстановимыми, однако необходимо сразу заметить, что механизм управления транзакциями, применяемый в системе PostgreSQL, гарантирует восстановимость.

6.4. Диспетчеры и протоколы

Примеры, приведенные в предыдущих разделах этой главы, показывают, что бесконтрольное выполнение операций, выполняемых для различных транзакций, может приводить к нежелательным результатам. Функции управления транзакциями реализуются в СУБД диспетчером транзакций. Диспетчер должен, с одной стороны, обеспечить выполнение некоторых критериев согласованности, чтобы исключить возможность появления аномалий, а с другой — обеспечить по возможности высокую пропускную способность системы за счет конкурентного выполнения транзакций.

Эти требования являются в некотором смысле противоположными: диспетчер, выполняющий все транзакции последовательно, обеспечивает идеальную корректность при очень низкой пропускной способности, и наоборот, диспетчер, который не накладывает никаких ограничений, дает высокую пропускную способность без каких-либо гарантий корректности.

Каждый диспетчер реализует некоторый протокол управления транзакциями, т. е. набор правил, определяющих условия выполнения операций транзакций.

Наиболее часто используемым протоколом является *протокол двухфазного блокирования* (two-phase locking, 2PL) в различных модификациях. Все протоколы этого семейства используют понятие *блокировки*: перед выполнением любой операции объект данных, который обрабатывает эта операция, должен быть заблокирован; при этом для разных операций используются разные типы блокировок. После выполнения операции (но не обязательно немедленно) блокировка должна быть снята. В простейших протоколах рассматриваются только операции чтения и записи, поэтому имеется всего два типа блокировок.

Блокировки являются несовместимыми, если:

- они связаны с одним объектом данных;
- устанавливаются для разных транзакций;
- по крайней мере, одна из двух операций является операцией записи.

Из этого следует, что блокировки разных транзакций, работающих с одним объектом данных, совместимы, только если обе операции являются операциями чтения.

По правилам работы с блокировками транзакция, которая пытается установить блокировку, не совместимую с уже установленной, переводится в состояние ожидания для тех пор, пока несовместимая блокировка не будет снята.

Важно отметить, что сами по себе блокировки и правила их установки никак не гарантируют корректности конкурентного выполнения транзакций.

Двухфазный протокол блокирования (2PL) состоит в том, что, после того как хотя бы одна блокировка была снята, рассматриваемая транзакция больше не может устанавливать новые блокировки. Другими словами, на первой фазе транзакция может устанавливать блокировки на те объекты данных, которые необходимы для ее выполнения, а на второй — только снимать ранее установленные блокировки.

Отметим, что хотя имеется возможность явного управления блокировками из приложения, обычно все необходимые блокировки устанавливаются и снимаются от имени транзакции автоматически. Более того, обычно приложение не должно явно управлять блокировками, т. к. диспетчеры могут использовать различные протоколы, в том числе протоколы, не основанные на применении блокировок, или использовать их не так, как требуется при соблюдении двухфазного протокола блокирования. При применении ослабленных критериев корректности (ослабленных уровней изоляции) блокировки могут устанавливаться не для всех операций или сниматься раньше, чем это предусмотрено протоколом 2PL.

6.5. Использование транзакций в приложениях

В соответствии с определением понятие согласованности зависит от логики приложения: система управления базами данных может реализовать свойства транзакций только в том случае, если приложение сообщает о том, какие именно операции составляют одну транзакцию. Для того чтобы это сделать, используют следующие операторы SQL.

START TRANSACTION указывает системе, что приложение начинает новую транзакцию. Такое же действие выполняет оператор BEGIN. В PostgreSQL этот оператор выдает сообщение об ошибке, если приложение уже выполняет транзакцию. Такое поведение соответствует стандарту SQL, однако некоторые другие СУБД в этом случае начинают вложенную подтранзакцию.

COMMIT сообщает о том, что все операции транзакции завершены и транзакцию необходимо зафиксировать. После фиксации приложение может начать новую транзакцию оператором BEGIN.

ROLLBACK вызывает обрыв текущей транзакции по инициативе приложения. СУБД откатывает все изменения, выполненные этой транзакцией, и завершает ее. После этого приложение может начать новую транзакцию.

Во всех трех операторах BEGIN, COMMIT и ROLLBACK можно указывать необязательное ключевое слово TRANSACTION, которое не оказывает никакого влияния на их работу. В следующем примере иллюстрируется использование операторов управления транзакциями:

```
demo=# BEGIN TRANSACTION;
BEGIN
demo=# SELECT *
FROM aircrafts
WHERE aircraft_code = '320';
 aircraft_code |      model       | range
---------------+------------------+-------
 320           | Аэробус A320-200 |  5700
(1 row)
demo=# UPDATE aircrafts
SET range = 6200
WHERE aircraft_code = '320';
UPDATE 1
demo=# SELECT *
FROM aircrafts
WHERE aircraft_code = '320';
 aircraft_code |      model       | range
---------------+------------------+-------
 320           | Аэробус A320-200 |  6200
(1 row)
demo=# ROLLBACK;
ROLLBACK
demo=# SELECT *
FROM aircrafts
WHERE aircraft_code = '320';
 aircraft_code |      model       | range
---------------+------------------+-------
 320           | Аэробус A320-200 |  5700
(1 row)
```

В этом примере выполняется обновление строки внутри транзакции, а затем эта транзакция обрывается. Операторы SELECT нужны, конечно, только для того, чтобы показать, как изменяется состояние базы данных. Заметим, что раз-

личие в результатах работы оператора SELECT не является индикацией аномалии нечеткого чтения, т. к. транзакция в данном случае видит изменения, выполненные ей же самой.

Отметим важную особенность транзакционного поведения PostgreSQL: обрыв транзакций не приводит к восстановлению использованных значений последовательностей. В приведенном ниже примере повторное выполнение той же самой транзакции дает другое значение для атрибута flight_id, несмотря на то что предыдущая транзакция была оборвана. Поскольку предполагаемое использование последовательностей — генерация уникальных значений, такое поведение системы корректно. Можно сказать, что обрывы транзакций возвращают базу данных в логически корректное состояние, эквивалентное ее состоянию до выполнения транзакции, но эти состояния не обязательно идентичны. Такое же поведение последовательностей реализуется и в других СУБД.

```
demo=# BEGIN;
BEGIN
demo=# INSERT INTO flights(
    flight_no, scheduled_departure, scheduled_arrival,
    departure_airport, arrival_airport, status, aircraft_code)
VALUES (
    'PG9999', now(), now() + interval '2 hours',
    'SVO', 'LED', 'Delayed', '321')
RETURNING flight_id;
 flight_id
-----------
     33122
(1 row)
INSERT 0 1
demo=# ROLLBACK;
ROLLBACK
demo=# BEGIN;
BEGIN
demo=# INSERT INTO flights(
    flight_no, scheduled_departure, scheduled_arrival,
    departure_airport, arrival_airport, status, aircraft_code)
VALUES (
    'PG9999', now(), now() + interval '2 hours',
    'SVO', 'LED', 'Delayed', '321')
RETURNING flight_id;
 flight_id
-----------
     33123
(1 row)
INSERT 0 1
```

```
demo=# ROLLBACK;
ROLLBACK
```

Такое поведение последовательностей позволяет несколько расширить возможности одновременного выполнения транзакций, использующих одну и ту же последовательность.

Если при выполнении оператора, находящегося внутри транзакции, происходит ошибка, то транзакция обрывается, и любая попытка выполнить какой-либо оператор (кроме оператора, завершающего транзакцию) в рамках той же транзакции, также приводит к индикации ошибки. При этом, конечно, пустой результат выполнения запроса или пустое множество изменяемых строк не считается ошибкой.

Если оператор SQL выполняется вне транзакции (например, приложение не выполняло оператор BEGIN), то поведение системы зависит от настроек клиента. Если установлен режим автоматической фиксации, каждый оператор превращается в отдельную транзакцию, которая не может быть оборвана по инициативе приложения, поскольку всегда завершается неявным оператором COMMIT (или ROLLBACK, если операция выполнена с ошибкой). Такой режим используется по умолчанию в psql. Если же автоматическая фиксация не установлена, то перед выполнением первого оператора транзакции выполняется неявная команда BEGIN.

6.6. Уровни изоляции

Полное выполнение всех ACID-свойств транзакций обеспечивает корректность совместного использования базы данных в том смысле, что работа каждого из одновременно (конкурентно) выполняемых приложений происходит так, как если бы с СУБД работало только одно приложение. Однако для реализации этих свойств сервер базы данных должен несколько ограничивать выполнение транзакций, что может приводить к снижению пропускной способности системы и к увеличению времени ожидания ответа для отдельных транзакций.

Это в особенности справедливо для простых протоколов управления транзакциями, основанных на использовании блокировок. Такие протоколы применялись в ранних реляционных СУБД и до сих пор используются в простых системах. Чтобы исключить снижение пропускной способности и сократить время ожидания приложений, возможны два пути:

- использовать другие, более сложные протоколы управления транзакциями, в меньшей степени снижающие производительность (пропускную способность) СУБД;
- полностью или частично отказаться от соблюдения ACID-свойств транзакций, т. е. снизить требования к согласованности данных.

По историческим причинам первые версии стандарта SQL были разработаны значительно раньше, чем появились высокопроизводительные протоколы управления транзакциями. По этим причинам стандарт SQL предусматривает возможности ослабления свойств ACID. В частности, ослабляются требования по изоляции транзакций, поэтому соответствующие режимы выполнения транзакций называются *уровнями изоляции*.

Некоторые из высокопроизводительных протоколов управления транзакциями реализованы в системе PostgreSQL. Заметим, что даже при использовании таких протоколов ослабление требований может приводить к некоторому увеличению производительности СУБД.

Уровни изоляции в стандарте SQL определены в терминах аномалий, т. е. для каждого из уровней указывается, какие из аномалий допустимы для этого уровня. Реализация СУБД, однако, может предотвращать появление некоторых аномалий, даже если они допускаются стандартом SQL для некоторого уровня.

Для всех уровней изоляции требуется атомарность, т. е. любая транзакция либо выполняется полностью, либо ее выполнение не оставляет никаких следов в базе данных. Кроме этого, для всех уровней изоляции не допускается аномалия потерянного обновления.

Стандартом предусмотрены следующие уровни изоляции:

Read Uncommitted разрешает доступ к результатам выполнения еще не зафиксированных транзакций и никак не ограничивает выполнения транзакций, тем самым допуская появление любых аномалий;

Read Committed — результаты других транзакций становятся доступными после их фиксации, т. е. запрещается аномалия грязного чтения;

Repeatable Read — повторное выполнение операций поиска и выборки данных дает такие же результаты, как первое, т. е. запрещаются аномалии грязного и нечеткого чтения;

Serializable требует, чтобы выполнение транзакций было эквивалентно некоторому последовательному выполнению.

В реализации PostgreSQL фактически используются только три последних уровня. Указание Read Uncommitted допускается, однако этот уровень работает точно так же, как Read Committed. Заметим, что в PostgreSQL применяются более сложные механизмы управления транзакциями, поэтому никакой потери производительности не происходит.

Уровень изоляции для отдельной транзакции указывается оператором SET TRANSACTION:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Возможно также указание (или изменение) уровня изоляции с помощью параметров сеанса или для всего сервера баз данных.

Если ничего не указывать, то PostgreSQL устанавливает уровень изоляции в соответствии со значением параметра default_transaction_isolation, обычно Read Committed. Это обеспечивает наиболее высокую пропускную способность, однако может приводить к появлению некоторых аномалий при обработке транзакций. В частности, возможно появление фантомов и аномалии нечеткого чтения.

Поскольку операторы поступают от приложений в произвольном порядке, СУБД не всегда может организовать выполнение транзакций, обеспечивая требуемый уровень изоляции. В этом случае транзакция, которую невозможно выполнить, обрывается по инициативе СУБД при попытке выполнения той операции транзакции, которую невозможно включить в общее расписание. В PostgreSQL приложение извещается об этой ситуации с помощью кода ошибки, возвращаемого при попытке выполнения операции. Такая ошибка может появляться, если приложение использует уровни Repeatable Read или Serializable. При ее появлении приложению следует повторить попытку выполнения транзакции заново (так, как будто она еще не выполнялась).

6.7. Точки сохранения

Полезность свойства атомарности состоит в том, что в случае неудачного выполнения некоторой транзакции частичные изменения, выполненные этой транзакцией, удаляются из базы данных и не мешают работе других транзакций. Следовательно, каждое приложение вправе ожидать, что оно начинает

работу с корректным (согласованным) состоянием базы данных. Однако ресурсы, затраченные на выполнение оборванной транзакции, оказываются безвозвратно потерянными. Это вполне допустимо для коротких транзакций, на выполнение каждой из которых затрачивается небольшое количество ресурсов, и если аварийные завершения случаются редко. В противоположность этому потеря результатов выполнения работы большого объема нежелательна.

Для транзакций, выполняющих большие объемы работы, может быть целесообразно использовать оператор SAVEPOINT, который запоминает состояние всех данных, измененных транзакцией к моменту его выполнения. В случае если при продолжении выполнения транзакции обнаруживается ошибка, можно восстановить состояние базы данных, выполняя операцию ROLLBACK с указанием имени того состояния, которое было предварительно записано оператором SAVEPOINT.

В следующем простом примере иллюстрируется работа операторов создания точек сохранения. В рамках транзакции выполняется изменение строки и создается точка сохранения; затем эта строка удаляется. После отката к точке сохранения с помощью оператора ROLLBACK вставленная строка снова появляется в базе данных. Наконец, вся транзакция обрывается — база данных восстанавливается в состояние, эквивалентное тому, которое было до начала транзакции.

```
demo=# BEGIN TRANSACTION;
BEGIN
demo=# SELECT *
FROM aircrafts
WHERE aircraft_code = '320';
 aircraft_code |      model       | range
---------------+------------------+-------
 320           | Аэробус A320-200 |  5700
(1 row)
demo=# UPDATE aircrafts
SET range = 6200
WHERE aircraft_code = '320';
UPDATE 1
demo=# SELECT *
FROM aircrafts
WHERE aircraft_code = '320';
 aircraft_code |      model       | range
---------------+------------------+-------
 320           | Аэробус A320-200 |  6200
(1 row)
```

```
demo=# SAVEPOINT svp;
SAVEPOINT
demo=# DELETE FROM aircrafts
WHERE aircraft_code = '320';
DELETE 1
demo=# ROLLBACK TO svp;
ROLLBACK
demo=# SELECT *
FROM aircrafts
WHERE aircraft_code = '320';
 aircraft_code |      model        | range
---------------+-------------------+-------
 320           | Аэробус A320-200 |  6200
(1 row)
demo=# ROLLBACK;
ROLLBACK
demo=# SELECT *
FROM aircrafts
WHERE aircraft_code = '320';
 aircraft_code |      model        | range
---------------+-------------------+-------
 320           | Аэробус A320-200 |  5700
(1 row)
```

Механизм точек сохранения дает возможность организовать достаточно сложное поведение в рамках одной транзакции, но для других транзакций ее поведение остается атомарным.

6.8. Долговечность

Свойство долговечности требует, в частности, чтобы СУБД гарантировала сохранность изменений, выполненных зафиксированными транзакциями, в случае системного отказа или отказа носителя. После перезапуска сервера база данных всегда оказывается в согласованном состоянии и содержит все изменения, выполненные зафиксированными транзакциями. Для того чтобы удовлетворить этим требованиям, СУБД выполняет некоторые дополнительные действия во время нормальной работы системы.

В большиствe систем (в том числе в PostgreSQL) для реализации этого свойства используются *журналы СУБД*, в которых последовательно записываются все изменения, выполненные всеми транзакциями, а также записи о фиксации

транзакций. Благодаря тому что запись в журнал ведется последовательно, такой способ обеспечения корректности восстановления оказывается наиболее эффективным.

Несмотря на относительную эффективность журналов, расходы на их ведение довольно значительны и в некоторых случаях в некоторых СУБД могут превышать затраты ресурсов на обновление основного хранилища СУБД. Может оказаться, что для части данных такая степень долговечности не нужна. Например, данные о состоянии сеанса связи с пользователем, скорее всего, очень часто обновляются, но становятся ненужными, когда сеанс прекращается. Поэтому можно повысить производительность системы, если исключить действия, обеспечивающие возможность восстановления этих данных в случае отказа системы.

В PostgreSQL есть возможность отказаться от регистрации изменений в журналах БД для отдельных таблиц. Заметим, что для таких таблиц ослабляется только один из аспектов долговечности, а именно — возможность восстановления после отказов. Поведение таких таблиц по отношению к фиксации транзакций не отличается от поведения обычных (журналируемых) таблиц.

6.9. Итоги главы

В этой главе приведены основные свойства транзакций и требования, которые предъявляются к их выполнению. Охарактеризованы методы, с помощью которых могут достигаться требуемые свойства, возможные компромиссы и последствия применения ослабленных критериев согласованности. Показаны операторы языка SQL, необходимые для управления транзакциями на уровне приложений.

6.10. Упражнения

Пример транзакции

Упражнение 6.1. Начните транзакцию (командой BEGIN) и создайте новое бронирование в таблице bookings сегодняшней датой. Добавьте два электронных билета в таблицу tickets, связанных с созданным бронированием.

Представьте, что пользователь не подтвердил бронирование и все введенные данные необходимо отменить. Выполните отмену транзакции и проверьте, что никакой добавленной вами информации действительно не осталось.

Упражнение 6.2. Теперь представьте сценарий, в котором нужно отменить не все данные, а только последний из добавленных электронных билетов. Для этого повторите все действия из предыдущего упражнения, но перед добавлением каждого билета создавайте точку сохранения (с одним и тем же именем). После ввода второго билета выполните откат к точке сохранения. Проверьте, что бронирование и первый билет остались.

Упражнение 6.3. В рамках той же транзакции добавьте еще один электронный билет и зафиксируйте транзакцию. Обратите внимание на то, что после этой операции отменить внесенные транзакцией изменения будет уже невозможно.

Уровень изоляции Read Committed

Упражнение 6.4. Перед началом выполнения задания проверьте, что в таблице bookings нет бронирований на сумму total_amount 1 000 рублей.

1. В первом сеансе начните транзакцию (командой BEGIN). Выполните обновление таблицы bookings: увеличьте total_amount в два раза в тех строках, где сумма равна 1 000 рублей.
2. Во втором сеансе (откройте новое окно psql) вставьте в таблицу bookings новое бронирование на 1 000 рублей и зафиксируйте транзакцию.
3. В первом сеансе повторите обновление таблицы bookings и зафиксируйте транзацию.

Осталась ли сумма добавленного бронирования равной 1 000 рублей? Почему это не так?

Уровень изоляции Repeatable Read

Упражнение 6.5. Повторите предыдущее упражнение, но начните транзакцию в первом сеансе с уровнем изоляции транзакций Repeatable Read. Объясните различие полученных результатов.

Упражнение 6.6. Выполните указанные действия в двух сеансах:

1. В первом сеансе начните новую транзакцию с уровнем изоляции Repeatable Read. Вычислите количество бронирований с суммой 20 000 рублей.
2. Во втором сеансе начните новую транзакцию с уровнем изоляции Repeatable Read. Вычислите количество бронирований с суммой 30 000 рублей.
3. В первом сеансе добавьте новое бронирование на 30 000 рублей и снова вычислите количество бронирований с суммой 20 000 рублей.
4. Во втором сеансе добавьте новое бронирование на 20 000 рублей и снова вычислите количество бронирований с суммой 30 000 рублей.
5. Зафиксируйте транзакции в обоих сеансах.

Соответствует ли результат ожиданиями? Можно ли сериализовать эти транзакции (иными словами: можно ли представить такой порядок последовательного выполнения этих транзакций, при котором будет получен тот же результат, что получился при параллельном выполнении)?

Уровень изоляции Serializable

Упражнение 6.7. Повторите предыдущее упражнение, но транзакции в обоих сеансах начните с уровнем изоляции Serializable.

Если вы правильно ответили на его последний вопрос, вы поймете, почему теперь эти действия приводят к ошибке. Если же результат этого упражнения стал для вас неожиданностью, четко сформулируйте различие уровней Repeatable Read и Serializable.

Вложенные транзакции

Упражнение 6.8. Некоторые СУБД (но не PostgreSQL) позволяют использовать вложенные транзакции. Если начать вторую транзакцию, не завершая уже открытую первую, то вторая транзакция будет считаться *вложенной*: ее результат фиксируется только в том случае, если фиксируется первая транзакция, но при этом ее результаты можно отменить независимо от первой транзакции.

Реализуйте такое поведение для PostgreSQL, т. е. предложите, какие команды следует выполнять для открытия вложенной транзакции, для ее отмены и для фиксации. Подсказка: используйте оператор SAVEPOINT.

Глава 7

Разработка приложений СУБД

Технологии разработки приложений не являются частью технологий баз данных, которые составляют основное содержание этой книги. Однако нельзя обойти вопросы, связанные с разработкой приложений, поскольку на сегодняшний день базы данных практически не используются сами по себе, а являются хранилищем для бизнес-приложений. Обычно пользователь работает с приложением через графический интерфейс, специально разработанный для решения задач предметной области. Интерфейс скрывает детали реализации системы, и пользователю не нужно знать ни код приложения, ни тем более структуру базы данных, и не нужно уметь применять язык запросов.

Методы разработки приложений относятся к группе дисциплин, составляющих программную инженерию, в которой применяются принципы и критерии, существенно отличающиеся от критериев, применяемых при разработке баз данных. В частности, обычно наиболее важными критериями оказываются стоимость и время, затраченные на разработку приложения, а не критерии производительности, рассмотренные в главе 1. Опытные разработчики хорошо знают, что в начале каждого проекта по созданию приложения им необходимо сделать выбор между следующими альтернативами:

- тщательное проектирование структуры базы данных и запросов на основе анализа требований;
- использование программных средств, обеспечивающих быструю разработку приложений.

Хорошо известно, что первый вариант обеспечивает высокое качество и высокую производительность, а второй — быстрое получение результатов и низкую стоимость, достигаемые за счет многочисленных компромиссов, ухудшающих качество и производительность результата. В подавляющем большинстве случаев руководство выбирает второй вариант.

Интересный анализ применения различных подходов к проектированию приложений, работающих с базами данных, содержится в докладе [15].

Наиболее серьезные проблемы, связанные с реализацией приложений, работающих с базами данных, так или иначе оказываются следствиями или проявлениями общей проблемы различия вычислительных моделей и моделей данных, применяемых в доминирующих языках программирования приложений и в системах управления базами данных. Это несоответствие известно под метафорическим названием «impedance mismatch». Как СУБД, так и языки программирования могут работать в терминах высокоуровневых абстракций, однако системы этих абстракций различаются, и, что еще важнее, взаимодействие между этими моделями обычно описывается в терминах низкоуровневых интерфейсов.

Задачи предметной области реализуются при помощи программного кода, так называемой бизнес-логики приложения. В программной инженерии плохой практикой считается программирование бизнес-логики на уровне пользовательского интерфейса, поскольку требования к интерфейсу меняются наиболее часто. Это удорожает разработку, осложняя повторное использование кода и применение уже готовых и протестированных сторонних (third-party) компонентов. Чтобы минимизировать влияние изменений требований, применяется архитектура приложения, включающая тонкого клиента, единственной задачей которого становится отображение данных.

Независимо от используемых технологий хорошо спроектированное (с точки зрения, принятой в программной инженерии) приложение имеет слой, абстрагирующий доступ к данным (data access layer). Такой подход позволяет ограничить доступ к данным на уровне программной архитектуры и дает возможность заменить СУБД на какое-нибудь альтернативное решение, не затрагивая бизнес-логику. Возможность работы с разными СУБД может быть полезна для тиражируемых программных продуктов, но совсем не имеет значения при разработке конкретных информационных систем и, более того, может приводить к снижению качества приложения. Заметим также, что этот аргумент игнорирует необходимость миграции данных, которая может оказаться более дорогостоящим проектом, чем разработка приложения. Другая проблема состоит в том, что слой, абстрагирующий доступ к данным, зачастую полностью блокирует доступ приложения к высокоуровневым и высокоэффективным средствам, предоставляемым СУБД.

Другим крайним вариантом является размещение бизнес-логики в хранимых процедурах. По данным, приведенным в [15], такое решение применяется примерно в 11 % приложений баз данных. Это позволяет использовать общий код, выполняемый в базе данных, в нескольких приложениях и создает благоприятные условия для эффективной реализации и настройки этих функций. Одно-

временно появляется возможность сохранить бизнес-логику при переходе на другой язык программирования приложения (хотя возможность смены языка программирования обычно не рассматривается). Такой подход снижает зависимость кода приложения от схемы базы данных, а также повышает производительность, однако приводит к существенному увеличению стоимости разработки.

В этой главе мы коснемся вопросов программирования приложений, взаимодействующих с базами данных. Несмотря на то что подробное освещение этой темы выходит за рамки данного курса, мы познакомим вас с основными технологиями доступа к данным и расскажем о проблемах интеграции баз данных и приложений.

7.1. Проектирование схемы базы данных

Методологии проектирования информационных систем, развитые в течение последних десятилетий XX века, предусматривали построение различных моделей, в том числе моделей данных, на основании анализа предметной области. Полученные модели использовались для создания схемы базы данных.

С развитием технологий многие организации стали ощущать потребность в документировании своих внутренних бизнес-процессов. Бизнес-процесс — это комплекс взаимосвязанных действий, направленных на создание продукта или услуги, представляющих ценность для потребителей. В задачах автоматизации предприятия описания бизнес-процессов используются в качестве требований к будущим информационным системам. Языки описания бизнес-процессов не моделируют данные, а описывают сообщения, которые являются основным способом коммуникации между участниками процесса.

Наряду с моделированием бизнес-процессов развитые методологии проектирования предусматривают создание целого ряда других моделей, в том числе диаграмм потоков данных и моделей данных, а также перекрестную проверку соответствия этих моделей. Однако создание таких моделей хотя и повышает качество проектируемой системы, но существенно увеличивает сроки разработки и ее стоимость. Поэтому дополнительные модели создаются редко, а современные инструменты не поддерживают их совместное применение. Это означает, что вы не можете нарисовать диаграмму бизнес-процессов и связать ее с моделью данных. На практике приходится использовать оба типа моделирования, а связывание производить вручную.

В этом курсе нас интересует построение моделей данных. Процесс моделирования начинается со сбора и анализа требований. На основании требований строится концептуальная модель, описывающая основные сущности информационной системы и взаимосвязи между ними. Далее множество сущностей концептуальной модели уточняется, к ним добавляются атрибуты и ограничения, и в результате создается логическая модель системы.

Логическая модель системы содержит все сведения, необходимые для написания программного кода. При этом существенно изменяется точка зрения на роль СУБД: если традиционно база данных рассматривалась как разделяемый ресурс, используемый несколькими приложениями, то при проектировании на основе логической модели предполагается, что база данных доступна только через монолитное приложение, созданное на основе логической модели. В результате создается иллюзия того, что многие средства СУБД, такие как независимое описание данных и поддержка конкурентного доступа, становятся ненужными. Отказ от использования функций СУБД вынуждает добавлять компоненты, дублирующие эти функции вне СУБД, что, по-видимому, не удешевляет систему, однако на той фазе жизненного цикла системы, когда необходимость в этих функциях становится очевидной, изменение архитектуры, как правило, оказывается уже невозможным.

На практике данные хранятся в реляционных СУБД, а программы, обрабатывающие эти данные, пишутся на современных объектно-ориентированных языках программирования, не приспособленных для использования высокоуровневых возможностей СУБД. То есть из логической модели нужно получить и схему базы данных, представляющую данные в виде реляционных таблиц, и объектную модель данных.

При проектировании схемы базы данных неизбежны компромиссы, обычно ухудшающие свойства базы данных, но упрощающие согласование с объектной моделью приложения. В частности, схемы низкого качества получаются при использовании инструментов, реализующих объектно-реляционные отображения. Причины, по которым компромиссы необходимы, связаны со следующими различиями в свойствах моделей.

Идентификация. В моделях данных СУБД применяется естественная идентификация, а в объектных моделях приложений — идентификация на основе суррогатов. Как правило, это приводит к включению суррогатных ключей, никак не связанных с предметной областью, в логическую модель данных и в схему базы данных.

Следствием такого решения может оказаться появление скрытых дубликатов вследствие ошибок в коде приложения, а также других дефектов данных, необходимость обработки которых может приводить к существенному усложнению кода приложения.

Представление связей. В моделях СУБД используются ассоциативные связи, основанные на значениях атрибутов и устанавливаемые динамически во время выполнения запросов, а в объектных моделях используются статические связи, представленные суррогатными объектными указателями.

Для того чтобы приблизить структуру базы данных к структуре модели приложения, связи в базе данных (первичные и внешние ключи) определяются в терминах суррогатов, и, таким образом, ответственность за корректность связей также возлагается на приложение.

Другие различия в моделях в меньшей степени влияют на схему БД, однако существенно влияют на вид запросов, обеспечивающих доступ к данным.

Современные методы разработки предполагают итеративный подход. Многие детали будут обнаружены после построения первых версий модели, а возможно, и после создания первых работающих версий системы. Изменения придется вводить в течение всего проекта, и необходимо будет поддерживать все модели в согласованном состоянии.

Для ускорения разработки применяется генерирование программных артефактов. Инструменты проектирования позволяют генерировать из логической модели объекты программного кода, файлы с объектно-реляционными отображениями и схему базы данных. Сгенерированные классы, как правило, не требуют серьезных изменений, но схему базы данных и объектно-реляционные отображения нельзя получить полностью автоматически, требуется ручная настройка.

Возможный вариант компромисса между требованиями методологий и систем разработки приложений (в первую очередь — каркасов) и требованиями, обеспечивающими высокое качество проектируемой схемы, получается при использовании следующих правил.

- Необходимо синхронизировать логическую и объектную модели. Логическая модель приложения — это артефакт, который будет использоваться не только программистами, но также аналитиками или тестировщиками, которые незнакомы с тонкостями ООП или реляционной теории. Необходимо, чтобы логическая модель всегда была актуальной.

- Первую версию объектно-реляционных отображений можно получить при помощи генерирования, однако затем необходимо уточнить типы, ограничения, отображения взаимосвязей и наследования.
- Схему базы данных нужно создавать или модифицировать при помощи скриптов SQL, используя типы и особенности, поддерживаемые выбранной СУБД. Генерирование при помощи каркасов, как правило, не бывает однозначным, и, генерируя схему в разных окружениях, можно получать разные результаты (например, порядок колонок в таблице).

Последняя из перечисленных рекомендаций связана с тем, что обычно каркасы не могут модифицировать уже существующую схему базы данных, а генерируют ее заново. При этом, конечно, игнорируются основные принципы СУБД, в частности принцип разделения (независимости) данных и программ, из которого следует, что порядок колонок в таблицах не имеет никакого логического значения.

В современном мире каркасы для разработки приложений появляются намного чаще, чем меняются технологии работы с данными. Разработанная система через некоторое время устареет и может быть переписана с помощью новых языков программирования и библиотек; при этом данные останутся и будут перенесены в новую систему. Развертывание новой версии приложения — это, как правило, быстрая и дешевая операция, которая проводится в реальном времени, а в случае неудачи можно вернуться к предыдущей версии. Миграция данных занимает значительно больше времени, а ошибки в миграции данных приводят к серьезным сбоям и даже остановкам системы. Поэтому в разработке информационных систем вопрос проектирования данных является одним из самых важных.

7.2. Объектно-реляционная потеря соответствия

Одновременное использование объектно-ориентированной и реляционной парадигм вызывает определенные трудности. В процессе проектирования мы получаем логическую модель информационной системы, которая, вообще говоря, не привязана к конкретным технологиям. На следующем этапе нам нужно получить модель, которую будет использовать приложение, и модель, представляющую данные в СУБД. Эти модели имеют серьезные различия, и здесь мы сталкиваемся с проблемой, которая называется *объектно-реляционной потерей соответствия* (object-relational impedance mismatch).

При обсуждении потери соответствия обычно указывают следующие различия.

Идентификация объекта. В реляционной базе данных запись идентифицируется своим первичным ключом, который является частью состояния записи. В объектном мире идентификация объекта не зависит от его состояния, несколько объектов с одинаковым состоянием могут иметь разную идентификацию. При проектировании приложений возникает вопрос, обсуждаемый в главе 2: как однозначно идентифицировать сущность в обеих парадигмах.

Системы типов. Типы данных в языках программирования не соответствуют типам в реляционных базах данных. Например, отличаются представление строк и диапазон числовых значений. Также объектно-ориентированные языки не поддерживают неопределенных значений для примитивных типов.

Этот вид различий моделей оказывается важным потому, что разработчики приложений игнорируют возможность задания ограничений целостности, поскольку такие ограничения трудно формулировать в объектных моделях. Так, ограничения на диапазоны значений зависят от предметной области приложения, а не от возможностей языка программирования, и могут быть легко выражены в схеме базы данных (например, определением доменов в PostgreSQL), но не могут быть выражены в объектной модели приложения.

Навигация между объектами. В отличие от реляционных систем, основанных на ассоциативных связях, объектные модели поддерживают возможность навигации с помощью ссылок. Если приложение выбирает множество записей и отображает их в объекты, нужно ли выбирать также объекты, на которые ссылается первое множество? Что делать, если бизнес-процесс требует только часть состояния объекта?

В действительности эти вопросы не связаны с несоответствием моделей, т. к. они сформулированы полностью в терминах объектов. Проблема здесь в том, что при использовании каркасов запросы на выборку данных генерируются автоматически, а логическая модель не дает достаточной информации о том, какие части или какие другие объекты необходимо выбирать из базы данных.

Навигационный поиск объектов может приводить и к более серьезным осложнениям, связанным с проверкой значений атрибутов.

Инкапсуляция. Состояние объекта изменяется только при вызове его методов. Считается, что вызов метода, выполняющего присваивание значения, безопаснее, чем оператор присваивания, потому что код метода может выполнить дополнительные проверки. Считается, что базы данных не имеют такой защиты и содержание записей может быть изменено сторонними приложениями. В базах данных аналогичную роль выполняют ограничения целостности и триггеры, а возможность доступа из разных приложений необходима для совместного использования данных.

С другой стороны, обязательность вызова методов приводит к тому, что объекты могут обрабатываться только по одному (метод объекта не применим к множеству), а высокая эффективность баз данных достигается, если запросы обрабатывают множества объектов.

Снова проблема не в том, что в объектных моделях есть нечто такое, что невозможно сделать в моделях баз данных, а в том, что эти модели различаются.

Различные библиотеки и каркасы пытаются решить вышеуказанные проблемы и сделать работу с базой данных прозрачной для программистов, занимающихся разработкой приложения. Однако необходимо заметить, что большая часть перечисленных осложнений возникает именно потому, что авторы каркасов пытаются добиться «прозрачности», игнорируя особенности моделей баз данных.

Намного более серьезные осложнения возникают вследствие других различий в моделях:

- объектные языки ориентированы на обработку объектов по одному, а реляционная модель данных оперирует множествами;
- ассоциативный поиск и связывание по значениям атрибутов, применяемые в реляционной модели данных, плохо согласуются с навигацией по объектным указателям.

Эти различия не только усложняют разработку кода приложения, но и могут приводить к ухудшению характеристик производительности вследствие низкой эффективности приложения, написанного без учета этих различий.

7.3. Использование каркасов объектно-реляционных отображений

Каркасы *объектно-реляционных отображений* (object-relational mapping, ORM) определяют, каким образом объекты приложения будут сохранены в таблицах базы данных, и позволяют выбирать записи, используя специальный объектный язык запросов.

Рассмотрим простой случай, когда у класса нет предков и потомков, а его атрибуты имеют примитивные типы. Такой класс отображается в отношение, а атрибуты класса — в атрибуты отношения. При этом имена классов и атрибутов сохраняются или преобразуются согласно правилам именования. Для атрибута отношения каркас отображения выбирает тип, максимально соответствующий типу атрибута класса. Также на уровне отображения устанавливаются ограничения, не поддерживаемые напрямую в объектных языках программирования. Например, допускаются ли неопределенные значения, является ли атрибут уникальным и из каких атрибутов состоит первичный ключ.

Каркасы минимизируют труд программистов по разработке кода приложений, что отвечает основному критерию программной инженерии. Однако получаемый таким способом код обычно оказывается низкоэффективным, и созданные приложения могут работать только с небольшими базами данных и при невысокой интенсивности их использования.

7.3.1. Наследование

Одним из наиболее важных и широко используемых понятий объектно-ориентированного подхода является понятие наследования. Существует большое количество разных моделей для описания и для применения наследования. Неформально наследование используется для указания того факта, что свойства объектов из некоторого множества имеются также у объектов подмножества этого множества. Это не только дает возможность исключить повторное описание данных свойств, но и позволяет обрабатывать объекты меньшего множества вместе с объектами большего. Например, свойства всех автомобилей применимы и к легковым автомобилям.

Поскольку в императивных объектно-ориентированных языках программирования наследование применяется довольно часто, возникает задача отображения наследования при использовании нескольких различающихся моделей,

в частности при организации взаимодействия между базой данных и приложением.

Реляционная теория не поддерживает напрямую понятие наследования, а наследование в модели данных «сущность-связь», а также наследование, реализованное в современных системах (в том числе в PostgreSQL), существенно отличаются от принятых в объектных моделях языков программирования. В этой главе обсуждаются варианты отображения структур наследования объектно-ориентированных языков в базу данных в предположении, что СУБД не поддерживает понятия наследования. Такие отображения важны, поскольку каркасы не используют средств наследования СУБД, даже если они имеются. Возможности применения наследования в системе PostgreSQL обсуждаются ниже в главе 8.

Для иллюстрации отображения иерархии наследования в табличную форму используем предельно упрощенный пример. Пусть класс person содержит информацию обо всех лицах, связанных с учебным заведением, подкласс professor содержит информацию о преподавателях, а подкласс student содержит информацию о студентах. Преподаватели имеют дополнительный атрибут — год присвоения ученой степени, а студенты — номер студенческого билета.

Для отображения наследования объектной модели на упрощенную табличную модель используется одна из следующих стратегий.

Отображение иерархии классов в одну таблицу. Все атрибуты классов в иерархии записываются в одну таблицу. Один специальный атрибут (дискриминатор) указывает, какой именно класс объекта соответствует записи.

Пример такого представления показан на рис. 7.3.1.

persons

class	name	title	degree	stud_id
person	Елена	секретарь		
person	Анастасия	лаборант		
professor	Аристарх	профессор	2001	
professor	Ксенофон	доцент	2012	
student	Анна	студент		1451
student	Виктор	студент		1432
student	Нина	студент		1556

Рис. 7.3.1. Представление иерархии наследования в одной таблице

При этом для выбора объекта не надо использовать операции соединения, что в некоторых случаях является преимуществом.

Недостатком подхода является то, что нельзя задавать ограничения целостности NOT NULL для атрибутов подклассов, поскольку другие подклассы будут иметь неопределенное значение в соответствующих колонках, а также многие колонки будут иметь пустые значения. Возможность записи неопределенных значений можно регулировать с помощью ограничений целостности CHECK, включающих значения атрибута-дискриминатора, но каркасы не генерируют ограничения такого типа.

Кроме этого, извлечение объектов некоторого подкласса может оказаться вычислительно неэффективным. Например, операция полного просмотра списка профессоров (относительно небольшое множество) потребует просмотра всей информации из таблицы persons (которая по размеру значительно больше).

Горизонтальное разделение на таблицы. Атрибуты класса, а также унаследованные атрибуты отображаются в колонки таблицы, при этом для каждого класса используется отдельная таблица.

Пример такого представления показан на рис. 7.3.2.

persons

name	title
Елена	секретарь
Анастасия	лаборант

professors

name	title	degree
Аристарх	профессор	2001
Ксенофон	доцент	2012

students

name	title	stud_id
Анна	студент	1451
Виктор	студент	1432
Нина	студент	1556

Рис. 7.3.2. Горизонтальная фрагментация таблиц при наследовании

В этом случае для загрузки объектов тоже не надо выполнять операцию соединения. Однако метод поиска во всей иерархии классов должен будет просмотреть все таблицы, соответствующие подклассам (т. е. выполнить

операцию UNION в терминах SQL). Как и в предыдущем варианте, идентификация объектов должна быть уникальна в иерархии.

Этот метод представления наследования близок к модели наследования, которая используется в системе PostgreSQL, и поддерживается расширенным синтаксисом SQL, однако каркасы для разработки приложений не используют возможности СУБД.

Вертикальное разбиение. Каждый класс отображается в таблицу и хранит только атрибуты своего класса (без унаследованных). Атрибут, являющийся первичным ключом, определяется в корневом классе. Таблицы, соответствующие подклассам, ссылаются при помощи внешнего ключа на таблицы, соответствующие предкам. Для загрузки объекта нужно выполнить операцию соединения.

Пример такого отображения показан на рис. 7.3.3.

persons

name	title
Елена	секретарь
Анастасия	лаборант
Аристарх	профессор
Ксенофон	доцент
Анна	студент
Виктор	студент
Нина	студент

professors

name	degree
Аристарх	2001
Ксенофон	2012

students

name	stud_id
Анна	1451
Виктор	1432
Нина	1556

Рис. 7.3.3. Фрагментация таблиц при наследовании в модели ER

Такой способ наследования применяется в модели данных «сущность-связь», если внешние ключи в таблицах, соответствующих подклассам, являются также первичными ключами в этих таблицах. В некоторых языках программирования, однако, для этого могут создаваться отдельные атрибуты, содержащие суррогатные ключи.

Возможны и другие варианты отображения. Например, в последнем варианте можно избыточно хранить в каждой таблице все атрибуты соответствующего класса, а не только специфические для этого класса. Такая избыточность может ускорить выполнение операций выборки данных, но, конечно, потребует дополнительных затрат при модификации данных.

7.3.2. Запросы

Для чтения объектов из базы данных используется объектный язык запросов, который обычно эквивалентен небольшому подмножеству SQL с несколько отличающимся синтаксисом. Запросы в нем формулируются в терминах объектной, а не реляционной модели и не зависят от схемы базы данных, поскольку отображение строится каркасом автоматически при преобразовании этих запросов в SQL. Запрос может быть представлен в виде строки или иметь объектную структуру. Результатом выполнения такого запроса является (в терминах теории графов) лес, каждое дерево в котором соответствует некоторому объекту верхнего, по отношению к запросу, уровня и некоторым объектам, достижимым из него через объектные ссылки.

Поскольку навигация между объектами осуществляется с помощью ссылок, важно определить, будут ли выбираться также объекты, на которые ссылаются объекты верхнего уровня. Такие объекты можно или загружать всегда, или использовать «ленивую загрузку», т. е. загружать объект, когда происходит доступ по ссылке. В случае ленивой загрузки доступ к каждому свойству навигации приводит к выполнению отдельного запроса к базе данных. Здесь можно столкнуться с проблемой, которая называется *проблемой эн плюс одной выборки*. Если мы выбираем объект, который ссылается на n объектов, то для получения всей структуры нам понадобится выполнить $n+1$ запрос. Однако в случае доступа ко всем сопряженным объектам эффективнее было бы выбрать всю структуру за один запрос. Важно понимать, как будет происходить доступ к ассоциированным объектам, потому что их загрузка серьезно влияет на производительность приложения.

Многие объектные языки запросов не поддерживают пакетного обновления или удаления объектов, что создает определенные неудобства и плохо влияет на производительность. Кроме этого, выделение слоя абстракции данных практически исключает возможность выполнения обновлений без считывания данных в память приложения, что во многих случаях удваивает количество запросов, выполняемых приложением.

7.3.3. Когда применять каркасы?

Каркасы объектно-реляционных отображений сокращают усилия, которые необходимо приложить программисту. Однако они работают в рамках унифицированного и сильно ограниченного подмножества стандарта SQL 1992 года

и не используют новых возможностей, предусмотренных более поздними версиями стандарта, и тем более расширений, специфических для СУБД. Не используются также многие конструкции SQL. Однако основной причиной низкой эффективности приложений, разработанных с использованием каркасов, оказывается слишком большое количество слишком мелких запросов. Эффект $n + 1$ запроса повторяется на каждом уровне иерархии объектов, и количество запросов, очевидно, растет экспоненциально с ростом глубины вложенности. Фактически при этом высокоуровневые операции базы данных (такие как соединение и агрегирование) выполняются кодом приложения, разумеется, без какой-либо оптимизации. Известны случаи, когда для формирования HTML-страницы приложение выполняло десятки тысяч запросов — в несколько раз больше, чем размер генерируемой страницы в байтах.

Все это приводит к тому, что части приложения, реализация которых с применением каркаса наименее эффективна, приходится переделывать с использованием запросов на SQL, забывая о правилах хорошего тона, рекомендуемых учебниками по программной инженерии. Поскольку проблемы производительности обычно возникают в частях приложения с наиболее сложными функциями, говорить о качестве кода не приходится. Получаемый при этом результат нельзя считать удовлетворительным и с точки зрения администратора базы данных, т. к. настройка запросов, собираемых динамически из констант, рассеянных по коду приложения, оказывается трудно выполнимой.

Можно сказать, что применение каркасов оправдано для быстрой реализации несложных приложений или частей приложений, для которых производительность не критична. Если ожидается, что проектируемая система будет использоваться с большой интенсивностью и, следовательно, создавать большую нагрузку, то целесообразно применить более сложные методы, обеспечивающие высокую эффективность работы системы.

7.4. Кеширование данных

Применение каркасов, как правило, приводит к тому, что приложение выполняет очень много небольших запросов, что создает повышенную нагрузку на вычислительную сеть, потому что каждый запрос выполняется синхронно. В результате приложение подавляющую часть времени находится в состоянии ожидания результатов выполнения запросов, несмотря на крайне низкую загруженность сервера БД. Другими словами, обе компоненты (приложение

и сервер) практически все время находятся в состоянии ожидания. Единственным радикальным решением проблемы является использование сложных запросов. Обычно это приводит к сокращению времени выполнения приложения на 2–3 порядка, однако требует существенных затрат труда. Поэтому на практике применяются приемы, не решающие основную проблему, но несколько снижающие отрицательные последствия. Зачастую при этом создаются новые проблемы.

Одним из таких приемов является кеширование на уровне приложения.

Современные высокопроизводительные СУБД широко используют самые разные виды кеширования, причем использование этих средств не требует никаких усилий со стороны программистов приложений. Использование кеша БД означает, что при выборке данные будут по возможности браться из буферного кеша, а не считываться с дисков, ранее выполнявшиеся запросы не обязательно будут повторно оптимизироваться, а результаты их выполнения сохраняются для возможного повторного использования. Однако, как правило, сервер баз данных физически расположен на отдельном компьютере, и кеширование на стороне БД не может снизить нагрузку на вычислительную сеть. Поэтому для приложения, выполняющего огромное количество мелких запросов, время ожидания ответа не может существенно сократиться.

Сэкономить это время помогает кеш на уровне приложения. Как правило, программисты не уделяют достаточного внимания вопросам кеширования, считая, что кеширование можно добавить потом, если возникнут проблемы с производительностью. Часто это связано с непониманием принципов работы кеша и приводит к ошибкам в приложении.

Приложение работает с графом объектов, имеющим сложную структуру. Один и тот же объект может быть выбран из базы данных разными способами, и это может привести к появлению дубликатов: нескольких объектов с одинаковыми идентификаторами и разными состояниями. Поэтому для корректной работы приложения необходимо либо использовать сложные запросы, либо следить за идентичностью всех загружаемых объектов на уровне приложения, и часто эта задача делегируется кешу.

Возникает вопрос: что произойдет, если разные пользовательские сеансы будут обращаться к одним и тем же объектам. Задача обеспечения корректности полностью решается средствами управления транзакциями на уровне СУБД, однако при использовании кеширования в приложении возникает необходимость решать эти задачи заново.

Существуют различные стратегии параллельного доступа к кешированным данным.

Транзакционная. Такой кеш связан с транзакцией и гарантирует, что состояние объектов является актуальным и обновления объектов в одной транзакции проводятся атомарно.

Чтение-запись. При изменении объекты блокируются в кеше, а при фиксации транзакции в БД блокировки снимаются. Если другая транзакция пытается получить заблокированный объект, то считается, что объекта нет в кеше и запрос перенаправляется к БД, которая отвечает в зависимости от установленного уровня изоляции транзакции.

Нестрогое чтение-запись. Согласованность между кешем и БД не поддерживается. Данные могут быть обновлены несколькими транзакциями без гарантии результата. После фиксации транзакции данные снова считываются из базы данных.

Только чтение. Предполагается, что данные не изменяются, и поэтому в кеше они не обновляются. Такую стратегию можно использовать для справочных данных.

Каркасы объектно-реляционных отображений обычно поддерживают два типа кеширования, дублируя функции СУБД.

Сеансовый кеш, или кеш первого уровня. Такой кеш является транзакционным. Выбранные во время сеанса объекты помещаются в кеш и при повторной выборке возвращаются из кеша без обращения к БД. Гарантируется, что если во время одного сеанса объект будет запрошен два раза, то вернется один и тот же объект приложения.

Разделяемый кеш, или кеш второго уровня. Этот кеш доступен всем клиентам приложения. При выборке объекта каркас проверяет, не находится ли этот объект в кеше первого уровня, а затем, если объект не найден, запрашивает кеш второго уровня. Этот кеш не обязательно является транзакционным и может содержать устаревшие данные. Для такого кеша нужно явно указать стратегию параллельного доступа и стратегию, по которой данные устаревают и удаляются из кеша.

Каркасы объектно-реляционных отображений поддерживают сеансовый кеш по умолчанию для всех объектов, а для кеша второго уровня нужно явно указать классы, которые будут кешироваться.

Кеш приложения может работать не только на уровне объектов, но и на уровне результатов запросов. В случае кеширования запросов ключом является запрос со всеми параметрами. Как правило в кеше хранятся не сами выбранные объекты, а их идентификаторы, поэтому кеш запросов используется вместе с кешем второго уровня. Если в кеше второго уровня не будет нужных объектов, то они будут выбираться из БД по одному, что плохо влияет на производительность.

При использовании кеширования могут возникнуть следующие проблемы:

- если базу данных обновляют другие приложения, то данные в кеше становятся некорректными, поскольку кеш не знает об этих изменениях;
- при загрузке большого количества объектов может не хватить оперативной памяти;
- если ограничения доступа к данным реализуются на уровне БД, то те же самые ограничения должны быть реализованы на уровне кеша (т. е. функции СУБД должны быть продублированы).

Правильно настроенный механизм кеширования эффективно работает для одного приложения и в пределах одного сервера приложений. При использовании нескольких серверов приложений задача синхронизации кешированных данных становится весьма сложной.

7.5. Взаимодействие с базой данных

Любой каркас генерирует запросы на языке SQL и передает их для выполнения на сервер СУБД через предназначенный для этого интерфейс. Любое приложение, как построенное на основе каркаса, так и без него, может формировать запросы SQL и передавать их для выполнения через такой же интерфейс.

Рассмотрим способы взаимодействия приложения с базой данных без использования каркасов объектно-реляционных отображений.

7.5.1. Параметры запросов

Способ оформления и передачи оператора SQL из приложения на сервер БД зависит от языка программирования и от типа интерфейса, но в любом случае

в конечном итоге запрос передается на сервер базы данных в виде текстовой строки, возможно, с параметрами.

Текстовое представление оператора, переданное на сервер, подвергается предварительной обработке, зачастую довольно сложной. *Этап подготовки* включает синтаксический анализ и проверку на соответствие схеме базы данных, оптимизацию и генерацию кода. После подготовки следует *этап интерпретации (выполнения)*, на котором полученный код исполняется с использованием параметров оператора и полученные результаты оформляются для передачи в клиентскую программу.

Обычно параметры могут задавать значения для операций поиска в базе данных. В этом случае оператор будет находить разные объекты в базе данных в зависимости от значений переменных программы, переданных в качестве параметров оператора.

Этапы подготовки и интерпретации оператора можно выполнять отдельно друг от друга. Это может быть полезно, если один и тот же оператор предполагается использовать с несколькими разными наборами значений параметров. Альтернативный метод состоит в совмещении подготовки и выполнения в одном обращении к СУБД.

Выбор между непосредственным выполнением операторов и предварительной подготовкой операторов зависит от многих факторов. Использование подготовленных операторов при их многократном выполнении позволяет исключить затраты на повторную подготовку оператора. С другой стороны, при непосредственном выполнении оптимизатору известны значения всех параметров оператора, что потенциально дает возможность генерировать разные планы. Это может быть важно при неравномерном распределении значений атрибутов, на которые заданы условия фильтрации. Заметим, что далеко не каждый оптимизатор эту возможность использует.

С другой стороны, при непосредственном выполнении запросы, отличающиеся только значениями констант, оказываются разными, что препятствует их кешированию (на сервере базы данных). Для того чтобы повысить эффективность кеширования запросов, некоторые СУБД могут заменять константы, заданные в запросе, на параметры. В этом случае новый запрос, отличающийся только значениями констант, будет найден среди ранее выполненных, т. е. фактически непосредственное выполнение не будет отличаться от выполнения подготовленного запроса.

В некоторых системах, в том числе в PostgreSQL, в кеш обычно заносится не готовый для выполнения план, а только результат синтаксического разбора. В этом случае оптимизатор может генерировать различающиеся планы в зависимости от фактических значений параметров подготовленного запроса, поскольку во время оптимизации значения параметров уже известны.

В отличие от запросов, полученных конкатенацией строк, содержащих части запроса и значения параметров, применение параметризованных (подготовленных) запросов исключает возможность их изменения при передаче значений параметров. Эта особенность важна для предотвращения атак типа внедрения SQL-кода.

После выполнения оператора необходимо получить результаты в переменные программы-клиента. Многие операторы (например, SELECT и любые операторы обновления, содержащие предложение RETURNING) возвращают множество объектов, удовлетворяющих критериям поиска. Для того чтобы получить результаты полностью, необходимо выполнить цикл, в теле которого обрабатывается очередная строка результата.

7.5.2. Унифицированные средства взаимодействия

Обычно для доступа приложения к базе данных применяются унифицированные драйверы (библиотеки), которые устанавливают соединение с сервером БД, посылают SQL-запросы в текстовом формате и возвращают приложению выбранные данные. Унифицированные драйверы имеют интерфейс, который не зависит от конкретной СУБД и реализует ограниченное подмножество SQL. Некоторые драйверы допускают также использование любых возможностей SQL, реализованных в СУБД. Способность работы с разными СУБД может быть полезна для «коробочных» продуктов, но не для приложений, разрабатываемых для конкретных информационных систем.

Использование запросов в виде текста может вызывать проблемы, если игнорируется принцип независимости данных и программ. В случае изменения схемы БД приложение об этом не узнает и будет посылать запросы, написанные для старой версии схемы. Если СУБД не поймет запрос, сервер вернет ошибку, а драйвер выбросит исключение. Драйверы поддерживают и вызов хранимых процедур, имена которых также задаются в текстовом виде.

Драйвер возвращает данные, полученные при выполнении запроса, в виде коллекции, состоящей из примитивных типов. Это ограничение вызвано тем,

что широко распространенные стандарты на интерфейс драйверов (например, ODBC и JDBC) основаны на устаревшем стандарте SQL 92. Преобразование полученного набора данных в соответствующие объекты должно проводиться в коде приложения. Если СУБД (например, PostgreSQL) допускает хранение и возврат сложных типов, таких как массивы, драйверы могут возвращать и такие типы данных, но это обеспечивается не каждым драйвером.

Существуют каркасы, которые вместо объектно-реляционных отображений устанавливают соответствие между объектами приложения и SQL-запросами. В этом случае SQL-запросы пишутся в коде приложения, а каркас преобразует полученные данные в объекты.

Драйверы баз данных имеет смысл использовать, когда приложение строится над уже имеющейся базой данных или если производительности SQL, который генерирует каркас объектно-реляционных отображений, недостаточно.

7.5.3. Интерфейс PostgreSQL для приложений

Возможности СУБД PostgreSQL, предоставляемые программам-клиентам, отражены в клиент-серверном протоколе. Наиболее полно эти возможности реализованы библиотекой libpq, которая может непосредственно использоваться в программах, написанных на языке C, а также является основой для интерфейсов многих других языков программирования, в том числе C++, Python, Perl, хотя имеются и альтернативные реализации этого протокола.

Чтобы начать работать с сервером, клиент должен установить соединение, указав необходимые параметры (в частности, имя пользователя и название базы данных) и пройдя аутентификацию.

Простой способ выполнения оператора состоит в передаче серверу текстовой строки с SQL-запросом без параметров. В ответ сервер возвращает клиенту результат выполнения полностью, сколько строк он бы ни содержал.

Расширенный протокол позволяет разбить выполнение команды на несколько этапов: подготовка (возможно, параметризованного) запроса, привязка фактических значений параметров, выполнение. Для операторов, возвращающих данные, можно получить информацию об именах и типах результирующих колонок результата. При выполнении оператора клиент может получать результат построчно (такая возможность обычно представляется в языках программирования понятием *курсора*).

Часто клиентские библиотеки неявно используют расширенный протокол, поскольку процедурная обработка множества объектов, включенных в результат, требует цикла по этим объектам.

Обычно обмен сообщениями между клиентом и сервером происходит синхронно: клиент посылает следующую команду, после того как обработает результаты предыдущей. Однако протокол PostgreSQL поддерживает и асинхронную отправку команд серверу, что позволяет клиенту обрабатывать результаты одной команды, в то время как сервер выполняет другой запрос. Кроме libpq, такая возможность поддерживается, например, драйвером для Erlang.

7.6. Некоторые общие задачи

В этом разделе мы рассмотрим некоторые типичные задачи, которые приходится решать почти в каждом проекте, независимо от его предметной области.

7.6.1. Ограничение доступа к данным

Требования к информационной системе должны содержать правила доступа к данным. Эти правила определяют для каждого конкретного пользователя, какие действия ему разрешено совершать. В настоящий момент наиболее распространенной является ролевая модель ограничения доступа, в которой роли назначаются пользователям, а разрешения назначаются ролям. При этом у пользователя может быть много ролей, а у роли много разрешений.

Что понимать под разрешением, зависит от конкретной задачи. Большинство задач требует детального контроля доступа к отдельным объектам. Современные базы данных, в том числе и PostgreSQL, предоставляют возможности определить правила доступа на уровне строк.

Для проектирования модели доступа к данным нужно решить, где будет осуществляться контроль доступа: на уровне СУБД или на уровне приложения. В случае если контроль доступа осуществляется на уровне СУБД, можно использовать встроенную поддержку безопасности. Однако такой подход приводит к некоторым осложнениям при проектировании приложения.

- Пользователи приложения должны быть также пользователями базы данных. Это может оказаться неприемлемым в системах, в которых предварительная регистрация не требуется (например, в интернет-магазинах),

или в тех случаях, когда количество пользователей слишком велико. Задача регистрации нового пользователя приложения включает в этом случае задачу создания пользователя СУБД, т. е. задачу администратора базы данных, а не приложения, а добавление строки в таблицу пользователей не рассматривается как задача администрирования.

- Исключается использование одного сеанса для обслуживания нескольких пользователей. В 90-е годы прошлого века было обнаружено, что подключение к СУБД является весьма дорогостоящей операцией. Поэтому на серверах приложений стали реализовывать *пул соединений*, повторно использующий заранее открытые соединения для обслуживания разных пользователей приложения.

  В результате усовершенствования СУБД и изменения мощностей компьютеров относительная стоимость создания нового соединения значительно снизилась, но пулы соединений по-прежнему применяются.

- Ограничения доступа задаются на уровне таблиц, строк и операций БД. Как правило, требования к модели доступа определяются в терминах предметной области, которая плохо отображается на базу данных. Например, право *заказать билет* нужно будет отобразить в операции вставки над соответствующими представлениями или таблицами.

- Как правило, ограничить доступ к базе данных недостаточно. Пользовательский интерфейс для пользователей с разными правами тоже выглядит по-разному.

Более распространенным является программирование ограничений доступа на уровне приложений. В этом случае все соединения к серверу могут происходить в контексте одного пользователя.

Такой подход также имеет некоторые недостатки:

- невозможно проводить аудит на уровне СУБД, т. к. неизвестно, какой пользователь приложения совершил операцию над данными;

- объектно-ориентированные языки не имеют встроенной поддержки ограничения доступа к объектам — придется использовать какую-либо библиотеку или реализовывать эту поддержку в коде приложения.

Возможны комбинированные решения, например роль приложения может соответствовать пользователю базы данных.

Какой бы подход ни был выбран, стоит пользоваться *принципом наименьших привилегий*: все пользователи должны иметь минимальный уровень доступа, необходимый для выполнения их задач.

7.6.2. Поддержка многоязычности

Информационные системы часто должны поддерживать несколько языков не только на уровне пользовательского интерфейса, но и на уровне данных. Это означает, что для некоторых атрибутов надо хранить несколько значений, по одному для каждого из поддерживаемых языков. В примере со студентами и дисциплинами мы можем хранить на нескольких языках названия курсов.

Существует несколько вариантов, как реализовать многоязычность на уровне базы данных.

Отдельный атрибут для каждого языка

В этом варианте для атрибута будет создано столько колонок, сколько мы поддерживаем языков. Этот подход показан на рис. 7.6.1.

courses

course_no	title_ru	title_en	credits
CS301	Базы данных	Databases	5
CS305	Анализ данных	Data analysis	10

Рис. 7.6.1. Атрибут для каждого языка

Недостатком такого варианта является необходимость модификации схемы базы данных и приложения при добавлении нового языка, а преимуществом — возможность непосредственного извлечения значения атрибута без обращения к каким-либо справочным таблицам.

Таблица с переводами для каждой сущности

Для каждой сущности, поддерживающей многоязычность, создается таблица с переводами, которая ссылается на главную таблицу. В примере, показанном на рис. 7.6.2, атрибут course_no является внешним ключом.

courses

course_no	credits
CS301	5
CS305	10

courses_translations

course_no	title	language
CS301	Базы данных	ru
CS301	Databases	en
CS305	Анализ данных	ru
CS305	Data Analysis	en

Рис. 7.6.2. Таблицы с переводами

В этом случае не требуется создавать новых структур данных при добавлении языков, а для получения данных достаточно одной операции соединения. Такой подход довольно сложен при использовании каркасов объектно-реляционных отображений, поскольку напрямую ими не поддерживается.

Общая таблица для хранения всех переводов

В этом варианте мы создаем одну общую таблицу для хранения переводов любых других таблиц. Поэтому в примере на рис. 7.6.3 таблица переводов названа просто translations, а связь осуществляется с помощью суррогатных идентификаторов.

courses

course_no	title_trans_id	credits
CS301	1	5
CS305	2	10

translations

trans_id	translation	language
1	Базы данных	ru
1	Databases	en
2	Анализ данных	ru
2	Data analysis	en

Рис. 7.6.3. Общая таблица переводов

Это наиболее гибкий подход. При его использовании также не нужно менять схему при добавлении новых языков, нет проблем с разреженными таблицами, и получается более правильное отображение в объекты.

Использование слабоструктурированных данных

Многие СУБД имеют специальные типы для поддержки слабоструктурированных данных. Так, PostgreSQL поддерживает форматы JSON и XML. Возможности этих средств в системе PostgreSQL рассматриваются далее в главе 8, а здесь мы ограничимся примером, показывающим, как можно применить такие средства для поддержки нескольких языков. Этот метод использован в демонстрационной базе данных PostgreSQL.

Информация об аэропортах хранится в таблице airports_data:

```
demo=# \d airports_data
                Table "bookings.airports_data"
    Column    |     Type     | Collation | Nullable | Default
--------------+--------------+-----------+----------+---------
 airport_code | character(3) |           | not null |
 airport_name | jsonb        |           | not null |
 city         | jsonb        |           | not null |
 coordinates  | point        |           | not null |
 timezone     | text         |           | not null |
Indexes:
    "airports_data_pkey" PRIMARY KEY, btree (airport_code)
...
```

Обратим внимание на то, что столбец airport_name имеет тип jsonb. Значения этого столбца содержат названия аэропорта на нескольких языках:

```
demo=# SELECT airport_name
FROM airports_data
LIMIT 10;
                          airport_name
-----------------------------------------------------------------
 {"en": "Yakutsk Airport", "ru": "Якутск"}
 {"en": "Mirny Airport", "ru": "Мирный"}
 {"en": "Khabarovsk-Novy Airport", "ru": "Хабаровск-Новый"}
 {"en": "Yelizovo Airport", "ru": "Елизово"}
 {"en": "Yuzhno-Sakhalinsk Airport", "ru": "Хомутово"}
 {"en": "Vladivostok International Airport", "ru": "Владивосток"}
 {"en": "Pulkovo Airport", "ru": "Пулково"}
 {"en": "Khrabrovo Airport", "ru": "Храброво"}
 {"en": "Kemerovo Airport", "ru": "Кемерово"}
 {"en": "Chelyabinsk Balandino Airport", "ru": "Челябинск"}
(10 rows)
```

Приложения, работающие с этой базой данных, должны использовать представление, которое выбирает название аэропорта на языке, установленном по умолчанию:

```
demo=# \d+ airports
...
View definition:
 SELECT ml.airport_code,
    ml.airport_name ->> lang() AS airport_name,
    ml.city ->> lang() AS city,
    ml.coordinates,
    ml.timezone
   FROM airports_data ml;
```

Функция lang вырабатывает код языка, установленный в конфигурационном параметре bookings.lang (устанавливается по умолчанию в значение ru, но может быть изменен). При выборке данных из этого представления получаем:

```
demo=# SELECT airport_name
FROM airports
LIMIT 10;
  airport_name
-----------------
 Якутск
 Мирный
 Хабаровск-Новый
 Елизово
 Хомутово
 Владивосток
 Пулково
 Храброво
 Кемерово
 Челябинск
(10 rows)
```

7.7. Настройка

Настройкой называется комплекс мер, направленных на приведение прикладной системы в соответствие с требованиями по производительности (в первую очередь по времени отклика и по пропускной способности), не изменяющий основных функций системы. Некоторые из таких мер часто называют «оптимизациями», однако в контексте курса по системам управления базами данных оптимизацией называется автоматический выбор плана выполнения запроса, который осуществляется оптимизатором.

В идеальном мире настройка рассматривается как важная составная часть всех этапов жизненного цикла прикладной системы, начиная с определения требований к системе, однако в реальности о настройке вспоминают только тогда, когда показатели производительности уже работающей системы оказываются неудовлетворительными.

Меры, которые можно считать элементами настройки прикладной системы, конечно, различны на разных фазах ее жизненного цикла. Очевидно, что точное определение требований по производительности создает условия для последующей настройки на всех этапах: выбор архитектуры и методологии разработки существенно влияет на все характеристики системы, определение как логической структуры данных, так и структуры хранения влияет на эффективность манипулирования данными, и т. д.

Меры, направленные на повышение производительности, могут применяться на различных уровнях.

Изменение (расширение) конфигурации оборудования обычно рассматривается как простой метод наращивания производительности, однако далеко не всегда масштабируемость позволяет решить проблемы производительности. При этом, как правило, удается повысить пропускную способность, но очень редко — улучшить время отклика. Неудачно написанный программный код может компенсировать любое увеличение мощности оборудования.

Выбор параметров сервера включает конфигурацию как операционной системы, так и сервера базы данных и является одной из основных задач администратора баз данных. Параметры конфигурации позволяют изменять размеры областей оперативной памяти, используемых сервером базы данных, количество процессов, выполняющих запросы пользователей, стратегии копирования и восстановления в случае отказов и т. п.

Управление схемой базы данных включает прежде всего управление хранением данных: размещение табличных пространств на устройствах; размещение таблиц, индексов и других объектов в табличных пространствах; выбор параметров, управляющих размещением данных в таблицах, и т. д. Сюда же включаются выбор материализованных представлений и другие модификации логической схемы базы данных.

Улучшение запросов. Как правило, оптимизатор базы данных вполне успешно выполняет свои функции и создает планы выполнения запросов, близкие

к оптимальным. Однако в некоторых случаях запрос может содержать избыточные операции, или по каким-либо другим причинам план, построенный оптимизатором, оказывается неудовлетворительным. В подобных ситуациях требуется ручная настройка запроса, обычно состоящая в переписывании его в другую эквивалентную форму, и зачастую могут потребоваться изменения в структуре хранения данных (например, создание дополнительных индексов или материализованных представлений).

Реструктуризация кода приложения обычно рассматривается как дорогостоящее и потенциально опасное действие, однако многие виды неэффективности приложения вызываются именно дефектами кода приложения и не могут быть полностью компенсированы на других уровнях. В частности, одной из наиболее частых причин плохого времени отклика является программный код, выполняющий слишком много слишком мелких запросов.

Наиболее результативной настройка будет только в случае ее проведения на всех этапах разработки и на всех уровнях от оборудования до программного кода приложения, с учетом как особенностей применяемых систем, так и функций прикладной системы и требований к ней.

7.8. Проектирование декларативных запросов

Операторы SQL, автоматически генерируемые средствами объектно-реляционных отображений, зачастую проигрывают по качеству запросам, спроектированным вручную. Поэтому при реализации прикладных систем, в которых необходима массовая обработка большого количества данных с высокими требованиями к производительности, целесообразно отделение функций манипулирования данными от остальных функций приложения. При этом доступ средств объектно-реляционных отображений к данным осуществляется через интерфейс виртуальных отношений, которые могут быть как хранимыми таблицами, так и реализованы другими способами, в том числе параметризованными запросами, представлениями и процедурами базы данных.

Реализованная в PostgreSQL полноценная поддержка отношений в качестве возвращаемых функциями значений является мощным инструментом. Преимуществами пользовательских функций, возвращающих отношения, по сравнению с представлениями, являются:

- возможность передачи параметров, задающих условия на формирование результата;

- возможность формирования динамического кода SQL в зависимости от переданных параметров.

Необходимо подчеркнуть, что разработка функций базы данных требует относительно высокой квалификации, в том числе необходимо глубокое понимание того, каким образом оптимизируются и выполняются запросы в СУБД. Вся существенная работа с данными должна описываться на декларативном языке (т. е. на SQL), хотя возможности языков, на которых реализуются функции базы данных, могут провоцировать интенсивное использование имеющихся в них императивных средств.

Проектирование и разработка декларативных запросов существенно отличаются от разработки программного кода на императивном языке. Декларативное программирование предполагает, как и следует из названия, задание условий и требований, которым должен удовлетворять результат вычислений, но не конкретный алгоритм вычисления. Однако этого недостаточно. Чтобы возможности СУБД использовались наилучшим образом, необходимо учитывать особенности модели данных: ориентацию на ассоциативный доступ и массовую обработку. Неформально можно сказать, что для получения высококачественного декларативного описания необходимо «думать в терминах множеств». Это выражение вынесено в заголовок книги [5], в которой детально обсуждаются методы конструирования запросов.

Хорошими руководствами по проектированию запросов и применению SQL являются также книги [12] и [4] (русский перевод [21]).

7.9. Итоги главы

В этой главе кратко характеризуются различные подходы, методологии и инструменты разработки приложений, а также различные варианты определения отображений между объектными моделями приложений и объектно-реляционными моделями баз данных. Кратко представлена методика разработки сложных запросов и сформулирована задача настройки приложений баз данных.

7.10. Упражнения

В демонстрационной базе поддержка нескольких языков реализована с использованием средств JSON. В следующих упражнениях требуется реализовать такую поддержку с помощью других вариантов, рассмотренных в этой главе.

Упражнение 7.1. Создайте новую схему демонстрационной базы таким образом, чтобы названия населенных пунктов хранились в соответствии с выбранным вами способом организации многоязычности. Данные должны быть представлены как минимум на русском и английском языках. Предусмотрите возможность расширения списка используемых языков.

Упражнение 7.2. Мигрируйте все данные из старой схемы демонстрационной базы в новую из предыдущего упражнения. Добавьте названия населенных пунктов на каких-либо других языках.

Упражнение 7.3. Напишите запрос, для каждого города показывающий количество пассажиров, прилетающих в него из Москвы в какой-нибудь определенный день. Запрос должен выдавать названия населенных пунктов на английском языке.

Упражнение 7.4. Реализуйте функцию покупки билета. С точки зрения пассажира покупка состоит из двух этапов. На первом система выдает список возможных перелетов для указанных параметров: пункты отправления и назначения, количество мест, дата вылета. На втором — выбранный пассажиром вариант оформляется в виде бронирования, и пассажиру возвращается номер бронирования.

Попробуйте различные способы: с помощью запросов SQL, с применением хранимых функций, на основе использования каркаса.

Глава 8

Расширения реляционной модели

В этой главе обсуждаются расширения традиционных технологий применения систем, основанных на модели данных SQL, вплоть до попыток полного отказа от функциональности языка запросов. Зачастую при обсуждении подобных расширений принято отождествлять реляционную модель данных с комплексом технологий, основанных на подмножестве SQL, однако мы будем различать эти понятия.

8.1. Ограниченность реализаций SQL

Напомним, что ранние реализации реляционной модели данных и вместе с ними ранние версии языка SQL предусматривали только простые скалярные типы данных в качестве значений атрибутов таблиц: числа, символьные строки, даты и время и т. п. Как только началось относительно широкое применение подобных систем (во второй половине 80-х годов), стало ясно, что модели и структуры данных, предоставляемые подобными системами, слишком ограничительны для целого ряда классов приложений.

В частности, для реализации систем автоматизированного проектирования (САПР, CAD) необходимо организовать хранение нескольких версий проекта. За время существования проекта может быть всего несколько десятков версий, однако каждая из них представляет собой сложный объект, объем которого может достигать значительных размеров. Для поддержки таких систем было введено понятие *вложенных отношений* или *отношений не в первой нормальной форме* (nested relations, non-first normal form, NFNF, NF^2). В таких отношениях значения атрибутов могут быть таблицами или отношениями (возможно, тоже содержащими вложенные таблицы).

Вложенные отношения были достаточно детально изучены теоретиками, однако до массового применения этой модели дело не дошло. Заметим, что структура данных вложенного отношения может быть эквивалентно отображена на

обычные (плоские) отношения, поэтому речь идет скорее о логической структуре, приближенной к структуре хранения данных, чем о действительно новой модели данных. Напомним, что теоретическая реляционная модель не занимается вопросами организации хранения данных.

Несколько более успешной оказалась идея объектно-ориентированных СУБД, появившихся в период бурного развития объектно-ориентированных языков программирования и методологий объектно-ориентированного программирования.

Модель данных объектно-ориентированных СУБД строится как совокупность классов или типов, описывающих как состояние (атрибуты), так и поведение (методы) объектов. Для описания взаимосвязей между объектами или их множествами используются явные или неявные указатели или ссылки, что, по существу, означает преимущественное применение навигации для поиска объектов в базе данных.

Ожидалось, что объектно-ориентированные СУБД вытеснят все остальные модели баз данных, однако фактически объектно-ориентированные системы смогли занять весьма скромную долю рынка СУБД и применялись для очень ограниченного круга прикладных областей, в которых нужны базы данных относительно небольшого объема, но требующие значительных вычислительных мощностей для обработки. При таких условиях размещение кода методов в базе данных оказывается вполне оправданным, т. к. это сокращает до минимума накладные расходы на передачу данных между местом хранения и местом, где фактически выполняются вычисления.

Применение объектно-ориентированных СУБД в других случаях оказалось неоправданным. В качестве причин, которые привели к такому результату, можно обозначить следующие:

- навигационный поиск весьма эффективен для доступа к отдельным объектам, однако оказывается неэффективным при массовой обработке;
- для объектно-ориентированных моделей не удается построить полноценный высокоуровневый декларативный язык запросов;
- модель данных оказывается привязанной к одному языку программирования (на котором записаны методы), что затрудняет использование базы данных другими приложениями. По существу, приходится отказаться от принципа независимости данных и программ.

Намного более успешным оказался альтернативный подход, получивший название объектно-реляционных СУБД. В настоящее время все высокопроизводительные системы, в том числе PostgreSQL, фактически относятся к этому классу систем. Центральным понятием в таких системах остается понятие таблицы, однако в качестве типов атрибутов могут использоваться объектные типы данных, как встроенные в систему, так и определенные в конкретной схеме базы данных.

В рамках такой системы можно использовать и сочетать самые различные способы организации данных и применять как высокоуровневый язык запросов, так и методы для обработки отдельных объектов данных.

Например, если схема базы данных содержит таблицу с единственным атрибутом и содержащую единственную строку, но тип этого атрибута является сложным объектным типом, по существу, такая база данных становится объектно-ориентированной, хотя некоторые возможности применения высокоуровневых запросов при этом сохраняются.

Таблица, в которой типы некоторых атрибутов представляют собой коллекции, может использоваться для хранения отношений не в первой нормальной форме (NFNF), упомянутых выше.

Язык запросов объектно-реляционных систем пополняется средствами для доступа к составляющим сложных объектов, которые могут быть значениями атрибутов отношений.

Несмотря на использование близкой терминологии, объектные средства в системах управления базами данных зачастую существенно отличаются от аналогичных средств в языках программирования.

Заметим, что хотя объектно-реляционные СУБД, безусловно, являются расширениями стандарта SQL, получившего очень широкое распространение, однако никак не рассогласуются с теоретической реляционной моделью данных, в которой в качестве значений атрибутов могут использоваться значения из произвольных доменов, которые реализуются объектными типами данных.

Напомним, что в теоретической реляционной модели домены могут определяться абстрактным типом данных, единственным требованием к которому является наличие предиката равенства значений из этого домена. Остальные операции, функции и предикаты на этом домене могут отражать особенности предметной области или применения. Так, можно различать домены весов, длин, денежных сумм, геометрических объектов определенного вида (точек, прямых, прямоугольников) и т. д.

Для того чтобы рассматривать сложные (нескалярные) значения атрибутов в теоретической реляционной модели, нужно ввести функции, преобразующие значение атрибута в структурный тип или коллекцию. Этот прием используется в объектно-реляционных СУБД для реализации, например, таких типов, как XML и JSON, рассматриваемых далее в данной главе.

В следующих разделах этой главы показано, каким образом объектные средства представлены в PostgreSQL. Некоторые из этих средств рассматриваются как отдельные расширения, однако концептуально это объектные типы с богатой семантикой и специфическими операциями.

Объектные возможности PostgreSQL в основном сводятся к определению новых структур данных (но не поведения, которое должно описываться функциями, определяемыми отдельно от описаний структур данных).

8.2. Реализация объектных расширений в PostgreSQL

Система PostgreSQL не предоставляет объектные расширения в виде какой-либо полной объектной модели данных. Вместо этого в системе имеются возможности для создания высокоуровневых объектных средств на основе имеющихся конструкций. В этом разделе рассматриваются главным образом возможности определения сложных структур данных и типы данных, необходимые для представления объектов и связей между ними. Поведение объектов может описываться с помощью функций, которые могут быть написаны на SQL или на любом из поддерживаемых в PostgreSQL императивных языков программирования.

Можно сказать, что наиболее важной характеристикой PostgreSQL является расширяемость, т. е. возможность добавления новых типов данных, функций, операторов и даже индексных структур без изменения ядра системы PostgreSQL.

8.2.1. Наследование

Применение наследования в объектно-ориентированных языках программирования уже обсуждалось в главе 7. Здесь мы кратко охарактеризуем возможности представления наследования на уровне базы данных в системе PostgreSQL.

В PostgreSQL наследование можно использовать для определения таблиц. Это отличает PostgreSQL от стандарта SQL, в котором начиная с издания 1999 года предусмотрено наследование для типов. Если при определении таблицы указано, что она наследует из другой (родительской) таблицы, то в состав атрибутов определяемой таблицы включаются все атрибуты родительской таблицы. При этом в запросах можно указывать, требуется ли выбрать строки только из указанной таблицы или из указанной и всех ее дочерних таблиц. При вставке новых строк (INSERT) операция выполняется на указанной таблице независимо от того, участвует эта таблица в наследовании или нет.

Ограничения ссылочной целостности (PRIMARY KEY, FOREIGN KEY) и ограничения на уникальность значений (UNIQUE) применяются только к каждой из таблиц по отдельности. Из этого следует, что в иерархии наследования могут появляться, например, дублирующиеся значения, даже если заданы ограничения UNIQUE на каждой из таблиц иерархии. Эти особенности заметно снижают полезность наследования; документация рекомендует использовать его с осторожностью.

Можно заметить, что наследование таблиц в PostgreSQL предполагает горизонтальную фрагментацию, т. е. каждая таблица содержит лишь строки, не принадлежащие наследующим из нее таблицам, и содержит все атрибуты, логически входящие в эту таблицу. В противоположность этому теоретическая модель «сущность-связь» неявно предполагает вертикальную фрагментацию.

8.2.2. Определение типов данных

В системе PostgreSQL можно определять несколько разновидностей пользовательских типов данных.

Составной тип данных (запись, record) представляет собой структуру, состоящую из нескольких атрибутов (подобно определению строки таблицы). На самом деле каждая таблица определяет и составной тип, атрибуты которого соответствуют описанию этой таблицы.

Тип диапазона (range) задает интервал значений некоторого другого типа, для которого должно быть определено отношение полного упорядочивания. Встроенными диапазонными типами являются диапазоны для различных числовых типов (int4range, int8range, numrange) и интервалов времени (tsrange, tstzrange) и дат (daterange).

Перечисляемый тип (enum) соответствуют перечислительным типам, которые могут быть определены во многих языках программирования. Такие типы могут принимать фиксированное в определении типа количество различных именованных значений.

Новые базовые типы могут определяться с помощью указания как внутреннего представления (через другие типы данных), так и функций, реализующих операции над этим типом и обеспечивающих возможность его эффективного использования (например, указания для оптимизатора запросов). Определение базовых типов требует некоторых знаний о внутреннем устройстве СУБД и достаточно трудоемко, поэтому вряд ли целесообразно создавать базовые типы при разработке отдельных приложений.

8.2.3. Домены

Понятие *домена* в SQL отличается от понятия домена теоретической реляционной модели данных: это не абстрактный скалярный тип данных, а ранее определенный тип данных с дополнительными ограничениями на его значения.

8.2.4. Коллекции

Для представления вложенных коллекций в PostgreSQL можно использовать *массивы* (array), во многом похожие на массивы в языках программирования. Массивы состоят из элементов одного типа, которые идентифицируются целочисленными индексами, задающими их положение. В PostgreSQL массивы могут быть и многомерными.

Кроме обычных операций вырезки (выделения подмассива меньшего размера или отдельных элементов), определены операции, позволяющие склеивать несколько массивов в один (операция конкатенации и несколько аналогичных по назначению функций). Операции конкатенации особенно полезны для одномерных массивов, т. к. с их помощью легко реализовать списки.

Значения массивов можно записывать как константы, добавлять или изменять элементы массивов, а также формировать массивы из значений, извлекаемых из таблиц:

```
demo=# SELECT array_agg(airport_code)
FROM (SELECT * FROM airports LIMIT 8) air;
             array_agg
-----------------------------------
 {AAQ,ABA,AER,ARH,ASF,BAX,BQS,BTK}
(1 row)
```

В этом запросе функция array_agg собирает в массив значения, находящиеся в разных строках таблицы, при этом количество элементов в массиве будет равно числу строк в отношении, полученном в результате выполнения подзапроса. Табличное выражение (подзапрос) в предложении FROM нужно, конечно, только для того, чтобы размер результата был обозримым, никакого содержательного смысла ограничение числа строк в данном случае не имеет. Аргументом функции array_agg может быть значение любого типа, а не только скалярное, как в нашем примере. Так, можно построить массив из записей или многомерный массив.

Обратное преобразование массива во множество (таблицу) можно выполнить с помощью функции unnest:

```
demo=# SELECT *
FROM unnest(ARRAY['AAQ','ABA','AER','ARH','ASF','BAX','BQS','BTK']);
 unnest
--------
 AAQ
 ABA
 AER
 ARH
 ASF
 BAX
 BQS
 BTK
(8 rows)
```

Здесь в качестве источника данных использован массив, сконструированный из констант, но, конечно, массив мог бы быть и значением некоторой колонки таблицы.

Необходимо отметить, что массивы не могут рассматриваться как полноценная замена таблиц, потому что операции выборки данных из таблиц и некоторые операции реляционной алгебры реализуются более эффективно. Некоторые свойства данных, например ограничения целостности, нельзя определить для элементов массивов. Тем не менее массивы в PostgreSQL обеспечивают много полезных возможностей, в том числе реализацию вложенных отношений, возможности специализированных СУБД для обработки массивов (научных данных).

8.2.5. Указатели

Одной из важных особенностей объектных моделей данных является навигационный доступ, который, как правило, поддерживается с помощью объектных указателей. Обычно предполагается, что значения указателей не могут изменяться и для них применяются целочисленные суррогаты, хотя в принципе в качестве указателей можно использовать любые типы данных. Связь между строками таблиц по первичному и внешнему ключам можно считать частным случаем применения объектных указателей. В других случаях значения объектных указателей могут быть собраны в массив.

Можно сказать, что объектный указатель — это не особый тип данных, а особый способ использования значений, хотя в объектных языках программирования указатели выделяются как отдельный тип.

Среди встроенных типов данных в PostgreSQL имеется несколько типов, которые могут использоваться как объектные указатели на различных уровнях. Один из этих типов — oid (object identifier) — используется в этом качестве самой СУБД, однако из-за его малого размера (4 байта) в прикладных системах рекомендуется применять другие типы, например длинные целые числа.

8.3. Функции

Наиболее мощные объектные возможности PostgreSQL реализуются при помощи аппарата функций, хранимых и выполняемых на сервере базы данных. Такие функции могут быть написаны на различных языках программирования, в том числе на SQL, однако наиболее развитые возможности манипулирования данными, хранимыми в БД, предоставляются в языке PL/pgSQL.

В системе PostgreSQL функции могут принимать аргументы и вырабатывать результат любых типов, как встроенных, так и пользовательских, включая скалярные типы, записи, массивы и множества (таблицы).

Как и любой мощный инструмент, процедурные языки могут быть опасными. Непродуманное использование процедурных языков может привести к серьезной потере вычислительной эффективности. Это связано с тем, что оптимизатор запросов, как правило, не имеет информации о стоимости выполнения функции и поэтому не может правильно оценить затраты ресурсов, необходимые для ее выполнения. Другая возможная причина снижения эффективности

при необдуманном применении функций связана с тем, что запросы, содержащиеся в теле функции, оптимизируются и выполняются отдельно. Это может привести к эффекту «слишком много слишком мелких запросов».

Если функция написана на языке SQL, обработчик запросов PostgreSQL может в некоторых случаях подставлять тело такой функции в запрос, содержащий ее вызов, и тогда потеря эффективности не происходит.

8.4. Слабоструктурированные данные: JSON

Ограниченность систем типов данных, реализованных в ранних СУБД, использующих SQL, привела к тому, что развитые на их основе методологии проектирования и технологии применения СУБД оказались слишком ограничительными для многих классов приложений. Зачастую эту ограниченность связывают с самой реляционной моделью данных, поэтому средства для преодоления этих ограничений называют расширениями реляционной модели, хотя фактически они только снимают ограничения технологий, построенных на ее основе.

Одним из таких ограничений является раздельное хранение схемы базы данных и самих данных. Такое разделение оказывается неудобным для приложений, использующих внешние источники данных (например, получаемые из информационных ресурсов, доступных в интернете). В подобных случаях схема может быть опубликована не полностью, и поэтому фактическая структура данных может отличаться от описанной в известной части схемы. Важной особенностью таких данных является также присутствие неструктурированных данных (например, текстов на естественном языке, изображений и т. п.). Подобные данные принято называть *слабоструктурированными* (semi-structured). Часто по-русски используется также термин *полуструктурированные*.

Для передачи таких данных по вычислительным сетям используются форматы XML и JSON, в которых имена атрибутов указываются вместе со значениями этих атрибутов. Это означает, что схема таких данных является динамической. Отметим, что при использовании таких ничем не ограниченных схем разработка приложения существенно усложняется, т. к. функции по управлению схемой, обычно выполняемые в СУБД, перекладываются на приложение. Кроме этого, приложение должно быть готово к обработке различных исключительных ситуаций, которые не могли бы возникать при наличии ограничений, обрабатываемых СУБД. Наиболее очевидными исключительными ситуациями

могут быть отсутствие элементов данных (атрибутов), необходимых для работы приложения, дублирование атрибутов, которые приложение считает уникальными, различие в именовании атрибутов и т. п.

Тем не менее обработка данных в таких форматах необходима, и для того чтобы ее обеспечить, в составе PostgreSQL имеются тип xml для хранения данных в формате XML и два типа (json и jsonb) для хранения в формате JSON, отличающихся способом внутреннего представления в базе данных.

Как и все остальные типы, эти типы можно задавать для отдельных атрибутов таблиц. Если в таблице имеется всего один атрибут, например типа json, и всего одна строка, то по существу такая таблица хранит один документ и работа с этим документом происходит так, как в базах данных, ориентированных исключительно на хранение документов; при этом возможности SQL почти не используются. Если же таблица содержит и другие атрибуты (возможно, некоторые из них также слабоструктурированного типа) или содержит несколько строк, то появляется возможность сочетать преимущества различающихся моделей данных при работе с одной базой данных.

В определении слабоструктурированных типов данных предусмотрены возможности конструирования значений этих типов из других, структурированных атрибутов объектов, хранящихся в базе данных, а также преобразования слабоструктурированных значений в структурированные с выделением значений отдельных атрибутов.

Следующий оператор вырабатывает результат, содержащий одну колонку типа json, при этом в каждой строке результата находятся данные только из одной строки исходной таблицы airports:

```
demo=# SELECT json_build_object(
    'code', airport_code,
    'name', airport_name)
FROM airports
LIMIT 8;
               json_build_object
-----------------------------------------------
 {"code" : "YKS", "name" : "Якутск"}
 {"code" : "MJZ", "name" : "Мирный"}
 {"code" : "KHV", "name" : "Хабаровск-Новый"}
 {"code" : "PKC", "name" : "Елизово"}
 {"code" : "UUS", "name" : "Хомутово"}
 {"code" : "VVO", "name" : "Владивосток"}
 {"code" : "LED", "name" : "Пулково"}
 {"code" : "KGD", "name" : "Храброво"}
(8 rows)
```

Вариант этого запроса собирает те же данные в один JSON-документ, представляющий собой массив (в смысле JSON) документов, полученных из отдельных строк таблицы с помощью функции агрегирования json_agg:

```
demo=# SELECT json_agg(
  json_build_object('code', airport_code, 'name', airport_name)
)
FROM (
  SELECT * FROM airports LIMIT 8
) air;
                                 json_agg
-------------------------------------------------------------------------
 [{"code" : "YKS", "name" : "Якутск"}, {"code" : "MJZ", "name" :
 "Мирный"}, {"code" : "KHV", "name" : "Хабаровск-Новый"}, {"code" :
 "PKC", "name" : "Елизово"}, {"code" : "UUS", "name" : "Хомутово"},
 {"code" : "VVO", "name" : "Владивосток"}, {"code" : "LED", "name" :
 "Пулково"}, {"code" : "KGD", "name" : "Храброво"}]
(1 row)
```

В следующем примере объект JSON преобразуется в таблицу пар ключ-значение.

```
demo=# SELECT *
FROM json_each('{"code" : "KVK", "name" : "Апатиты-Кировск"}');
 key  |       value
------+-------------------
 code | "KVK"
 name | "Апатиты-Кировск"
(2 rows)
```

Еще один пример показывает, каким образом объект JSON можно преобразовать в структуру, соответствующую строке таблицы:

```
demo=# SELECT *
FROM json_populate_record(
    NULL::airports,
    '{"airport_code" : "KVK",
      "name" : "Апатиты-Кировск",
      "city" : "Кировск"}'
);
 airport_code | airport_name |  city   | coordinates | timezone
--------------+--------------+---------+-------------+----------
 KVK          |              | Кировск |             |
(1 row)
```

При этом преобразовании:

- значения атрибутов, ключи которых совпадают с именами колонок таблицы, включаются в результат (airport_code и city);

- атрибуты, для ключей которых не нашлось колонки, выбрасываются (name);
- колонки, для которых не нашлось атрибутов объекта JSON, заполняются неопределенными значениями (airport_name, coordinates, timezone).

Если документ в формате JSON представляет собой массив объектов, можно преобразовать этот документ в таблицу, содержащую каждый элемент в отдельной строке. В следующем примере массив JSON записан в явном виде, но он может передаваться из приложений или храниться в качестве значений в столбцах таблиц типа json.

```
demo=# SELECT *
FROM json_populate_recordset(
  NULL::airports,
  '[{"airport_code" : "YKS", "airport_name" : "Якутск"},
    {"airport_code" : "MJZ", "airport_name" : "Мирный"},
    {"airport_code" : "KHV", "airport_name" : "Хабаровск-Новый"},
    {"airport_code" : "PKC", "airport_name" : "Елизово"},
    {"airport_code" : "UUS", "airport_name" : "Хомутово"},
    {"airport_code" : "VVO", "airport_name" : "Владивосток"},
    {"airport_code" : "LED", "airport_name" : "Пулково"},
    {"airport_code" : "KGD", "airport_name" : "Храброво"}]'::json
);
 airport_code |  airport_name   | city | coordinates | timezone
--------------+-----------------+------+-------------+----------
 YKS          | Якутск          |      |             |
 MJZ          | Мирный          |      |             |
 KHV          | Хабаровск-Новый |      |             |
 PKC          | Елизово         |      |             |
 UUS          | Хомутово        |      |             |
 VVO          | Владивосток     |      |             |
 LED          | Пулково         |      |             |
 KGD          | Храброво        |      |             |
(8 rows)
```

Структуру получаемого множества можно также задавать в самом запросе, а не ссылаться на заранее созданный тип записи:

```
demo=# SELECT *
FROM json_to_recordset(
  '[{"airport_code" : "YKS", "airport_name" : "Якутск"},
    {"airport_code" : "MJZ", "airport_name" : "Мирный"},
    {"airport_code" : "KHV", "airport_name" : "Хабаровск-Новый"},
    {"airport_code" : "PKC", "airport_name" : "Елизово"},
    {"airport_code" : "UUS", "airport_name" : "Хомутово"},
    {"airport_code" : "VVO", "airport_name" : "Владивосток"},
    {"airport_code" : "LED", "airport_name" : "Пулково"},
    {"airport_code" : "KGD", "airport_name" : "Храброво"}]'::json
) AS (airport_code text, airport_name text);
```

```
 airport_code |  airport_name
--------------+-----------------
 YKS          | Якутск
 MJZ          | Мирный
 KHV          | Хабаровск-Новый
 PKC          | Елизово
 UUS          | Хомутово
 VVO          | Владивосток
 LED          | Пулково
 KGD          | Храброво
(8 rows)
```

8.5. Слабоструктурированные данные: XML

Преобразование данных из таблиц в формат XML осуществляется с помощью функций, определенных стандартами XML и реализованными в PostgreSQL, а также рядом дополнительных функций, упрощающих такое преобразование.

Наиболее важными из стандартных функций для генерации XML являются:

- xmlelement — строит один элемент XML с заданным именем, значением и, возможно, атрибутами;
- xmlforest — строит список из нескольких элементов XML;
- xmlagg — агрегатная функция, объединяющая в один документ элементы, полученные из нескольких строк таблицы.

Например, рассмотрим следующий запрос:

```
demo=# SELECT f.flight_no,
  dep.airport_code dep,
  arr.airport_code arr
FROM ticket_flights tf
  JOIN flights f ON tf.flight_id = f.flight_id
  JOIN airports dep ON f.departure_airport = dep.airport_code
  JOIN airports arr ON f.arrival_airport = arr.airport_code
WHERE tf.ticket_no = '0005432369015'
ORDER BY f.scheduled_departure;
 flight_no | dep | arr
-----------+-----+-----
 PG0233    | VKO | BZK
 PG0649    | BZK | EGO
 PG0481    | EGO | AAQ
 PG0480    | AAQ | EGO
 PG0650    | EGO | BZK
 PG0237    | BZK | VKO
(6 rows)
```

Запрос возвращает все перелеты для одного из билетов в демонстрационной базе данных. Та же информация в формате XML может быть получена как таблица, содержащая единственную колонку типа xml:

```
demo=# SELECT xmlelement(
    name flight,
    xmlforest(
      dep.airport_code AS dep,
      arr.airport_code AS arr
    )
  )
FROM ticket_flights tf
  JOIN flights f ON tf.flight_id = f.flight_id
  JOIN airports dep ON f.departure_airport = dep.airport_code
  JOIN airports arr ON f.arrival_airport = arr.airport_code
WHERE tf.ticket_no = '0005432369015'
ORDER BY f.scheduled_departure;
                  xmlelement
-----------------------------------------------
 <flight><dep>VKO</dep><arr>BZK</arr></flight>
 <flight><dep>BZK</dep><arr>EGO</arr></flight>
 <flight><dep>EGO</dep><arr>AAQ</arr></flight>
 <flight><dep>AAQ</dep><arr>EGO</arr></flight>
 <flight><dep>EGO</dep><arr>BZK</arr></flight>
 <flight><dep>BZK</dep><arr>VKO</arr></flight>
(6 rows)
```

Для того чтобы вывести ту же информацию в виде одного документа, применим функцию агрегирования:

```
demo=# SELECT xmlagg(xmlelement(
    name flight,
    xmlforest(
      dep.airport_code AS dep,
      arr.airport_code AS arr
    )
  ) ORDER BY f.scheduled_departure)
FROM ticket_flights tf
  JOIN flights f ON tf.flight_id = f.flight_id
  JOIN airports dep ON f.departure_airport = dep.airport_code
  JOIN airports arr ON f.arrival_airport = arr.airport_code
WHERE tf.ticket_no = '0005432369015';
                                 xmlagg
-----------------------------------------------------------------------
 <flight><dep>VKO</dep><arr>BZK</arr></flight><flight><dep>BZK</dep>
 <arr>EGO</arr></flight><flight><dep>EGO</dep><arr>AAQ</arr></flight
 ><flight><dep>AAQ</dep><arr>EGO</arr></flight><flight><dep>EGO</dep
 ><arr>BZK</arr></flight><flight><dep>BZK</dep><arr>VKO</arr></fligh
 t>
(1 row)
```

Предложение ORDER BY в этом случае должно быть размещено в качестве параметра агрегирующей функции.

В системе PostgreSQL имеются также высокоуровневые функции, преобразующие в формат XML содержимое таблицы, результат выполнения запроса, содержимое курсора или схемы. В этом случае имена элементов выбираются на основе соответствующих имен в реляционной схеме, а преобразование типов выполняется в соответствии с правилами для неявных преобразований. Например, функцию query_to_xml можно использовать для преобразования результата выполнения запроса в XML:

```
demo=# SELECT query_to_xml(
  'SELECT aircraft_code, range FROM aircrafts WHERE range < 3000',
  true, false, ' '
);
                          query_to_xml
---------------------------------------------------------------
 <table xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns=" ">                                                   +
                                                              +
 <row>                                                        +
   <aircraft_code>CN1</aircraft_code>                         +
   <range>1200</range>                                        +
 </row>                                                       +
                                                              +
 <row>                                                        +
   <aircraft_code>CR2</aircraft_code>                         +
   <range>2700</range>                                        +
 </row>                                                       +
                                                              +
 </table>                                                     +

(1 row)
```

Преобразование XML в табличный формат лучше всего выполнять с помощью функции xmltable, предусмотренной в стандарте SQL/XML и включенной в основной комплект PostgreSQL начиная с версии 10. Функция xmltable описывает, как извлекать данные в форме таблицы из документа в формате XML:

```
xmltable( путь-к-таблице
  PASSING документ-источник
  COLUMNS (список-столбцов)
)
```

где

- *путь-к-таблице* — XPath-запрос, задающий путь к данным таблицы в исходном документе;

- *документ-источник* — выражение SQL, содержащее исходный документ;
- *список-столбцов* — описание форматов и методов вычисления значений в результирующей таблице.

Описание каждой колонки включает ее имя, тип SQL (в который следует преобразовать данные) и выражение XPath, которое указывает путь от корня таблицы (заданного первым параметром функции xmltable) к значению (элементу исходного документа) и которое преобразуется в значение описываемой колонки.

Функция xmltable вырабатывает значение типа отношение (таблица), которое можно использовать в обычных операторах SQL.

В следующем примере функция xmltable использована для формирования таблицы из XML-документа, который записан как константа в том же запросе. В реальных применениях, конечно, вместо констант используются значения переменных. Например, это может быть значение из столбца таблицы или входное сообщение, полученное из другой системы.

```
demo=# SELECT *
FROM xmltable( '//flights/flight'
  PASSING (
    SELECT xmlconcat(
      '<ticket>
       <number>0005432369015</number>
       <flights>
       <flight><from>VKO</from><to>BZK</to></flight>
       <flight><from>BZK</from><to>EGO</to></flight>
       <flight><from>EGO</from><to>AAQ</to></flight>
       <flight><from>AAQ</from><to>EGO</to></flight>
       <flight><from>EGO</from><to>BZK</to></flight>
       <flight><from>BZK</from><to>VKO</to></flight>
       </flights>
       </ticket>'
    )
  )
  COLUMNS
    departs_from text PATH 'from',
    arrives_to   text PATH 'to'
);
 departs_from | arrives_to
--------------+------------
 VKO          | BZK
 BZK          | EGO
 EGO          | AAQ
 AAQ          | EGO
 EGO          | BZK
 BZK          | VKO
(6 rows)
```

Существуют также функции более низкого уровня, извлекающие значения отдельных элементов из XML-документов на основе выражений XPath или XQuery.

8.6. Активные базы данных

Обычно системы управления базами данных выполняют действия только по запросам приложения. Рассмотрим СУБД, которые выполняют не только действия, явно указанные приложением, но также реагируют на события, возникающие в самой БД. Такие системы называются *активными базами данных*. Например, счетчик комментариев на сайте может быть рассчитываемой величиной, а может храниться в отдельной таблице и обновляться при добавлении нового комментария. В этом случае логика обновления счетчика реализована в БД как функция, которая вызывается при вставке новой записи в таблицу комментариев.

Формально поведение таких систем определяется в терминах предписаний, содержащих три компонента:

1) событие;
2) дополнительные условия;
3) описания действий, которые должны выполняться при указанном событии, если условия оказались истинными.

В научной литературе такие предписания называются *ECA-правилами* (event-condition-action). Мы используем термин *предписания*, для того чтобы избежать путаницы с *правилами*, которые применяются в PostgreSQL для совсем других целей.

На практике, в том числе в PostgreSQL, активность базы данных реализуется с помощью аппарата *триггеров*. Триггером называют функцию, обычно написанную на процедурном языке, которая вызывается системой автоматически при срабатывании связанных с ней предписаний. В спецификации триггера могут быть определены дополнительные условия.

В системе PostgreSQL различают триггеры, срабатывающие при модификации данных (INSERT, UPDATE, DELETE и TRUNCATE), и триггеры событий, срабатывающие при выполнении операторов языка описания данных, к числу которых относятся ALTER, CREATE, DROP, GRANT, REVOKE.

Процедурный код, реализованный в триггере, может существенно дополнить или изменить семантику стандартных операторов SQL. Например, триггеры можно использовать для проверки условий целостности, которые невозможно описать стандартными средствами языка SQL, или для регистрации изменений, выполняемых приложением, в другой таблице.

Важно подчеркнуть, что действия, предусмотренные в триггере, будут выполняться всегда, когда возникает специфицированная ситуация, и выполняются в рамках той же транзакции. Приложение не имеет никакой возможности обойти или отменить действие триггера. Это, с одной стороны, делает триггеры особенно полезными, например для регистрации действий пользователей, с другой — делает механизм триггеров потенциально опасным, поскольку ошибки в коде триггеров могут привести к существенному разрушению функциональности СУБД.

Действия, которые будут выполняться триггером, в системе PostgreSQL задаются функцией, которая должна быть определена в базе данных до создания триггера. Обычно функция триггера не имеет явно описанных параметров, потому что информация о контексте вызова функции может быть получена другим способом, и возвращает значение типа trigger. Такие функции могут быть написаны на любом процедурном языке программирования, который можно использовать в PostgreSQL, при этом способ доступа к контексту, в котором возбужден триггер, зависит от языка программирования. В функциях, написанных на языке PL/pgSQL, для этого доступны предопределенные переменные.

Рассмотрим, как в PostgreSQL определяются триггеры, возбуждаемые при модификации данных. Для определения триггера модификации используется оператор CREATE TRIGGER, в котором указывается следующая информация.

- Уровень триггера.

  Триггеры могут быть определены на уровне операторов SQL (FOR EACH STATEMENT) или на уровне строк (FOR EACH ROW). На уровне строк триггер вызывается для каждой строки таблицы, которая обновляется оператором SQL. На уровне оператора триггер выполняется один раз при исполнении возбуждающего оператора.

- Операторы, выполнение которых возбуждает триггер (INSERT, UPDATE, DELETE, TRUNCATE).

- Объект базы данных, при модификации которого запускается триггер (таблица или представление).

- Относительное время выполнения триггера (BEFORE, AFTER, INSTEAD OF).

  Триггеры BEFORE срабатывают непосредственно до, а триггеры AFTER — сразу после возбуждающего оператора. Триггеры INSTEAD OF используются только для представлений и позволяют определить, какие действия должны выполняться вместо операций модификации данных. Таким образом, после определения триггеров представления можно сделать неотличимыми (по функциям) от хранимых отношений не только для оператора выборки данных SELECT, но и для операторов обновления данных INSERT, UPDATE, DELETE.

- Дополнительные условия, ограничивающие запуск триггера (WHEN).

  Эти условия можно рассматривать как реализацию условий ECA-предписаний.

- Функция триггера, выполняющая необходимые действия.

- Возможно, дополнительные параметры функции триггера.

Заметим, что одна и та же функция триггера может использоваться для определения разных триггеров.

В PL/pgSQL для триггеров уровня строк определены переменные OLD и NEW, содержащие соответственно старое и новое значения строки. При этом для оператора INSERT не существует старого значения, а для DELETE — нового.

Триггеры BEFORE могут изменять значения атрибутов в переменной NEW. Для того чтобы выполнение оператора, возбудившего триггер, было нормально продолжено, функция триггера должна вернуть непустое (определенное) значение. В триггерах, определенных для операторов INSERT и UPDATE, это значение будет использоваться в качестве нового значения обновляемого кортежа, поэтому функция должна вернуть исходное или измененное значение переменной NEW.

Если функция триггера возвращает неопределенное значение NULL, то для триггеров уровня строк прекращается обработка соответствующей строки, а для триггеров уровня оператора прекращается выполнение всего оператора. Однако откат транзакции ни в том, ни в другом случае не производится.

Приведенная далее функция триггера выводит значения переменных, определяющих контекст вызова триггера, и не выполняет никаких других действий. Мы используем эту функцию для иллюстрации возможных определений триггеров.

```
demo=# CREATE OR REPLACE FUNCTION show_trigger_parameters()
RETURNS trigger
AS $$
BEGIN
  RAISE NOTICE '%: % %.% % %',
    TG_NAME, TG_OP, TG_TABLE_SCHEMA, TG_TABLE_NAME, TG_WHEN, TG_LEVEL;
  IF TG_OP = 'DELETE' THEN
    RETURN OLD;
  ELSE
    RETURN NEW;
  END IF;
END;
$$ LANGUAGE plpgsql;
CREATE FUNCTION
```

Воспользуемся определенной функцией для создания нескольких триггеров:

```
demo=# CREATE TRIGGER row_before
  BEFORE INSERT OR DELETE OR UPDATE
  ON flights
  FOR EACH ROW
  EXECUTE PROCEDURE show_trigger_parameters();
CREATE TRIGGER
demo=# CREATE TRIGGER row_after
  AFTER INSERT OR DELETE OR UPDATE
  ON flights
  FOR EACH ROW
  EXECUTE PROCEDURE show_trigger_parameters();
CREATE TRIGGER
demo=# CREATE TRIGGER stmt_before
  BEFORE INSERT OR DELETE OR UPDATE
  ON flights
  FOR EACH STATEMENT
  EXECUTE PROCEDURE show_trigger_parameters();
CREATE TRIGGER
demo=# CREATE TRIGGER stmt_after
  AFTER INSERT OR DELETE OR UPDATE
  ON flights
  FOR EACH STATEMENT
  EXECUTE PROCEDURE show_trigger_parameters();
CREATE TRIGGER
```

При выполнении операции обновления будут возбуждены все описанные выше триггеры:

```
demo=# BEGIN TRANSACTION;
BEGIN
```

```
demo=# UPDATE flights
  SET status = 'Cancelled'
  WHERE flight_id = 12345;
NOTICE:  stmt_before: UPDATE bookings.flights BEFORE STATEMENT
NOTICE:  row_before: UPDATE bookings.flights BEFORE ROW
NOTICE:  row_after: UPDATE bookings.flights AFTER ROW
NOTICE:  stmt_after: UPDATE bookings.flights AFTER STATEMENT
UPDATE 1
demo=# ROLLBACK;
ROLLBACK
```

Откат транзакции необходим, конечно, только потому, что мы не намерены оставлять подобные изменения в демонстрационной базе данных.

Для спецификации триггеров событий используется команда CREATE EVENT TRIGGER. Написание функций для этих триггеров и определение самих триггеров в целом аналогичны определениям функций и триггеров модификации данных.

Внутри функции триггера можно выполнять любые операторы SQL, допустимые в функциях. Это может привести к каскадному запуску другого или того же самого триггера, в том числе может вызвать бесконечную рекурсию, ответственность за предотвращение которой возложена на программиста. Заметим, что рекурсия может возникнуть вследствие определения триггера для других таблиц. Например, триггер на таблицу T_1, регистрирующий изменения в таблице T_2, может стать некорректным после того, как на таблице T_2 определяется триггер, модифицирующий T_1, т. к. обновление любой из этих таблиц приведет к запуску триггера, модифицирующего другую таблицу, т. е. к бесконечной рекурсии. Ошибки, связанные с применением триггеров, трудно обнаруживать потому, что триггеры вызываются неявно.

Триггеры являются удобным методом для решения некоторых задач, например для журналирования изменений в таблицах. Однако перенос логики на уровень триггеров базы данных несет в себе существенные риски. Хорошее обсуждение различных видов применений триггеров можно найти в [6].

Существует альтернативный метод описания способа модификации запросов с помощью *правил* (rules). Механизм правил позволяет переопределять не только операторы обновления, но и операторы выборки данных из таблиц и представлений. Мы не будем обсуждать этот метод.

8.7. Итоги

В этой главе мы обсудили различные расширения языка SQL, реализованные в системе PostgreSQL. Значительная часть этих расширений может быть охарактеризована как объектные средства. В частности, это относится к возможностям определения новых базовых типов данных, наследованию, использованию объектных указателей и определению пользовательских функций. Все это дает основания для того, чтобы называть PostgreSQL объектно-реляционной системой управления базами данных.

Особое место среди расширений занимают типы xml, json и jsonb, предназначенные для хранения и манипулирования данными в слабоструктурированных форматах. Эти типы не только иллюстрируют возможность создания новых типов данных, но и особенно важны в связи с их широким применением, например при передаче данных по информационным сетям.

Наконец, обработка событий в базе данных реализуется как в PostgreSQL, так и в других СУБД с помощью механизма триггеров.

8.8. Упражнения

Упражнение 8.1. Напишите запрос, выдающий список самолетов из демонстрационной базы в формате JSON.

Упражнение 8.2. Напишите запрос, выдающий список рейсов из демонстрационной базы в формате XML.

Упражнение 8.3. Напишите запрос, выдающий заданное бронирование в формате JSON, включая все входящие в него билеты и перелеты для каждого из билетов.

Упражнение 8.4. Решите задачу, обратную предыдущей: получив бронирование в формате JSON, вставьте в таблицы демонстрационной базы данных соответствующие строки.

Упражнение 8.5. Выполните два предыдущих упражнения, используя формат XML вместо JSON.

Упражнение 8.6. Создайте триггер, реализующий правило целостности в демонстрационной базе: рейсы могут совершать только те типы самолетов,

максимальная дальность полета которых превышает расстояние между аэропортами. Для расчета расстояния воспользуйтесь расширением earthdistance.

Упражнение 8.7. Создайте в базе данных триггер, который не позволит выполнять операторы CREATE в ночное время.

Упражнение 8.8. Создайте в демонстрационной базе вспомогательную таблицу и триггеры для аудита изменений рейсов. Изменения можно записывать в таблицу с тем же набором полей, а можно — в один JSON-столбец (что позволит избежать проблем при изменении структуры таблицы).

Упражнение 8.9. Создайте в демонстрационной базе событийный триггер, автоматически создающий для новых таблиц обычные триггеры для аудита изменений в этих таблицах.

Глава 9

Разновидности СУБД

9.1. Классы приложений БД

Как уже отмечено в главе 1, одной из основных предпосылок для выделения систем управления базами данных как отдельного класса программных систем стало появление устройств хранения данных с произвольным доступом относительно большой емкости. Именно возможности быстрого доступа к любым участкам пространства, в котором могут храниться данные, открыло возможности для создания приложений, работающих в режиме оперативного доступа и характеризующихся малым временем отклика.

В последующие десятилетия класс задач и приложений, для реализации которых использовались СУБД, постепенно расширялся. В настоящее время принято выделять два больших класса задач, отличающихся по характеру использования СУБД:

OLTP (On-Line Transaction Processing) — системы оперативной обработки (коротких) запросов;

OLAP (On-Line Analytical Processing) — системы оперативной аналитической обработки (больших объемов) данных.

Эти два больших класса не исчерпывают всех известных классов приложений СУБД, и границы этих классов нельзя считать четкими.

Системы оперативной обработки стали исторически первыми применениями баз данных, что в значительной мере определило требования к СУБД, упомянутые в главе 1. Подчеркнем, что в этом контексте термин *транзакция* обозначает совокупность действий, выполняемых приложением при однократном запуске, т. е. выполнение некоторой функции приложения, а не транзакцию в базе данных. В обоих случаях термин заимствован из прикладной области (банковские транзакции), однако в реальных системах обработка банковской транзакции обычно включает несколько транзакций в смысле баз данных. Даже такая

простая операция, как получение наличных в банкомате, состоит из нескольких транзакций СУБД, выполняемых в разных базах данных (банка — владельца банкомата, банка держателя карты и авторизационного центра).

Классические приложения класса OLTP характеризуются относительно большим потоком очень коротких транзакций, каждая из которых обрабатывает (считывает и зачастую изменяет) небольшое количество очень коротких записей (единицы или десятки записей, содержащих десятки или единицы сотен байтов каждая). Все транзакции в потоке независимы и выполняются от имени разных владельцев прав доступа. Все это определяет важность требований согласованности, целостности, быстрого поиска и обновления, отказоустойчивости, разграничения доступа и других.

К этому классу примыкают современные интернет-приложения, в которых в качестве транзакций выступают HTTP-запросы. Для таких приложений характерны несколько большие объемы обрабатываемых данных и несколько меньшая доля обновлений. Для небольшого числа лидеров отрасли наиболее важным требованием является масштабируемость, трудно (и чрезмерно дорого) достижимая при использовании СУБД общего назначения, что ведет к разработке специализированных систем и реализации упрощенных моделей управления данными. Однако для компаний малого и среднего размеров применение технологий традиционного типа, в частности на основе PostgreSQL, оказывается более эффективным.

Системы оперативной аналитической обработки предназначены для формирования обобщенных отчетов, получаемых обработкой всех или значительной доли записей, хранящихся в базе данных. Такие системы характеризуются полным отсутствием обновлений, что делает многие требования к СУБД, например согласованность и целостность, менее актуальными. Для того чтобы удовлетворить требования по времени отклика, обычно необходимо создавать вторичные копии данных, организация хранения которых существенно отличается от хранения первичных источников данных, и материализовывать промежуточные результаты, необходимые для быстрого получения окончательных отчетов.

Особый класс составляют системы совместного редактирования и разработки, в том числе системы автоматизированного проектирования (САПР). Подобные системы характеризуются относительно малым количеством объектов очень большого размера и специфическими требованиями, отличающимися от требований к СУБД общего назначения. Так, в системах этого класса обычные

требования к транзакциям оказываются слишком ограничительными: изменения, вносимые в процессе редактирования, могут занимать большое время, поэтому изоляция и атомарность оказываются нежелательными, хотя согласованность и долговечность остаются важными.

В таких системах хорошо проявили себя объектно-ориентированные СУБД, однако зачастую в них (вместо СУБД) используются специализированные надстройки над файловыми системами. В недавнем прошлом базы данных для хранения и обработки документов в слабоструктурированных форматах позиционировались как отдельный класс систем, однако в настоящее время их функциональность интегрирована в СУБД общего назначения.

Особые классы приложений связаны с обработкой очень длинных последовательностей (например, временны́х рядов или геномов), а также с хранением и обработкой больших графов.

9.2. Структуры хранения

Многообразие классов приложений, в которых используются СУБД, приводит к необходимости реализовывать и поддерживать большое разнообразие структур хранения.

Так, с точки зрения структур хранения, для задач OLTP характерны объекты с небольшим числом атрибутов, при этом почти все атрибуты используются почти в каждом запросе, обрабатывающем такой объект. Поэтому системы, ориентированные на этот класс задач, размещают атрибуты каждого объекта данных по возможности в смежных участках памяти, а взаимосвязанные объекты, которые часто обрабатываются вместе, также могут группироваться. В терминах реализаций реляционной модели данных такое хранение принято называть «хранением по строкам». Такой способ организации хранения рассматривается как основной (если не единственный) в большинстве СУБД общего назначения, в том числе в PostgreSQL.

С другой стороны, для задач OLAP характерны записи, содержащие большое количество атрибутов, однако в каждом запросе используется только их небольшое подмножество. Это обстоятельство делает альтернативный способ хранения «по колонкам» более эффективным для таких задач в силу целого ряда факторов: более быстрое последовательное сканирование, возможность

сжатия данных, отсутствие обновлений и др. В то же время колоночная структура хранения существенно усложняет такие операции, как выборка всех атрибутов одного объекта, что делает ее менее эффективной для задач OLTP.

Исследования относительной эффективности строкового и колоночного представлений ведутся с середины 70-х годов, и к настоящему времени сформировалось понимание того, в каких случаях тот или другой способ оказывается более эффективным.

Точно так же относительная полезность и эффективность индексных структур существенно зависят от класса приложения. Например, индексы, которые хорошо работают для OLTP, оказываются малополезными для OLAP. Большое разнообразие индексных структур, реализованных в системе PostgreSQL, и в особенности возможности добавления новых типов индексов создают огромный потенциал для разработки высокоэффективных систем на основе этой СУБД.

9.3. Архитектуры связи с приложениями

Одним из основных требований к системам управления базами данных является возможность совместного использования данных различными приложениями (или разными экземплярами одного приложения).

Поскольку основную нагрузку в ранних системах создавали приложения OLTP, для которых важно малое время отклика, необходимо было обеспечить совместное одновременное использование данных, т. е. запросы к БД должны были выполняться параллельно. По этим причинам практически все ранние СУБД использовали архитектуру, в которой приложение и программные компоненты системы выполняются в разных процессах. При этом процесс приложения направляет запросы и получает ответы на них, взаимодействуя с процессом, в котором выполняется СУБД.

Несколько позже для обозначения таких взаимодействий стали применяться термины *клиент* и *сервер*. Эти термины используются в разных смыслах. В этой книге мы обозначаем ими роли при взаимодействии программных или аппаратных компонент, при этом одна и та же компонента может иметь разные роли в разных взаимодействиях. В контексте баз данных клиент-серверная архитектура взаимодействия позволяет обеспечить доступ к базе данных нескольких сотен и даже тысяч сеансов одновременно.

Появление персональных компьютеров привело к относительно широкому распространению однопользовательских систем управления базами данных, которые, как правило, использовали табличное представление логической структуры данных, но упрощенный язык манипулирования данными, не включающий вычислительно сложные операции, например операции соединения. Недостаточная мощность вычислительных систем, на которых работали такие СУБД, заставила отказаться от ряда требований, в частности от совместного использования данных, транзакций и т. п. Это привело к отказу от клиент-серверной архитектуры для этого класса СУБД. Совместное использование данных в таких системах реализуется размещением файлов базы данных на файл-сервере. Конечно, подобная архитектура не может обеспечить одновременную работу с общей базой данных большого количества пользователей (сопоставимого с возможностями клиент-серверных систем).

В настоящее время однопользовательские базы данных широко применяются в качестве встроенных систем в мобильных устройствах. В этом случае необходимость совместного использования данных на уровне базы данных не возникает, а потребление вычислительных ресурсов ограничивается емкостью аккумуляторной батареи.

Создание соединений с сервером базы данных и ведение активных сеансов требуют некоторых ресурсов на сервере БД. Поэтому системы управления базами данных, использующие архитектуру клиент-сервер, оказывались недостаточно масштабируемыми по числу активных соединений для некоторых классов OLTP-приложений, характеризующихся очень большим количеством пользователей.

Решением этой проблемы стала *многослойная* (multi-tier) архитектура, в которой часть приложения выполняется на системе клиента, а другая часть — на сервере. Ранние реализации таких серверов назывались мониторами транзакций, а позже стали называться *серверами приложений* (application servers). В такой архитектуре приложение по-прежнему является клиентом сервера базы данных и одновременно — сервером, обслуживающим следующий уровень клиентов (например, выполняя HTTP-запросы).

Масштабируемость по количеству клиентов достигается за счет использования ограниченного пула сеансов для работы сервера приложений с базой данных. Поскольку сервер приложений не хранит никаких общих данных, появляется возможность горизонтального масштабирования, т. е. установки дополнительных серверов приложений, но возникает необходимость в переносе некоторых

функций (например, разграничения доступа) на уровень сервера приложений или в само приложение.

Увеличение пропускной способности прикладных систем, достигаемое за счет использования серверов приложений, может приводить к некоторым негативным последствиям, компенсация которых требует дополнительных усилий при разработке приложений. Так, многократное использование соединений (сеансов работы) с базой данных может приводить к накоплению временных объектов на сервере базы данных, которые обычно удаляются при завершении сеанса. Кеширование данных на сервере приложений, как обсуждалось в главе 7, может потребовать дополнительных усилий для согласования обновлений, выполняемых через разные серверы приложений в одной базе данных.

9.4. Оборудование

9.4.1. Носители данных

На протяжении многих десятилетий единственным видом устройств, пригодным для хранения больших баз данных, оставались вращающиеся магнитные диски. В качестве основных свойств этих устройств, наиболее важных для выбора физической организации хранения данных и алгоритмов их обработки, можно назвать:

- время доступа к данным мало зависит от расположения этих данных на носителе;
- время доступа на несколько порядков превышает время доступа к данным в оперативной памяти;
- скорость передачи данных между устройством и оперативной памятью относительно высока;
- последовательное чтение или запись соседних участков носителя может выполняться на 1–2 порядка быстрее, чем произвольный доступ.

Из этих свойств, в частности, следует, что время считывания и записи относительно больших блоков (страниц) данных почти не отличается от времени обработки коротких записей, поэтому практически все структуры хранения баз данных размещают данные в блоках, которые считываются в оперативную память для обработки с последующей записью измененного состояния на диск.

Другими словами, фактически обработка происходит не на самом носителе, а в кеше (буфере), находящемся в оперативной памяти.

Разница во времени доступа к дискам и к оперативной памяти определяет метрики, необходимые для оценки, например индексных структур и моделей стоимости для основных операций базы данных, используемых оптимизатором. В случае если операция использует основное хранилище данных (на диске), доминирующим слагаемым в оценке ее стоимости является количество обращений (или время доступа) к дисковой памяти.

Появление твердотельных накопителей (SSD) достаточно большой емкости привело к необходимости пересмотра моделей стоимости и некоторых алгоритмов, поскольку характеристики SSD существенно отличаются от характеристик вращающихся дисков. Однако пока это не привело к радикальному изменению подходов к организации хранения данных в СУБД.

Быстрый рост размеров и снижение стоимости оперативной памяти вызвал рост интереса к системам баз данных, хранящим все данные в оперативной памяти. Основной проблемой таких систем является энергозависимость такой памяти, что требует специальных мер для предотвращения потери данных при отключении электропитания.

Можно выделить следующие основные направления и подходы, связанные с базами данных в оперативной памяти.

- Традиционные дисковые базы данных с очень большим размером оперативной памяти, выделенной для кеширования. Увеличение производительности в этом случае достигается за счет того, что вся обработка выполняется в оперативной памяти. Очевидным недостатком этого подхода является ограничение на размер базы данных, а узким местом часто оказывается запись обновленных данных на диск.
- Системы, не обеспечивающие сохранности данных при отключении электропитания. Такие системы зачастую используются для кеширования и ставятся перед традиционными дисковыми системами. Отличие от предыдущего класса систем состоит в том, что копирование данных в память происходит на уровне логической схемы, а не на уровне структур хранения.
- Распределенные системы баз данных в оперативной памяти, в которых каждый элемент данных хранится в нескольких копиях на разных узлах

системы. Сохранность данных в этом случае основана на предположении о том, что одновременное отключение питания на всех узлах, хранящих копии данных, маловероятно. Производительность подобных систем ограничивается пропускной способностью сети, связывающей узлы распределенной системы, и необходимостью синхронизации разных копий каждого элемента данных.

- Системы, использующие данные почти исключительно для чтения (например, базы данных для аналитической обработки).

Размещение данных в оперативной памяти открывает широкие возможности для применения альтернативных структур хранения, например для индексов.

Появление в последние годы энергонезависимых запоминающих устройств, обеспечивающих время доступа, сопоставимое со временем доступа к оперативной памяти (например, памяти с изменением фазового состояния, phase-change memory), открывает очень широкие возможности для систем управления базами данных, но потребует радикального пересмотра архитектуры средств хранения в составе СУБД.

9.4.2. Вычислительные ресурсы

Увеличение количества одновременно работающих устройств является концептуально наиболее простым способом увеличения мощности системы. В контексте высокопроизводительных СУБД параллельные системы существуют на протяжении десятилетий. В последние годы в связи со снижением темпов роста производительности отдельных устройств интерес к параллельной обработке значительно вырос.

Важно отметить, что увеличение вычислительной мощности (например, количества процессоров) не приводит автоматически к соответствующему увеличению производительности СУБД и программных комплексов, работающих на их основе. Для того чтобы потенциал параллельного оборудования реализовался, необходимо, чтобы структуры данных и алгоритмы могли эффективно использовать возможности такого оборудования.

Основными характеристиками качества параллельных систем являются:

- ускорение;
- масштабируемость по пропускной способности;

- масштабируемость по времени отклика.

В зависимости от конфигурации оборудования различаются следующие классы параллельных систем управления базами данных:

SM (Shared Memory) — многопроцессорные системы с общей оперативной памятью (и дисками), в которых не требуется пересылка данных для выполнения операций на разных процессорах;

SD (Shared Disks) — системы, в которых каждый из процессоров имеет свою оперативную память, недоступную другим процессорам, но все дисковые устройства доступны с любого процессора;

SN (Shared Nothing) — системы без разделения каких-либо устройств, взаимодействие между частями параллельной системы всегда использует вычислительную сеть.

Многие высокопроизводительные СУБД автоматически распознают многопроцессорные конфигурации и могут их использовать без какой-либо дополнительной ручной настройки. Однако возможности масштабирования многопроцессорных систем (SM), очевидно, ограничены.

Особо следует отметить многоядерные процессоры. Хотя отдельные ядра могут работать как независимые процессоры, оказывается, что такая работа может быть малоэффективна в контексте баз данных. Дело в том, что ядра используют общий кеш, и обычные алгоритмы параллельного выполнения основных операций баз данных эффективно работают только при малых объемах данных, не переполняющих этот кеш, т. е. для нагрузок типа OLTP, но не OLAP.

Системы класса SN допускают практически неограниченное масштабирование оборудования, однако пропускная способность сети может быть узким местом, в особенности при использовании сложных запросов. Фактически достигаемые значения характеристик производительности таких систем могут существенно зависеть от размещения данных и от смеси запросов, выполняемых сервером баз данных.

Отметим, что системы класса SN по аппаратной конфигурации мало отличаются от распределенных систем и иногда называются распределенными. Мы тем не менее будем различать эти термины. Параллельные системы баз данных создаются с целью повышения производительности систем, в то время как целью создания распределенных систем является в первую очередь повышение доступности и отказоустойчивости.

9.5. Хранилища данных

Реляционные СУБД по-прежнему являются доминирующими, однако нельзя игнорировать популярность альтернативных архитектур систем хранения данных, объединяемых направлением баз данных NoSQL. Сторонники таких систем обещают более высокую производительность и масштабируемость, по сравнению с реляционными СУБД, и пытаются сократить потери, возникающие из-за несоответствия между моделями и представлением данных в приложении и в хранилище.

По мере того как растет количество пользователей интернета, растет и объем производимых данных. Например, в 2016 году пользователи Twitter каждую секунду генерировали 6 000 записей, пользователи YouTube каждую минуту добавляли 300 часов видео, а пользователи Instagram каждый день публиковали 80 млн фотографий. С другой стороны, обработка этих данных довольно проста и, во всяком случае, не требует выполнения каких-либо сложных запросов.

Для обработки такого объема данных используют горизонтальное масштабирование. Это означает, что нагрузка распределяется между большим количеством серверов, которые объединены в кластер, и увеличение производительности достигается путем добавления новых машин. Конфигурация коммерческих баз данных для работы на кластерах требует настройки и дополнительных программных компонент, поэтому в NoSQL-сообществе считается, что реляционные системы плохо приспособлены к горизонтальному масштабированию. Действительно, вместо конфигурирования СУБД в системах NoSQL достаточно подготовить конфигурационные файлы, обычно в формате XML.

Поскольку для таких приложений, как правило, используются достаточно простые структуры данных и почти не используются сложные запросы, мощные возможности традиционных СУБД оказываются незадействованными. В частности, традиционные СУБД не могут показать высокую производительность при отсутствии запросов на массовую обработку.

Эти факторы привели к появлению NoSQL-систем, которые изначально проектировались распределенными, причем распределенность достигается «из коробки» и не требует сложной настройки. Сложность, конечно, не исчезает, однако настройка выполняется на другом уровне.

Экстремизм первых пропагандистов движения NoSQL, полностью отрицавших полезность высокоуровневых языков запросов, сменился более осторожными формулировками. Термин NoSQL в последние годы означает «Not only SQL»,

т. е. «не только SQL». Возможно, это связано с тем, что многие NoSQL-системы обзавелись реализациями ограниченных подмножеств SQL (которые, однако, не могут соревноваться с реляционными системами по производительности).

Разумеется, системы обработки данных, не использующие SQL или использующие не только SQL, существовали и раньше. Однако именно термин NoSQL стал популярным после того, как его начали использовать в качестве метки (хештега) в различных информационных сообществах для обозначения нереляционных систем, созданных в XXI столетии.

Некоторые авторы, однако, относят к этому классу также и интенсивно разрабатывавшиеся в начале 90-х годов объектно-ориентированные системы баз данных.

Согласно утверждениям сторонников этого движения, NoSQL-системы обладают некоторыми общими характеристиками.

Не используют схему данных. Отказ от использования схемы данных означает, по существу, отказ от независимости данных и приложений и приводит к фактической невозможности совместного использования базы данных различными приложениями. Этот отказ становится возможным благодаря использованию средств синхронизации данных (обмена сообщениями) на уровне приложений.

Слабо поддерживают транзакционную семантику. В отсутствие совместного использования данных задачи поддержки согласованности решаются на уровне приложения, а не СУБД.

Приспособлены для использования на кластерах. Предельная простота применяемых структур хранения и отсутствие явных взаимосвязей между элементами данных, известными на уровне системы (а не приложения), практически устраняет необходимость координации при обработке данных, размещенных на различных серверах, входящих в вычислительный кластер.

Пропоненты NoSQL-систем в качестве отличительного свойства указывают на то, что такие системы являются системами с открытым кодом. В действительности это свойство не может считаться отличительным, потому что многие СУБД с открытым кодом (в частности, PostgreSQL) никак не могут быть отнесены к NoSQL, и не все системы NoSQL являются системами с открытым кодом. Открытость кода нельзя считать техническим свойством, однако оно оказывается существенным при выборе архитектуры, в особенности если систему

предполагается устанавливать на большое количество компьютеров для реализации горизонтальной масштабируемости.

Разные классы приложений предъявляют разные требования к целостности, согласованности и непротиворечивости данных. Очевидно, что у финансовых приложений такие требования значительно выше, чем, например, у социальных сетей. Базы данных класса NoSQL потенциально позволяют получить высокую производительность за счет ослабления ограничений целостности, более слабой поддержки транзакционной семантики и отказа от использования высокоуровневых языков запросов.

NoSQL-системы можно разделить на два типа: агрегатно-ориентированные базы данных и базы данных на основе графов.

9.5.1. Агрегатно-ориентированные базы данных

В NoSQL-системах выделяют 3 класса агрегатно-ориентированных баз данных.

Системы ключ-значение. Эти системы хранят пары ключ-значение. При этом значение может быть любым: например, строкой, документом или картинкой. Хранилище сохраняет значение в двоичном виде, ничего не знает о его структуре и не контролирует его тип. Чтобы получить значение, нужно обязательно знать ключ; выборки по каким-либо другим параметрам значения не поддерживаются. Соответственно, такие системы используются, когда нужен доступ по (единственному) ключу, а другие виды поиска и взаимосвязи между объектами не требуются.

Самые известные представители этого класса — Redis и Memcached.

Документоориентированные системы. Эти системы хранят произвольные структуры данных, как правило, в формате JSON. В отличие от систем ключ-значение, документоориентированные системы поддерживают запросы, позволяющие находить документы по значениям определенных полей. Несмотря на то что документ может иметь произвольную структуру, запрос, выбирающий документы с определенным значением поля, подразумевает наличие этого поля, т. е. схему данных.

По своим функциям эти системы занимают нишу, в которой ранее находились системы, ориентированные на хранение и обработку данных в формате XML (базы данных Native XML).

Примерами таких систем являются MongoDB и CouchDB.

Хранилища семейств колонок. Такие системы являются расширением систем ключ-значение. Данные в них хранятся в виде разреженной матрицы, строки и столбцы которой используются как ключи. Такие системы принято выделять в отдельный тип баз данных NoSQL, но мы описываем их в разделе 9.1, поскольку именно этот способ организации данных используется для OLAP-систем (хотя структуры хранения могут существенно отличаться).

Примеры хранилищ семейств колонок — HBase и Cassandra.

В трех перечисленных типах NoSQL-систем сохраняемый объект может иметь сложную структуру, однако система всегда сохраняет это значение целиком, т. е. работает с ним, как с агрегатом. Агрегаты не зависят друг от друга, поэтому их несложно распределить по узлам кластера, увеличивая масштабируемость системы. ACID-транзакции в таких системах могут поддерживаться только на уровне агрегатов.

9.5.2. Базы данных на основе графов

Данные в этих системах представляют в виде графов, ребро показывает наличие взаимосвязи между объектами. Хранятся ребра, поэтому дополнительных вычислений для обхода графа не требуется. Такие базы данных используются в рекомендательных системах, например для предложений новых продуктов на основании предыдущих покупок пользователя или в социальных сетях, когда нужно получать друзей друзей. Для выборки подграфов используется специальный язык запросов. В отличие от агрегатно-ориентированных, базы данных на основе графов поддерживают ACID-транзакции.

Примером такой системы является Neo4j.

9.6. Выбор СУБД для построения информационных систем

Выбор СУБД является важным решением, и при его принятии нужно учитывать функциональные и нефункциональные требования конкретного проекта. Ошибка в выборе СУБД стоит очень дорого, поскольку перевод проекта на новую СУБД неизбежно повлечет переработку программного кода и миграцию

данных. При этом нужно учитывать не только требования для ближайшей запланированной версии, но и вероятность последующих изменений функциональности, а также предположительный рост нагрузки. В случае создания программного продукта, который будет дорабатываться у заказчика, важна легкость внесения изменений для специфических требований проекта.

Для небольших систем обычно достаточно теоретической проверки, насколько выбранная СУБД соответствует требованиям. В более сложных случаях может быть оправдано создание испытательного проекта (proof of concept), который протестирует несколько кандидатов и поможет выбрать наиболее подходящего. Однако кандидаты для такого проекта также должны выбираться на основании некоторых критериев. Мы выделяем следующие группы требований.

Функциональные требования. Все современные реляционные СУБД поддерживают базовый набор операторов SQL для определения схемы, выборки и манипулирования данными. Следует обратить внимание на требования, для реализации которых могут потребоваться нестандартные типы данных (XML, JSON, геолокационные данные, типы для работы с временны́ми интервалами). Для многих бизнес-приложений необходимы полнотекстовый поиск и многоязыковая поддержка. Принципиальным является вопрос, какая часть логики будет реализована на сервере баз данных, а какая — в приложении. Чем больше логики планируется разместить на уровне СУБД, тем более важной становится поддержка хранимых процедур и триггеров.

Нефункциональные требования. Сюда относятся требования к производительности системы, масштабируемости, безопасности, созданию резервных копий и процессу восстановления после сбоев. Данные требования тесно связаны с бюджетом проекта, поскольку часто стоимость лицензий зависит от необходимого аппаратного обеспечения. Обычно это прямая зависимость от количества ядер процессоров.

Возможности для сопровождения системы. Сопровождение включает в себя инсталляцию и настройку системы, обновление программных компонент, обеспечение бесперебойного функционирования. Необходимо обратить внимание на инструменты для администрирования СУБД и оценить, насколько трудоемкими являются рутинные задачи сопровождения.

В процессе разработки или эксплуатации систем могут быть найдены ошибки в программном коде самого сервера баз данных, для устранения

которых нужно будет обратиться к разработчикам СУБД. В случае использования коммерческой СУБД обычно заключается договор поддержки на определенный срок, в который включено исправление ошибок.

Если выбрана система с открытым программным кодом, важны распространенность СУБД и полнота ее документации. Также следует обратить внимание на активность сообщества ее пользователей. Применение систем с открытым кодом дает возможность выбора между внешней поддержкой на основе договора и поддержкой собственными силами. При этом, как правило, второй вариант оказывается более дорогостоящим.

Бюджет проекта. Важной особенностью значительной части коммерческих СУБД является высокая стоимость лицензий на их использование. Эта особенность оказывается особенно ощутимой для высокопроизводительных (например, параллельных) и высоконадежных конфигураций. При этом сопровождение систем, построенных на основе таких СУБД и в таких конфигурациях, также оказывается технически сложным и дорогостоящим. И конечно, значительная часть бюджета проекта закладывается на заработную плату сотрудников, занимающихся разработкой и сопровождением. Эта сумма также различается в зависимости от выбранной СУБД. Обязательно нужно учитывать, насколько сложно найти сотрудников, обладающих необходимой квалификацией.

Финансовый вопрос часто оказывается решающим. В связи с этим в последние годы (и, возможно, пару десятилетий) растет популярность альтернативных моделей создания и эксплуатации систем, предполагающих интенсивное использование данных. В рамках этой тенденции можно выделить следующие популярные направления.

Отказ от использования функциональности СУБД. По существу, это означает перенос функциональности СУБД на уровень приложения, на которое возлагается ответственность за выполнение сложных операций обработки данных, поддержка согласованности и целостности, а также обмен данными с другими приложениями (поскольку совместное использование данных становится затруднительным).

Использование СУБД с ограниченными характеристиками или NoSQL-систем. Это хороший вариант для случая, когда выбранная СУБД полностью удовлетворяет всем требованиям проекта. Иначе необходимые характеристики достигаются за счет применения избыточных аппаратных ресурсов и, как и в предыдущем случае, ограничения функциональности системы.

Использование реляционных СУБД с открытым кодом. На сегодняшний день системы управления базами данных с открытым кодом являются достаточно привлекательными. Благодаря качественному коду и хорошей документации эти СУБД начинают обращать на себя внимание как малых предприятий, так и больших корпораций.

В этом контексте особую роль играет СУБД PostgreSQL, которая, оставаясь системой с открытым кодом, по своим эксплуатационным характеристикам приближается к высокопроизводительным коммерческим системам.

9.7. Итоги главы и книги

Системы управления базами данных появились как отдельный класс программного обеспечения в 60-е годы прошлого века. На протяжении всего периода существования систем этого класса происходит постоянное развитие их возможностей, связанное с непрерывным расширением области применения. Ранние СУБД применялись в небольшом количестве предметных областей, характеризующихся хорошо проработанной формализацией (финансы, телекоммуникации), а в настоящее время базы данных находятся в ядре любой информационной системы.

Развитие практически используемых промышленных систем сопровождалось развитием теории, в первую очередь теоретической реляционной модели данных, создавшей основания для применений высокоуровневых языков запросов и средств концептуального моделирования структур данных.

В современных системах управления базами данных реализуются высокоэффективные алгоритмы хранения, индексирования, оптимизации и выполнения запросов, протоколы, обеспечивающие корректное совместное использование разделяемых (общих) данных между несколькими приложениями и пользователями, а также средства надежного хранения и восстановления при отказах системы, которые могут обеспечить сколь угодно высокую степень защищенности данных от самых разнообразных отказов и разрушений.

Развитие больших систем управления базами данных, предоставляющих мощные возможности для создания разнообразных информационных систем, сопровождается появлением специализированных систем, предназначенных для решения вновь возникающих задач обработки данных. В последние годы подобные системы объединяются общим термином NoSQL. Ранее подобную

роль выполняли системы для хранения данных в формате XML, а в конце прошлого века — объектно-ориентированные СУБД.

Методы и технологии, отработанные в реализациях специализированных систем, зачастую включаются в состав больших универсальных систем управления базами данных. Так, все современные универсальные СУБД реализуют объектные расширения базовой реляционной модели данных, средства хранения и манипулирования XML и JSON. Современные СУБД могут работать в неоднородных распределенных системах, обрабатывая не только данные, которые хранят сами, но и данные из других источников.

Система PostgreSQL дает широкие возможности для выбора конфигурации и обеспечивает эффективную работу в очень широком диапазоне требований по объемам хранимых и обрабатываемых данных, типам и структурам данных, составу оборудования и контексту взаимодействия с другими системами.

Как известно, геометрические теоремы опровергались бы, если бы они задевали интересы людей. Поскольку системы управления базами данных интенсивно используются на практике, работы в этой области задевают интересы людей, поэтому в технической, популярной и даже в научной литературе можно найти огромное количество неточных или неверных утверждений, непроверенных или неверных фактов. Любые аргументы, связанные со сравнительными характеристиками систем, необходимо проверять по другим источникам или экспериментально.

9.8. Упражнения

Упражнение 9.1. Приведите основные отличительные черты OLTP и OLAP.

Упражнение 9.2. По каким критериям принято оценивать качество параллельных систем?

Упражнение 9.3. Какими общими характеристиками обладают системы, относящиеся к классу NoSQL?

Упражнение 9.4. Какие факторы определяют выбор СУБД для построения информационных систем?

Список литературы

1. *Bachman C. W.* The Programmer As Navigator // Commun. ACM. — 1973. — Vol. 16, no. 11. — Pp. 653–658. — ISSN 0001-0782. — DOI: 10.1145/355611.362534.
2. *Bernstein P. A.* Synthesizing Third Normal Form Relations from Functional Dependencies // ACM Trans. Database Syst. — 1976. — Vol. 1, no. 4. — Pp. 277–298. — ISSN 0362-5915. — DOI: 10.1145/320493.320489.
3. *Cattell R.*, *Barry D.*, *Berler M.* The Object Data Standard: ODMG 3.0. — Morgan Kaufmann, 2000. — ISBN 978-1-558-60647-0.
4. *Celko J.* Joe Celko's SQL for Smarties: Advanced SQL Programming. — 5th ed. — Morgan Kaufmann, 2014. — P. 852. — ISBN 978-0-128-00761-7.
5. *Celko J.* Joe Celko's Thinking in Sets: Auxiliary, Temporal, and Virtual Tables in SQL. — Morgan Kaufmann, 2008. — P. 384. — ISBN 978-0-123-74137-0.
6. *Ceri S.*, *Cochrane R.*, *Widom J.* Practical Applications of Triggers and Constraints: Success and Lingering Issues (10-Year Award) // Proceedings of the 26th International Conference on Very Large Data Bases. — Morgan Kaufmann, 2000. — Pp. 254–262. — (VLDB '00). — ISBN 978-1-558-60715-6.
7. *Chen P. P.-S.* The Entity-relationship Model: Toward a Unified View of Data // SIGIR Forum. — 1975. — Vol. 10, no. 3. — Pp. 9–9. — ISSN 0163-5840. — DOI: 10.1145/1095277.1095279.
8. *Codd E. F.* A Relational Model of Data for Large Shared Data Banks // Commun. ACM. — 1970. — Vol. 13, no. 6. — Pp. 377–387. — ISSN 0001-0782. — DOI: 10.1145/362384.362685.
9. *Codd E. F.* Extending the Database Relational Model to Capture More Meaning // ACM Trans. Database Syst. — 1979. — Vol. 4, no. 4. — Pp. 397–434. — ISSN 0362-5915. — DOI: 10.1145/320107.320109.
10. *Curtice R. M.* Data Base Design Using IMS/360 // Proceedings of the December 5–7, 1972, Fall Joint Computer Conference, Part II. — ACM, 1972. — Pp. 1105–1110. — (AFIPS '72). — DOI: 10.1145/1480083.1480146.
11. *Fagin R.* The Decomposition Versus Synthetic Approach to Relational Database Design // Proceedings of the 3rd International Conference on Very Large Data Bases — Volume 3. — VLDB Endowment, 1977. — Pp. 441–446. — (VLDB '77).
12. *Faroult S.*, *Robson P.* The Art of SQL. — O'Reilly Media, 2006. — P. 372. — ISBN 978-0-596-55536-8.

13. Feature Analysis of Generalized Data Base Management Systems: CODASYL Systems Committee, May 1971 / B. K. Bhargava [et al.]. — ACM, 1971.

14. *Maier D.* Theory of Relational Databases. — Computer Science Press, 1983. — P. 656. — ISBN 978-0-914-89442-1.

15. *Pavlo A.* What Are We Doing With Our Lives?: Nobody Cares About Our Concurrency Control Research // Proceedings of the 2017 ACM International Conference on Management of Data. — ACM, 2017. — Pp. 3–3. — (SIGMOD '17). — ISBN 978-1-4503-4197-4. — DOI: 10.1145/3035918.3056096.

16. *Winand M.* SQL Performance Explained: Everything Developers Need to Know about SQL Performance. — Markus Winand, 2012. — P. 204. — ISBN 978-3-9503078-2-5.

17. Информационные системы общего назначения: аналитический обзор систем управления базами данных / под ред. Е. Ющенко. — М.: Статистика, 1975. — С. 472.

18. *Кузнецов С. Д.* Базы данных: учебник для студ. учреждений высшего проф. образования. — М.: Академия, 2012. — С. 496. — (Университетский учебник. Прикладная математика и информатика). — ISBN 978-5-7695-8430-5.

19. *Мейер Д.* Теория реляционных баз данных. — М.: Мир, 1987. — С. 608.

20. *Моргунов Е.* PostgreSQL. Основы языка SQL.: учеб. пособие / под ред. Е. Рогова, П. Лузанова. — СПб.: БХВ-Петербург, 2018. — С. 336. — ISBN 978-5-9775-4022-3. — URL: https://postgrespro.ru/education/books/sqlprimer.

21. *Селко Д.* SQL для профессионалов. Программирование. — М.: Лори, 2004. — С. 456. — ISBN 978-5-85582-219-9.

22. Лекции лауреатов премии Тьюринга за первые 20 лет / под ред. Ю. Баяковского. — М.: Наука, 1993. — С. 560. — ISBN 978-5-03-002130-0.

Предметный указатель

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.

При оформлении заказа следует указать адрес (полностью), по которому
должны быть высланы книги; фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: www.a-planeta.ru.
Оптовые закупки: тел. +7 (499) 782-38-89.
Электронный адрес: books@alians-kniga.ru.

Новиков Борис Асенович
Горшкова Екатерина Александровна

Основы технологий баз данных

Учебное пособие

При поддержке Postgres Professional
https://postgrespro.ru

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Редактор *Рогов Е. В.*
Корректор *Синяева Г. В.*
Дизайн обложки *Климковский А. В.*

Формат $70\times100^{1}/_{16}$.
Гарнитура «ПТ Сериф». Печать цифровая.
Усл. печ. л. 19,5. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.ru