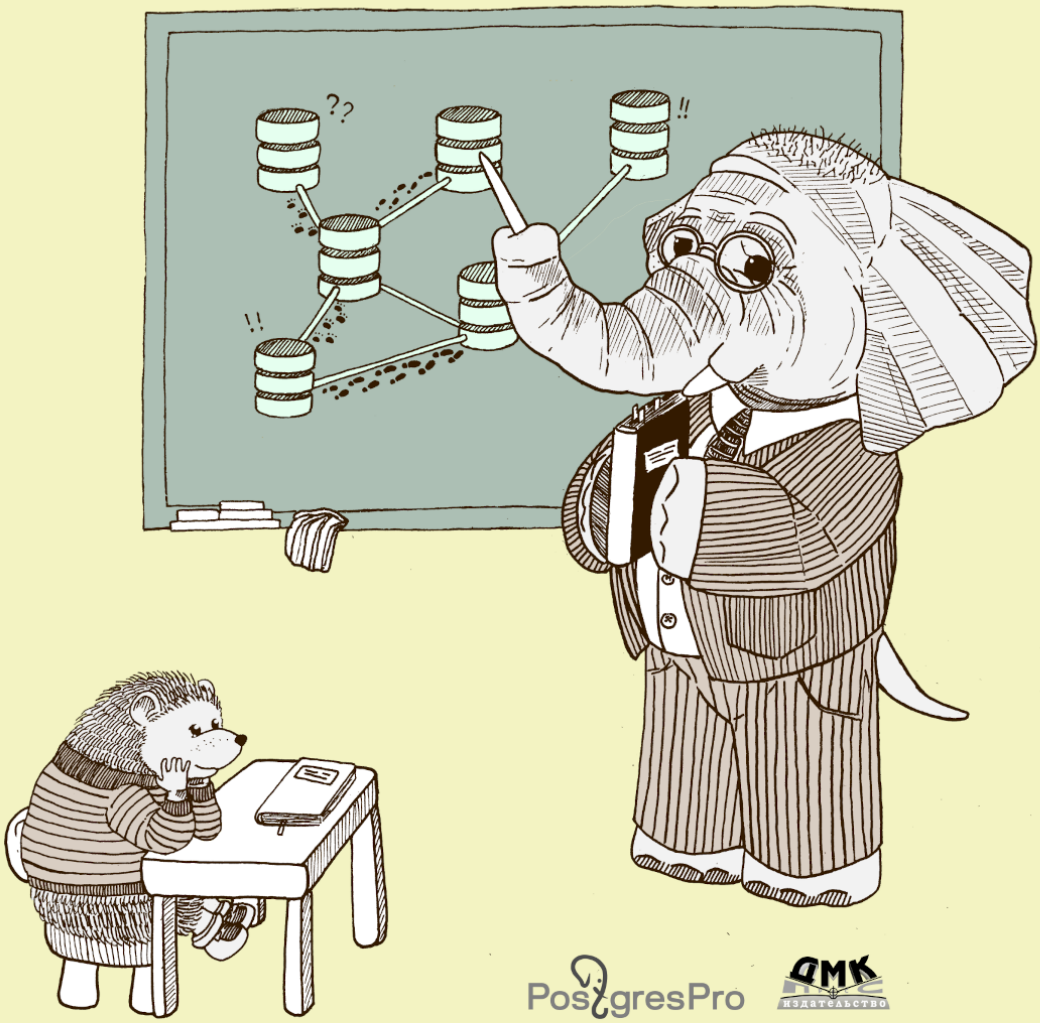
Владимир КОМАРОВ

ПУТЕВОДИТЕЛЬ ПО БАЗАМ ДАННЫХ



PostgresPro

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Компания Postgres Professional

Владимир Комаров

Путеводитель по базам данных

Рекомендовано Институтом системного программирования им. В. П. Иванникова Российской академии наук (ИСП РАН) в качестве учебного пособия

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Москва, 2024

УДК 004.65
ББК 32.972.134
К63

Комаров В. И.

К63 Путеводитель по базам данных. — М.: ДМК-Пресс, 2024. — 520 с.
ISBN 978-5-93700-287-7

Книга рассказывает об архитектурных принципах, на которых базируются все современные системы управления базами данных, а также об алгоритмах и структурах данных, которые в них используются. Особое внимание уделено сравнению реализаций одних и тех же подходов в близких по функциональности платформах. Кроме того, предлагается оригинальная классификация СУБД, рассматриваются вопросы эксплуатации и обеспечения безопасности.

Для широкого круга ИТ-специалистов и студентов профильных вузов.

Сайт книги: postgrespro.ru/education/books/dbguide.

УДК 004.65
ББК 32.972.134

ISBN 978-5-93700-287-7

Оглавление

Содержание

Моей дочери Ольге,
благодаря которой всё это имеет смысл

От автора

> Я собирался прочесть массу других книг. В колледже я много читал, а однажды даже написал серию банальных и напыщенных статей для «Йельских новостей». И теперь я решил вернуться к этому занятию и стать самым ограниченным из всех специалистов, «всесторонне развитым человеком».
>
> *Фрэнсис Скотт Фицджеральд, «Великий Гэтсби»*

О чём эта книга?

Эта книга — о базах данных.

Не самоучитель, который позволит освоить программирование за 21 день, не сборник рецептов, не детальное описание какой-то конкретной платформы. Эта книга — обо всех базах данных сразу. Об алгоритмах, структурах данных и принципах проектирования, лежащих в основе всех современных платформ.

«Но позвольте, — скажет искушённый читатель, — о базах данных написана добрая тысяча книг. Зачем нужна тысяча первая?»

Для того, чтобы дать практическим навыкам прочный фундамент в виде понимания внутреннего устройства инструментов, с которыми вы привыкли работать.

Давным-давно знаменитый мастер боевых искусств после семинара для всех желающих провёл ещё один — закрытый, для «чёрных поясов». Среди приглашённых были опытные бойцы, инструкторы, даже чемпионы Европы.

Сенсей дал задание: «Встали в стойку и пошли в стойке до противоположной стенки, там удар и идём обратно». А сам принялся ходить между участниками и поправлять технику перемещения.

Минут через двадцать кто-то не выдержал и высказал мастеру своё недоумение — мол, это же семинар для «чёрных поясов», не пора ли от базовых упражнений для новичков перейти к чему-то серьёзному? Мастер пожал плечами, снова скомандовал всем встать в стойку и прошёл вдоль строя, толкая каждого в плечо или в грудь.

Чемпионы полетели на землю, как кегли, на ногах остался стоять только один боец.

И тогда мастер объяснил, что именно базовыми вещами спортсмены часто пренебрегают, стремясь поскорее перейти от скучных упражнений к «приёмчикам» и спарингам. И в итоге добираются до чёрного пояса, не умея как следует стоять. А если люди не умеют твёрдо стоять, они не могут ни нанести сильный удар, ни ловко маневрировать, ни эффективно защищаться. Вся их техника оказывается ущербной. А потом снова поставил учеников в стойку и заставил шагать до стенки и обратно.

Понимание устройства современных систем управления базами данных, алгоритмов и структур, на которых они базируются, подобно базовой технике в боевых искусствах.

Для кого эта книга?

Для всех, кто считает себя специалистом в области информационных технологий, прежде всего — для архитекторов информационных систем. Для тех, кто хочет войти в элиту.

Знаете ли вы, что изобретателем туалета считается сэр Джон Харингтон (Sir John Harington)[1], крестник королевы Елизаветы I? Конечно, в XVII веке туалет был предметом роскоши, доступным лишь королям и высшей знати. Позже он получил широкое распространение, и сегодня пользователями туалета является большинство населения Земли, а мастер, их устанавливающий, никак не ассоциируется у нас с рыцарским достоинством.

Но где-то существуют люди, проектирующие городские канализационные коллекторы или изобретающие новые материалы для труб и стандарты работы с ними. Это и есть настоящая инженерная элита.

Такой же путь проходит любая другая технология, включая и базы данных. Чуть больше полувека назад инженер, работающий с базой данных, казался если не волшебником, то как минимум посвящённым какой-то высокой ступени. Сегодня пользователем баз данных является каждый, у кого есть мобильный телефон, а для создания простых приложений, работающих с базами данных, достаточно прочесть тоненькую брошюрку или даже посмотреть на видео краткий курс лекций. Но для того, чтобы обеспечить эту кажущуюся простоту, нужны люди, на самом глубоком уровне понимающие, как эффективно хранить и обрабатывать данные. И это — тоже элита, системные программисты и архитекторы.

[1] www.thoughtco.com/who-invented-the-toilet-4059858.

С точки зрения программиста, «вошедшего в ИТ» после трёхмесячных курсов, архитектор — это некий мудрец в башне из слоновой кости, изрекающий прописные истины или, наоборот, грезящий о несбыточном. Однако если заглянуть в будущее чуть дальше, чем на год, то окажется, что «прописные истины» прописаны далеко не везде, а несбыточное неожиданно сбывается. В этой книге есть место и прописным истинам, и парадоксальным на первый взгляд утверждениям, которые заставят читателя задуматься и, возможно, пересмотреть отношение к некоторым «истинам». Вероятно, книга также изменит ваши представления о несбыточном и «сбыточном».

Кто же автор,

взявший на себя смелость писать такую книгу?

Такой же специалист, как и вы, — программист, администратор баз данных, архитектор. Специалист, для которого расхожее выражение о том, что знание нескольких закономерностей избавляет от необходимости помнить множество деталей, стало своеобразным профессиональным кредо, и которого это кредо ни разу не подводило.

Сегодня я хочу поделиться найденными закономерностями, ни в коей мере не посягая на ваше право узнать детали и разобраться в них лучше всех, став признанными экспертами.

Приятного чтения!

Благодарности

На обложке книги стоит единственное имя, но эта книга никогда бы не появилась на свет, если бы не помощь многих людей, которым я от души хочу сказать спасибо.

Директору Института системного программирования Российской академии наук **Арутюну Аветисяну** и замечательной команде программистов под его руководством — **Олегу Борисенко** и **Денису Турдакову**. В далёком 2015 году мы проводили большой совместный исследовательский проект, результаты которого оказались несколько неожиданными. Начав глубже разбираться в возникших вопросах, я обнаружил, что ответы на них непосредственно следуют из базовых принципов, на которых строятся все информационные системы. Именно тогда у меня возникла идея написать учебник, в котором эти принципы мог бы прочесть каждый желающий. А когда текст был готов, Олег внимательно его прочитал, с дотошностью настоящего учёного проверив все факты и утверждения.

Бизнес-тренеру, эксперту по партизанскому маркетингу **Александру Левитасу**. В самый разгар так называемой «пандемии» Александр организовал вебинар «Пишем книгу с Левитасом». Пусть я был не самым прилежным учеником, но без концентрированных знаний, полученных на этом вебинаре, книга вышла бы хуже и позже.

Моему коллеге и давнему другу **Егору Рогову**. Егора можно смело назвать соавтором этой книги. Именно он первым читал все написанные куски, и именно он исправил невероятное количество опечаток, оговорок, неточностей и неоднозначностей. Корректность формулировок и правильность языка — его заслуга. Кроме того, описывая устройство PostgreSQL, я всё время сверялся с его книгой «PostgreSQL изнутри».

Ведущему российскому эксперту в области баз данных **Константину Осипову**. Слушая его выступления, я понял, что выпускник факультета вычислительной математики и кибернетики МГУ имени Ломоносова просто обязан написать если не собственную платформу (а Константин создал Tarantool и Picodata, а также внёс значительный вклад в разработку MySQL и ScyllaDB), то хотя бы книгу о том, как устроены чужие платформы. Кроме того, благодаря замечаниям Константина эта книга превратилась, я надеюсь, из сухого перечисления фактов в рассказ не только о том, *как* устроены базы данных, но и о том, *почему* они так устроены.

Наталье Пивоваровой, которая руководила проектом внедрения корпоративного хранилища данных в компании «Вымпелком», одним из самых интересных проектов в моей карьере. Именно на этом проекте я почувствовал, что базы данных — это то, чему стоит посвятить свою жизнь. Сейчас Наталья Владимировна читает курс по базам данных в МГТУ имени Баумана, и именно она первой опробовала материал ещё не дописанной книги на студентах.

Компании «Постгрес Профессиональный» и лично **Олегу Бартунову**, её директору, за помощь в издании книги. Мы живём в мире, где информации слишком много, и не читатель ищет автора, а автор должен искать читателя. Многие книги, статьи, рассказы не нашли своего читателя потому, что усилий автора не хватило, чтобы пробиться через глухую стену равнодушия, коммерческой целесообразности и простых случайностей. То, что делает Олег и его команда для популяризации знаний, поистине бесценно.

Кроме того, хочу сказать спасибо моим коллегам, прочитавшим черновик книги и высказавшим множество бесценных замечаний: **Владимиру Харчикову** (Сбербанк-Технологии), **Александру Токареву** (Xsolla), **Игорю Мельникову**

(Постгрес Профессиональный), **Алексею Перегудову** (Инфосистемы Джет), **Андрею Кувалдину** (Сбербанк), **Валерию Марушеву** (ВТБ).

И отдельное спасибо моей дочери **Ольге**, которая не только вдохновляла меня на протяжении всей работы над книгой, но и нарисовала для неё обложку, вместившую символ понимания и мудрости — Ежа — и символ одной из лучших баз данных — Слона.

Часть I

Классификация баз данных

Глава 1
Модели данных

> Переход от неформального к формальному
> существенно неформален.
>
> *М. Р. Шура-Бура*

Модель данных — это набор абстракций и приёмов, с помощью которых мы пытаемся имитировать понятия реального мира. Модель данных напрямую ничего не говорит ни о производительности базы данных, ни о надёжности, ни о возможности масштабирования. Тем не менее именно модель в большинстве случаев является ключевым фактором выбора платформы.

1.1. Реляционные БД

Ещё в начале XXI века само понятие «база данных» подразумевало именно реляционную базу. Появившись в 70-х годах XX века, реляционные платформы до сих пор остаются наиболее распространёнными и универсальными. В рейтинге популярных баз данных, который ведёт портал db-engines.com, первые четыре места занимают именно реляционные СУБД — Oracle, MySQL, Microsoft SQL Server и PostgreSQL, а среди 50 самых популярных баз данных реляционных — больше половины.

Исторический экскурс

В 50-е и 60-е годы XX века главными покупателями ЭВМ были крупный бизнес и государственные учреждения, которые привыкли работать с каталогами и картотеками. Именно эти инструменты управления большими объёмами информации легли в основу первых навигационных моделей данных — иерархической и более поздней сетевой.

Основным объектом навигационных моделей был тип сегмента (segment type) — структура, описывающая некий объект реального мира: пациента, товар, договор. Тип сегмента состоял из полей (fields), содержавших данные, — например

фамилию или год рождения. Экземпляр сегмента (segment instance) описывал конкретный объект.

В иерархической модели у каждого типа сегмента был родительский тип — например, «родителем» каждого товара мог быть производитель. Хорошим примером иерархической модели может служить файловая система, которой мы пользуемся каждый день. Важным отличием сетевой модели от иерархической было то, что сегмент мог иметь несколько «родителей».

Для доступа к данным программист получал набор функций, работавших в терминах указателей на конкретные «карточки» — экземпляры сегментов. Типичными функциями были «найти сегмент и указать на первый экземпляр», «перейти к следующему экземпляру», «перейти к родительскому (или дочернему) экземпляру», «найти экземпляр, удовлетворяющий условию».

Навигационные СУБД прекрасно решали задачи, которые ставил перед ними бизнес, особенно с учётом ограниченной мощности компьютеров того времени. Яркими примерами таких платформ могут служить IMS (IBM) и ADABAS (Software AG). Несмотря на то что «золотым веком» навигационных платформ считаются 60-е годы прошлого столетия, они развивались вплоть до 80-х годов — достаточно вспомнить такие продукты, как dBase, FoxBase, Clipper.

Началом эры реляционных баз данных принято считать статью «Реляционная модель данных для больших совместно используемых банков данных» («A Relational Model of Data for Large Shared Data Banks»), опубликованную в 1970 году Эдгаром Коддом (Edgar Frank Codd).

С точки зрения самой структуры данных предложенная Коддом реляционная модель практически не отличалась от навигационной — все отличия касались способов описания структуры и доступа к данным. По мнению Кодда, главный недостаток существующих на тот момент систем — зависимость приложений от физической организации данных. В статье приводится три зависимости:

- Зависимость от порядка, в котором записаны данные. Если приложение предполагает, что данные записаны в определённом порядке (например, фамилии клиентов отсортированы по алфавиту), то нарушение этого порядка приведёт к ошибкам в работе приложения.
- Зависимость от индексации. Индексом называется вспомогательная структура данных, служащая для ускорения поиска. Если приложение рассчитано на использование какого-то индекса, то уничтожение этого индекса приведёт к неработоспособности приложения, и наоборот — использование нового индекса потребует изменения кода.

- Зависимость от путей доступа. Каждый сегмент хранится в некотором файле, и перемещение сегмента в другой файл или добавление в этот же файл нового сегмента может нарушить работу приложения.

Эдгар Кодд предложил строить описание модели и язык доступа к данным на базе математической теории, называемой реляционной алгеброй. Решение оказалось крайне успешным и обеспечило, пусть и не сразу, стремительный взлёт реляционных СУБД и их безусловное доминирование на протяжении многих лет.

Любопытно, что многие навигационные системы обзавелись реляционными фасадами, тем самым значительно продлив свою жизнь. Так, например, у ADABAS появились компоненты, позволяющие писать запросы к базе на языке SQL, из FoxBase вырос FoxPro, а Db2/i до сих пор, помимо SQL, поддерживает навигационный стиль программирования на языке RPG.

Реляционная алгебра

Слово «реляционная» произошло от слова «relation», отношение. Это математическая структура, определяющая взаимосвязи объектов. Рассмотрим, например, отношение «кратно». Это бинарное (или двухместное) отношение, то есть отношение, у которого два атрибута. Запись `{ делимое: 16, делитель: 4 }` говорит о том, что кортеж (то есть набор атрибутов) находится в отношении «кратно».

Атрибутов у отношения может быть сколько угодно — от одного и больше[1]. И вместо того чтобы формально определять отношения через математические функции, мы можем перечислить все кортежи, то есть наборы атрибутов, находящиеся в отношении. Например, если определить бинарное отношение «Квартет» с атрибутами «музыкант» и «инструмент», то кортеж будет в этом отношении, если в квартете играет «музыкант» на «инструменте»:

```
{
  "квартет": [
    {"музыкант":"мартышка", "инструмент":"скрипка"},
    {"музыкант":"осёл", "инструмент":"альт"},
    {"музыкант":"козёл", "инструмент":"скрипка"},
    {"музыкант":"мишка", "инструмент":"бас"}
  ]
}
```

Если вынести названия атрибутов в заголовок, получится привычная таблица:

[1] Теоретически возможно отношение без атрибутов, а некоторые платформы позволяют даже создать таблицу без колонок, но практического смысла такая структура лишена.

музыкант	инструмент
мартышка	скрипка
осёл	альт
козёл	скрипка
мишка	бас

Как правило, когда говорят о реляционных базах данных, вместо терминов теории множеств используют интуитивно понятные термины:

Русский термин теории множеств	Английский термин теории множеств	Русский интуитивный термин	Английский интуитивный термин
Отношение	Relation	Таблица Представление	Table View
Атрибут	Attribute	Столбец, колонка, поле	Column, field
Кортеж	Tuple	Строка, запись	Row, record

Несмотря на очевидные сходства, объекты реляционных баз данных отличаются от «идеальных» математических объектов:

- Строки в таблице упорядочены — как правило, по времени добавления в таблицу или по времени изменения, хотя существуют платформы, которые упорядочивают строки в порядке возрастания или убывания значений какого-либо столбца. При выборке порядок строк можно изменить. Кортежи в отношении не упорядочены, т. к. отношение — это множество.
- Столбцы (поля) в строке упорядочены; порядок столбцов в строке соответствует порядку их перечисления в команде создания таблицы. В некоторых платформах порядок перечисления столбцов может влиять на объём, занимаемый данными, и на скорость обработки. При выборке порядок столбцов можно изменить. Атрибуты в кортеже не упорядочены, т. к. кортеж — это тоже множество, как и отношение.
- В таблице может быть несколько одинаковых строк, в то время как все кортежи отношения (элементы множества) различны.
- В таблице могут быть «скрытые», или «системные», столбцы — например, временны́е метки или указатели на физическое расположение строки.

Реляционная алгебра определяет операции над отношениями, результаты которых также являются отношениями. Количество операций бесконечно, однако практические требования по обработке данных покрываются ограниченным набором реляционных операций:

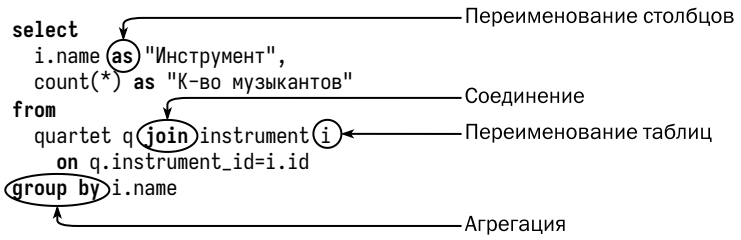
- переименование (rename) — изменение имён отношений и атрибутов (таблиц и колонок) в рамках текущей операции;
- проекция (projection) — выборка подмножества колонок[1];
- фильтрация (selection) — выборка подмножества строк, соответствующих заданному условию;
- соединение (join) — составление новых строк, состоящих из колонок соединяемых таблиц или представлений. Соединение — пожалуй, наиболее мощная и востребованная возможность реляционных баз данных. Теория множеств определяет соединение как «подмножество декартова произведения», но проще представить себе его как формирование новых кортежей, часть атрибутов которых берётся из первого отношения, а часть — из второго;
- объединение (union) — выборка, в которую попадают строки и из первой, и из второй объединяемых таблиц, причём строка, существующая в обеих таблицах, попадёт в выборку единственный раз. В отличие от «чистой» теории множеств, в реляционных БД есть и объединение без исключения дубликатов (union all);
- разность (complement) — выборка, в которую попадают строки, которые есть в первой таблице, но отсутствуют во второй таблице;
- пересечение (intersection) — выборка строк, которые есть и в первой, и во второй таблице;
- агрегация (aggregation) — расчёт «обобщённых» строк на базе значений нескольких строк исходной таблицы.

В реляционных платформах операции над данными записываются на языке SQL (structured query language). Аббревиатура «SQL» читается либо как «эс-кю-эль», либо как «сиквел» — в память о названии SEQUEL, под которым этот язык впервые увидел свет в СУБД IBM System R. Любопытно, что сегодня SQL считается языком программирования, хотя создавался он как язык конечного пользователя.

Пусть у нас есть таблица с участниками квартета, где музыкальный инструмент обозначен специальным кодом (ключом), а также таблица со справочником

[1] Здесь и далее будем использовать терминологию реляционных БД; описания останутся верными и в том случае, если заменить эти термины терминами теории множеств.

музыкальных инструментов. Надо получить названия инструментов и количество музыкантов, играющих на этих инструментах. Запрос, решающий описанную задачу, мог бы выглядеть так:



Описание языка SQL выходит за рамки данной книги, отметим лишь один фундаментальный принцип. SQL — язык декларативный, а не императивный. Это значит, что программа на языке SQL (обычно вместо слова «программа», program, используются термины «запрос», query, или «утверждение», statement) описывает результат, который программист хочет получить, в терминах операций над множествами (таблицами), но не описывает алгоритм получения этого результата.

За преобразование запроса в алгоритм отвечает компонент СУБД, называемый «оптимизатором» (optimizer), а сам процесс превращения запроса в алгоритм называется не компиляцией, а оптимизацией. Алгоритм, полученный в результате оптимизации, называется планом запроса.

Одни и те же операции могут быть выполнены несколькими способами и в разном порядке, то есть в зависимости от внешних условий СУБД может сгенерировать для одного и того же запроса совершенно разные планы.

Как и в случае с императивными языками, один и тот же результат может быть получен при помощи разных запросов. Изменение логики запроса может оказать гораздо более сильное влияние на производительность, чем выбор последовательности действий для реализации конкретного запроса.

Пусть, например, у нас есть таблица, каждая строка которой соответствует накладной. Колонками этой таблицы являются номер накладной (id) и идентификатор клиента (client_id). Задача состоит в том, чтобы выбрать клиентов, которые покупали товар несколько раз. Ниже приведены два решающих её запроса.

Первый запрос пытается каждой накладной при помощи операции соединения поставить в соответствие накладную с другим идентификатором, выписанную тому же покупателю, а потом из полученного списка пар накладных выбрать разных покупателей:

```
select
  distinct l1.client_id
from
  invoice i1 join invoice i2 on i1.client_id=i2.client_id
where l1.id<>l2.id
```

Второй запрос из всего списка накладных выбирает разных покупателей, а потом оставляет только тех, кто встретился два раза или больше:

```
select
  client_id
from invoice
group by client_id
having count(distinct id)>1
```

Скорее всего, второй запрос будет исполнен в несколько раз быстрее первого, независимо от качества оптимизации. Программисту, работающему с базами данных, следует научиться строить и понимать планы запросов. Подробнее о них поговорим в главе 4 «Обработка данных».

Ограничения целостности

Ещё один мощный инструмент, который реляционные базы данных предоставляют разработчику, — декларативные ограничения целостности, то есть ограничения, для соблюдения которых не нужно писать код — достаточно их описать.

Запрет неопределённых значений (not null) запрещает записывать в поле неопределённое значение, null. В некоторых источниках можно встретить описание null как «пустого значения», но это неверно. Пустая строка (`''`) — определённое значение, в отличие от null[1].

Первичный ключ (primary key) — это поле (или набор полей), которые уникально идентифицируют строку. Если какой-то набор полей объявлен как первичный ключ, то в ни в одном поле из этого набора не может быть неопределённых значений, и не может быть двух строк, в которых значения полей этого набора одинаковы.

Большинство реляционных платформ позволяют создавать таблицы без первичного ключа, однако почти у каждой таблицы первичный ключ всё же есть.

На практике в качестве первичных часто используются суррогатные ключи (surrogate keys). Суррогатом называется значение, не имеющее никакого смысла, единственное свойство которого — уникальность. Для создания таких значений

[1] Кроме СУБД Oracle, где пустая строка — то же самое, что и null.

в базах данных предусмотрены специальные механизмы — последовательности (sequences), поля с автоинкрементом (autoincrement columns) или генераторы глобально уникальных значений (uuid).

Любопытно, что одно и то же значение может быть как суррогатом, так и бизнес-ключом. Так, например, Пенсионный фонд России генерирует для каждого гражданина девятизначный[1] суррогатный ключ — СНИЛС (страховой номер индивидуального лицевого счёта). В то же время для любой базы данных, эксплуатируемой вне ПФ РФ, СНИЛС уже является бизнес-ключом, поскольку поступает в систему извне.

Споры между сторонниками суррогатных и натуральных (осмысленных) ключей не утихают. Основным недостатком суррогатных ключей считается усложнение структуры базы и увеличение пространства, занимаемого данными.

В то же время лишь суррогатный ключ может гарантировать наличие определённых значений и уникальность этих значений. Можно попытаться идентифицировать человека по полному имени, дате и месту рождения, но, во-первых, ссылаться на такой ключ неудобно, а во-вторых, дубликаты возможны. И даже такой идентификатор, как ИНН (индивидуальный номер налогоплательщика) физического лица, который должен быть уникальным и неизменным согласно Налоговому кодексу, может измениться — автору известен такой случай! Изменение первичного ключа — очень тяжёлая операция, а многие платформы такую операцию вообще не поддерживают.

Уникальный ключ (unique) — это поле или набор полей, в котором не может быть одинаковых значений в двух и более строках. В отличие от первичного ключа, уникальное поле может иметь неопределённое значение, если для него не задано ограничение not null. Ограничение уникальности хорошо подходит для полей, значения которых должны быть уникальны, но создаются вне приложения, — например, ИНН или СНИЛС.

До недавнего времени null не участвовал в определении уникальности, то есть в колонке, на которую наложено ограничение уникальности, могло содержаться сколько угодно неопределённых значений. Стандарт SQL:2023 добавил возможность считать уникальным значением и null тоже, и в колонке с таким ограничением может быть не больше одного неопределённого значения.

Проверка значения (check) ограничивает список значений, допустимых в поле, некоторым диапазоном. Например, можно проверять, что количество товара

[1] СНИЛС состоит из 11 цифр, но последние две цифры представляют собой контрольную сумму.

на складе неотрицательно или что в поле «пол» не записано ничего, кроме «M» и «F». В поле с проверкой может быть записано и неопределённое значение, null, если только на колонку не наложено дополнительное ограничение.

Внешний ключ (foreign key) — поле или набор полей, которые содержат первичный ключ другой таблицы. Таблица с внешним ключом называется дочерней (child), а таблица, на которую она ссылается, — родительской (parent). Таблица может ссылаться и сама на себя — например, в таблице «сотрудники» может быть ссылка на руководителя, который также является сотрудником и находится в той же таблице.

Если в базе данных объявлен внешний ключ, то база не позволит не только вставить в дочернюю таблицу строку, ссылающуюся на несуществующий ключ родительской таблицы, но и удалить строку из родительской таблицы, если на неё есть ссылки из дочерних таблиц. Возможны также варианты, когда при удалении строки из родительской таблицы все ссылающиеся на неё строки дочерних таблиц также удаляются (каскадное удаление), либо во все строки дочерней таблицы, ссылавшиеся на удалённую строку, в поля внешнего ключа записывается null (каскадное обнуление).

Вспомогательные структуры данных

Одним из ключевых достоинств реляционных платформ является независимость программ, работающих с базой данных, от физического представления данных. Эта особенность позволяет создавать вспомогательные структуры данных.

Эти структуры содержат подмножества данных, хранимых в таблицах, и их уничтожение не приводит к потере информации. Поскольку в большинстве баз данных чтение и поиск выполняются значительно чаще, чем запись, использование вспомогательных структур позволяет ускорить чтение ценой замедления записи и использования дополнительного пространства для хранения.

Создать вспомогательную структуру должен администратор БД, однако уже существуют системы, способные подсказывать, пусть и не всегда верно, какие именно вспомогательные структуры могут оказаться полезными.

К вспомогательным структурам относятся индексы (indexes) и снимки (snapshots, или materialized views).

Любой поиск в базе данных теоретически может быть выполнен путём полного просмотра таблицы. Но многие задачи поиска, особенно если требуется извлечь относительно небольшой объём данных, решаются значительно быстрее

за счёт индексирования. В **индексе** значения поля (или набора полей) упорядочены, а для поиска элемента в упорядоченном списке разработан ряд эффективных алгоритмов — например, двоичный поиск (binary search) или интерполяционный поиск (interpolation search).

Большинство индексов в реляционных БД построены с использованием B-деревьев, однако возможно и использование других структур. Индексы решают две важные задачи:

- ускорение доступа к данным;
- поддержка уникальности значений.

Если в таблице создан первичный ключ или ограничение уникальности, то каждый раз при добавлении строки БД должна просмотреть все значения полей, чтобы проверить, что вставляемое значение действительно уникально. Этот просмотр можно быстро выполнить только с использованием индекса, поэтому если ключевое поле в момент создания ограничения не проиндексировано, то индекс по ключевому полю будет создан автоматически.

Индекс по первичному ключу называется первичным (primary index), однако на таблице может быть создано произвольное количество вторичных индексов (secondary indexes) для ускорения поиска по неключевым атрибутам.

Представление (view) — это сохранённый запрос (select) на языке SQL. Представление может использоваться в запросах так же, как обычная таблица, но все операции, описанные в представлении, каждый раз при выполнении запроса, обращающегося к этому представлению, будут выполнены заново.

В отличие от «обычного» представления, **снимок (snapshot), или материализованное представление (materialized view)**, сохраняет результат исполнения запроса над базовыми таблицами и использует сохранённые данные точно так же, как данные базовых таблиц. Единственная разница между снимком и таблицей состоит в том, что для изменения данных в снимке необходимо изменить данные в таблицах, на базе которых этот снимок построен, а затем обновить снимок — автоматически или вручную.

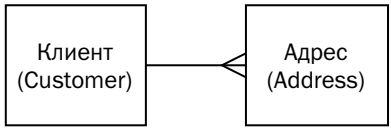
Диаграммы «сущность—связь»

Работа с реляционной БД начинается с проектирования схемы данных, то есть набора объектов и связей между ними, а также их отображения на таблицы и внешние ключи. Для проектирования на бумаге или в системах автоматизированного проектирования (ERWin, Rational Rose, Power Designer, ModelRight и др.)

используются диаграммы «сущность—связь» (entity-relationship diagram), где сущности (объекты) изображаются как блоки, а связи между ними — как линии между блоками Диаграммы используются и для проектирования физической модели данных, и тогда блоки обозначают таблицы, а линии — внешние ключи.

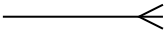
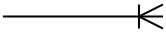
В большинстве случаев логическая модель совпадает с физической, то есть каждой сущности соответствует таблица, а каждой связи — внешний ключ. Однако существуют и исключения, например:

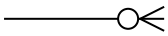
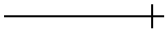
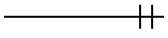
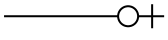
- в реляционной модели невозможно непосредственно создать связь «многие ко многим»;
- несколько сущностей (например, справочников) могут храниться в одной таблице;
- одна сущность может распадаться на несколько таблиц: часть атрибутов в одной таблице, часть — в другой;
- одна таблица может хранить текущее состояние объекта, а другая — историю состояний.



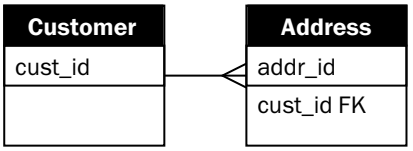
На диаграмме выше изображены две таблицы — клиент (customer) и адрес (address), причём таблица адресов ссылается на таблицу клиентов, то есть содержит внешний ключ. В данном случае логическая модель совпадает с физической.

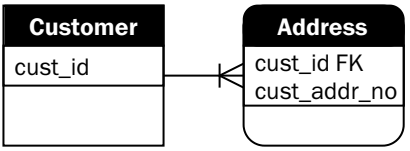
Существует несколько нотаций для диаграмм, однако наиболее распространённая — «вороньи лапки» (crow's foot). Для моделирования «на салфетке» достаточно линий, как на рисунке вверху — «лапка» цепляется к дочерней сущности, а линия без декораций — к родительской. Однако полная нотация включает в себя несколько вариантов окончания линий, указывающих на количество связей:

Изображение	Название	Пояснение
	Много	
	Один или много	В дочерней таблице должна быть минимум одна строка для каждого ключа родительской таблицы. В реляционных СУБД не существует декларативного правила для проверки этого условия

Изображение	Название	Пояснение
	Ноль или много	Допускаются строки в родительской таблице, на которые не ссылается ни одна строка дочерней таблицы
	Один	То же, что и окончание без декораций
	Один и только один	Внешний ключ не может быть пустым (поле с внешним ключом объявлено как not null)
	Ноль или один	Внешний ключ может быть пустым

Некоторые средства разработки (например, ERWin) могут также менять форму блока для таблицы: блок с закруглёнными углами означает наличие мигрирующего ключа.



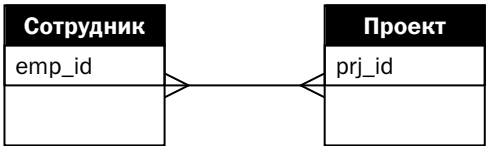


Пусть, например, в базе есть таблица «клиент» с суррогатным ключом. Для дочерней таблицы «адрес» мы можем также генерировать суррогатный ключ, а ключ клиента добавить обычным полем — внешним ключом. В этом случае диаграмма будет выглядеть как на рисунке слева.

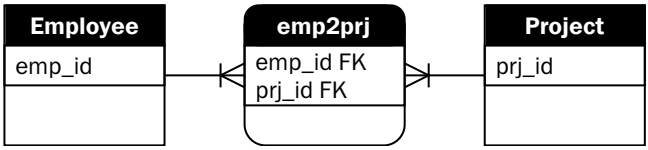
Другой способ идентификации адреса — идентификатор клиента и номер адреса для этого клиента, то есть внешний ключ является частью первичного ключа. В этом случае диаграмма будет выглядеть как на рисунке справа.

Классическая ситуация, когда без миграции ключей не обойтись, — вспомогательные[1] таблицы, реализующие связь «многие ко многим». Самый распространённый пример такой модели — сотрудники и проекты, когда в проекте работают несколько сотрудников, а сотрудник может участвовать в нескольких проектах одновременно. Логическая модель содержит два объекта и связь «многие ко многим»:

[1] В разговорной речи такие таблицы называют также «развязочными».



В физической модели появляется вспомогательная таблица с мигрировавшими ключами:



Автору встречались попытки ввести во вспомогательную таблицу собственный суррогатный первичный ключ и наложить на мигрировавшие ключи ограничение уникальности вместо ограничения первичного ключа, но такая структура лишь приводит к дополнительным расходам памяти и процессорного времени, не давая никакого выигрыша.

Нормальные формы

При моделировании реляционных данных сложились некоторые правила, призванные улучшить модель. Эти правила оформлены в виде требований и называются «нормальными формами» (normal forms), а приведение модели данных к нормальной форме называется «нормализацией».

Рассмотрим таблицу, в которой хранятся остатки на счетах клиентов банка:

Код клиента	Клиент
42	`{ "клиент": {"имя": "Василий","счёт": [ { "название": "моя карта", "код продукта": 800, "тип": "текущий счёт", "остаток": 10000 }, { "название": "кредит", "код продукта": 917, "тип": "автокредит", "остаток": -75200 } ] } }`
91	`{ "клиент": { "имя": "Анна", "счёт": [ { "название": "моя карта", "код продукта": 800, "тип": "текущий счёт", "остаток": 7000 }, { "название": "на чёрный день", "код продукта": 954, "тип": "вклад", "остаток": 30000 } ] } }`

Вопреки сложившемуся стереотипу, современные реляционные СУБД умеют не просто хранить большие тексты (CLOB — Character Large OBject), но и понимать структуру хранимых текстов (XML или JSON) и корректно работать с этой структурой. Однако в реляционной модели всё же принято хранить каждый атрибут отдельно.

Итак, говорят, что таблица (отношение) находится **в первой нормальной форме**, если все её колонки содержат атомарные значения, то есть такие значения, части которых не имеют значения сами по себе[1]. Значения ключа атомарны, т. к. сами по себе цифры «4» и «2» бессмысленны, а значения поля «Клиент» — нет.

После приведения к первой нормальной форме таблица примет вид:

Код клиента	Имя	Название	Код продукта	Тип	Остаток
42	Василий	моя карта	800	текущий счёт	10 000
42	Василий	кредит	917	автокредит	−75 200
91	Анна	моя карта	800	текущий счёт	7 000
91	Анна	на чёрный день	954	вклад	30 000

Очевидно, что поле «Код клиента» не является первичным ключом этой таблицы, т. к. его значение одинаково для всех строк, относящихся к одному клиенту. Первичным ключом будет набор полей «Код клиента, Название».

Работать с такой таблицей неудобно. Во-первых, нет простого способа извлечь из таблицы информацию о клиенте, не относящуюся к его счетам, — например, имя. Для этого придётся выполнять группировку по идентификатору клиента, и совершенно не очевидно, что делать, если в строках с одним и тем же идентификатором окажутся разные имена. Способов проследить, чтобы значения поля «имя» у всех строк с одинаковым идентификатором были одинаковыми, не существует. Во-вторых, возникают аномалии записи:

- если клиент решит сменить имя, то придётся обновлять столько строк, сколько у него счетов;
- невозможно сохранить информацию о клиенте, если у него нет ни одного счёта.

[1] Ценность частей значения — предмет разумных договорённостей. Теоретически можно как считать атомарным значением текст, содержащий структурированные данные, так и вкладывать отдельное значение в каждый разряд ключевого поля.

Для устранения этих аномалий таблица должна быть разбита на две другие таблицы, каждая из которых является проекцией исходной таблицы. Таким образом, процесс нормализации можно также назвать процессом проектирования.

Посмотрим, от чего зависит значение каждого поля:

Значения полей «код продукта», «тип» и «остаток» зависят от значения первичного ключа. Если не существует подмножества полей первичного ключа, определяющего значения этих полей, то такая зависимость называется неприводимой. Поле «имя» зависит только от части первичного ключа, что ведёт к аномалиям записи. Говорят, что таблица находится **во второй нормальной форме**, если значения всех колонок неприводимо зависят от первичного ключа.

Для приведения таблицы ко второй нормальной форме разобьём её на две. Первая таблица — «Клиент»:

Код клиента	Имя
42	Василий
91	Анна

И вторая — «Счёт»:

Код счёта	Код клиента	Название	Код продукта	Тип	Остаток
1001	42	моя карта	800	текущий счёт	10 000
1002	42	кредит	917	автокредит	–75 200
1003	91	моя карта	800	текущий счёт	7 000
1004	91	на чёрный день	954	вклад	30 000

Обратите внимание на два важных изменения:

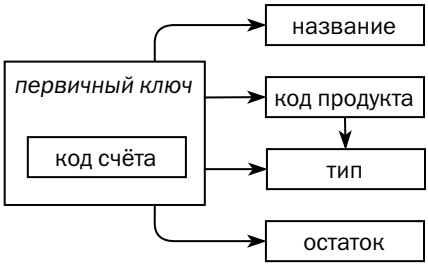
- в таблице «Счёт» сохранилось поле «Код клиента», и теперь оно является внешним ключом на таблицу «Клиент»;

- пара полей «Код клиента» и «Название» является уникальным ключом, как и в оригинальной таблице, но мы ввели дополнительное поле «Код счёта» с суррогатным ключом.

Проведя нормализацию, мы избавились от части аномалий записи, но аномалии по-прежнему остались:

- если банк расширит линейку продуктов и переименует «текущий счёт» в «счёт классический», придётся обновить столько строк, сколько «текущих счетов» открыто в банке;
- невозможно добавить информацию о продукте, не создав хотя бы один счёт;
- если будет закрыт последний счёт с продуктом, то информация о продукте пропадёт.

Зависимости в таблице «Счёт» выглядят так:



Значение первичного ключа однозначно определяет значения всех полей, то есть таблица находится во второй нормальной форме. Однако на самом деле значение поля «Тип» определяется значением поля «Код продукта», которое, в отличие от предыдущего примера, не является частью первичного ключа. Зависимость поля «Тип» от первичного ключа называется транзитивной, а если значения всех полей нетранзитивно зависят от первичного ключа, то говорят, что таблица находится **в третьей нормальной форме**. Для приведения таблицы к третьей нормальной форме разобьём её на две. Первая — «Счёт»:

Код счёта	Код клиента	Название	Код продукта	Остаток
1001	42	моя карта	800	10 000
1002	42	кредит	917	−75 200
1003	91	моя карта	800	7 000
1004	91	на чёрный день	954	30 000

И вторая — «Продукт»:

Код продукта	Тип
800	текущий счёт
917	автокредит
954	вклад

Обычно, когда говорят «база нормализована», имеют в виду, что все таблицы в ней приведены именно к третьей нормальной форме. Существуют и нормальные формы более высокого порядка, вплоть до шестой, где таблица состоит из первичного ключа и максимум одного неключевого поля. Однако большинство примеров таблиц, находящихся в третьей нормальной форме, но не в нормальной форме более высокого порядка, выглядят оторванными от практики. Поэтому описание нормальных форм более высокого порядка остаётся за рамками книги.

Хотелось бы отметить, что «нормальная форма» и «правильная модель» — не синонимы. Нормализация всего лишь избавляет нас от избыточности хранения и аномалий записи, но при этом ведёт к увеличению затрат на чтение данных из-за необходимости соединения таблиц.

В аналитических базах, где запись полностью контролируется, а объём чтения на несколько порядков превышает объём записи, нередко используются таблицы, находящиеся только во второй или даже только в первой нормальной форме. Существует несколько методологий проектирования хранилищ данных, которые по-разному разрешают компромисс между скоростью извлечения данных и компактностью хранения:

Методология	Русское название	Описание
Star schema	Схема-«звезда»	Таблицы-факты в третьей нормальной форме и таблицы-измерения в первой и второй нормальных формах
Snowflake schema	Схема-«снежинка»	Таблицы-факты и таблицы-измерения в третьей нормальной форме
Data vault	Свод данных	Таблицы-хабы в пятой нормальной форме, таблицы-связи в четвёртой нормальной форме и таблицы-сателлиты в любой нормальной форме — от первой до пятой
Anchor modeling	Якорное моделирование	Таблицы-якоря, атрибуты и узлы в шестой нормальной форме и таблицы-связи в четвёртой нормальной форме

1.2. Хранилища «ключ—значение»

Реляционные платформы практически безраздельно властвовали на рынке баз данных вплоть до начала 2000-х годов. Однако бурное развитие электронной коммерции и облачных вычислений потребовало появления более производительных, пусть и менее функциональных платформ для управления данными.

Основным требованием к новым платформам была возможность горизонтального масштабирования, то есть увеличения производительности базы данных не увеличением мощности единственного узла, а добавлением узлов.

Реляционные базы плохо поддаются горизонтальному масштабированию, поскольку, глядя на запрос на языке SQL, невозможно предположить, где находятся данные, которые потребуются этому запросу. Новое направление развития баз данных получило название NoSQL, и первыми NoSQL-платформами стали хранилища «ключ—значение». Одним из первых интерфейсов для работы с такими хранилищами является Java caching API, зарегистрированный в сообществе Java-разработчиков под номером JSR 107.

Java caching API

Интерфейс, появившийся в 2001 году, определяет следующий набор функций для работы с хранилищем:

- `containsKey(K)` — проверяет, есть ли в хранилище объект с ключом K;
- `get(K)` — возвращает объект с ключом K;
- `put(K,V)`– сохраняет объект V с ключом K;
- `putIfAbsent(K,V)` — сохраняет объект V с ключом K, только если в хранилище до этого не было объекта с таким ключом;
- `remove(K)` — удаляет из хранилища ключ K и ассоциированный с ним объект;
- `replace(K,V)` — сохраняет объект V с ключом K, только если в хранилище до этого уже был объект с таким ключом.

Это простейший набор операций, который в литературе часто называется CRUD (аббревиатура от «create, replace, update, delete»).

Кроме перечисленных простейших операций, интерфейс определяет ряд расширений:

- функции, изменяющие объект, связанный с ключом, и возвращающие предыдущее значение, — `getAndPut()`, `getAndRemove()`, `getAndReplace()`;

- функции, считывающие, записывающие или удаляющие сразу несколько ключей, — `getAll()`, `putAll()`, `removeAll()`;
- функции, обеспечивающие атомарные операции сравнения с обменом (compare-and-set), которые удаляют или заменяют объект только в том случае, если его значение совпадает с заданным, — `remove()`, `replace()` (в спецификации JSR 107 имена функций перегружены, и выбор нужной функции осуществляется по полному прототипу, включающему количество и тип параметров).

Хранилище не имеет никакого представления о структуре объекта, поэтому оно не может вернуть часть объекта или как-то обработать объект. Единственная возможность обработки объекта на стороне хранилища — методы `invoke()` и `invokeAll()`, позволяющие передать в хранилище класс вместе с кодом его методов и вызвать эти методы для объектов, ассоциированных с одним или несколькими ключами соответственно. Естественно, чтобы иметь возможность выполнить метод `invoke()`, само хранилище тоже должно быть написано на Java. На сайте сообщества Java-разработчиков приведён список платформ, реализующих спецификацию JSR 107: Oracle Coherence, Ehcache, Hazelcast, Infinispan, IBM WebSphere eXtreme Scale, Apache Ignite и др.

Логичным развитием хранилищ «ключ—значение» стало появление документоориентированных СУБД и семейств колонок.

Документоориентированные БД

Документоориентированная база данных — это хранилище «ключ—значение», где значение является не просто набором байтов, а **документом**, то есть в самом объекте содержится описание его структуры. Физически документ представляет собой текст в любом из структурированных форматов, например JSON или XML, или структурированные двоичные данные. Так, Couchbase и Firebase хранят данные в виде JSON, а MongoDB — в собственном двоичном формате BSON.

Никаких ограничений на структуру документов не существует, да и структура разных документов, хранящихся в одной коллекции, не обязана совпадать. Так, например, рядом могут храниться документы-клиенты, документы-счета и документы-продукты. Разумеется, на практике разработчики стараются хранить в каждой коллекции документы, соответствующие какой-либо одной сущности, но структура этих документов всё равно может отличаться. Например,

вместе хранятся клиенты — физические лица и клиенты — юридические лица, или все документы, которые компания отправляет клиенту, — накладные, счета и т. д.

Из-за свободы в части модели данных, которую документоориентированные платформы предоставляют программисту, данные в этих платформах иногда называют «слабоструктурированными» (semi-structured).

Отсутствие ограничений чрезвычайно удобно на этапе разработки, пока структура документов не устоялась, но доставляет немало проблем при эксплуатации, когда приложение получает документ незнакомой структуры. Поэтому современные документоориентированные платформы (MongoDB, Couchbase) позволяют задать схему документа, то есть список допустимых полей, а также их тип и диапазон значений.

Поскольку у базы данных есть доступ к структуре документа, она может возвращать не только документ целиком, но и обработанные данные — например, отдельные поля документа или даже агрегаты. Кроме того, база может индексировать атрибуты документа и выполнять по ним поиск.

Набор операций, выполняемых над хранимыми данными, отличается от платформы к платформе, и не существует формального критерия, позволяющего однозначно сказать, является платформа документоориентированной или ещё нет. Обычно документоориентированные СУБД умеют, помимо возврата документа (или набора документов) целиком по значению ключа, возвращать часть атрибутов документа (проекция), выбирать документы по значениям каких-либо полей (фильтрация), сортировать полученные данные и агрегировать их. Так же привычно, что документоориентированные СУБД не умеют выполнять операций над несколькими наборами результатов — пересечения (intersection), вычитания (complement), объединения (union) и, конечно же, соединения (join).

Вопреки распространённому заблуждению, для эффективной работы с документоориентированной БД модель данных всё же важна. Правда, в отличие от реляционной модели, подходы к проектированию основываются не на строгой математической теории, а на эмпирических правилах (rules of thumb). Так, например, MongoDB предлагает несколько вариантов реализации внешних ключей, причём выбор реализации зависит от кардинальности отношения, то есть от того, сколько дочерних объектов имеется у каждого родительского объекта.

Для отношений небольшой кардинальности, в пределах десятка дочерних записей, например, телефонов, предлагается включение. Родительский объект полностью содержит в себе дочерние объекты:

```
{
  "name": "Иванов Василий Петрович",
  "taxpayer_number": "123-456-7890",
  "phones": [
    { "phone_number": "+70001234567" },
    { "phone_number": "+70002345678" }
  ]
}
```

Этот подход позволяет одним запросом прочитать все дочерние объекты вместе с родительским, однако добавляет накладные расходы в ситуациях, когда нужны только родительские либо только дочерние структуры. Кроме того, при таком подходе невозможно выявить ситуацию, когда несколько клиентов указали один и тот же телефон.

Для отношений кардинальности от десятков до сотен записей предлагается включение идентификаторов. Например, если нужно сохранить список торговых точек на маршруте доставки, это можно сделать так:

```
{
  "_id": ObjectID("1A574E90"),
  "points_of_sale": [   // идентификаторы точек продаж
    ObjectID("DEADBEEF"), ObjectID("FEEDB0AA"),
    ObjectID("4455A896"), ...
  ]
}
```

При этом каждая запись о точке продаж выглядит, например, так:

```
{
  "_id": ObjectID("DEADBEEF"),
  "name": "Мясная лавка",
  "address": { "street_name": "Ленина ул.", "house": 33 }
  ...
}
```

В отличие от реляционной БД, где информация об экипаже и всех его точках извлекается одним запросом, документоориентированная БД при использовании такой структуры данных потребует двух запросов: первый извлечёт общую информацию о маршруте, включая список точек доставки, а второй — информацию обо всех точках по набору их идентификаторов.

Важно понимать, что на самом деле указанная структура реализует отношение не «один ко многим», а «многие ко многим», то есть через одну и ту же точку может проходить несколько маршрутов. В данном случае **контроль уникальности ссылки должно обеспечить приложение**.

Включение идентификаторов допускает и денормализацию — насколько вообще термин «нормальная форма» применим к нереляционным базам:

```
...
  "points_of_sale": [   // идентификаторы точек продаж
    { "_id": ObjectID("DEADBEEF"), "name": "Мясная лавка" },
    ...
  ]
...
```

Третий вариант структуры, который рекомендуется применять, когда дочерних записей действительно много,— это хранение ссылок — в точности то же, что внешние ключи в реляционных базах. Пусть, например, мы хотим сохранить в базе информацию обо всех маршрутах, по которым экипаж когда-либо проехал. Тогда запись о маршруте будет выглядеть так:

```
{
  "_id" : ObjectID("FF146ED7"),
  "crew_id": ObjectID("1A574E90"),
  "start_date": "2020-07-13 11:34:00 MSK"
  ...
}
```

В этом случае для работы с информацией о маршрутах конкретного экипажа потребуется вторичный индекс по полю `crew_id`. Надо только помнить, что документоориентированная БД не может контролировать корректность ссылок.

Разработчик может комбинировать подходы, — например хранить список дочерних объектов в родительском объекте и одновременно хранить в дочернем объекте ссылку на родительский объект. Структура данных регулируется исключительно соображениями здравого смысла.

Для работы с базой данных используется либо собственный API (Firebase), либо REST API, где для описания параметров запроса применяется JSON-документ (Amazon DynamoDB, MongoDB, CouchDB) либо диалект языка SQL (SQL++ в Couchbase). Стандарта на формат запроса не существует — запросы, работающие с одной базой, на другой базе выполнены не будут.

Отдельный подкласс документоориентированных баз данных представляют платформы для хранения текстовой информации — логов и текстов на естественном языке, например статей или постов в блогах. Эти платформы называются поисковыми движками (search engines). Помимо классических операций поиска объектов по идентификатору или значению неключевых атрибутов, они содержат мощные инструменты для полнотекстового поиска, то есть поиска с учётом морфологии, близости слов и других атрибутов текста на естественном языке. Наиболее известной библиотекой для полнотекстового поиска является Apache Lucene, на базе которой выпускаются такие продукты, как Elasticsearch, Solr, CrateDB. Также широко известны такие платформы, как Splunk и Sphinx.

Форматы хранения документов

Самый простой формат файлов для хранения данных — CSV, comma-separated values. Каждая строка такого файла представляет собой строку данных, а значения атрибутов разделены запятыми или каким-либо другим символом — табуляцией, символом конвейера «|» и др.

Все строки в CSV-файле имеют одинаковую структуру, как строки в таблице реляционной базы данных. Но есть и более сложные форматы, позволяющие хранить документы сложной структуры. Каждый формат обладает своим набором достоинств и недостатков, и для его оценки важны следующие характеристики:

- Способ кодирования — текстовый или двоичный. Текстовый формат легче для восприятия человеком, однако двоичный формат компактнее и требует меньше вычислительных ресурсов для интерпретации. Важно также помнить, что «человекочитаемость» текстовых форматов сохраняется лишь для относительно небольших документов, а применение специализированных средств просмотра и отладки позволяет читать двоичные документы.
- Наличие схемы в каждом документе даёт возможность хранить рядом документы совершенно разной структуры, но зато внешнее описание схемы позволяет существенно сократить объём данных.
- Поддержка сжатия позволяет извлекать из большого документа (или набора документов) только часть, не распаковывая все данные. Если сам по себе формат не поддерживает сжатие, документ может быть целиком сжат внешними средствами.
- Большинство форматов хранят документы, однако существуют и форматы, которые группируют вместе отдельные атрибуты документов. Документ такого формата намного сложнее в сохранении, зато поатрибутное, или колоночное, хранение даёт существенный выигрыш при аналитических запросах, где требуется обработать подмножество атрибутов всех документов.

Те же форматы, которые используются для хранения данных, могут использоваться и для передачи данных — например, при удалённом вызове процедур.

В таблице ниже собраны наиболее популярные форматы:

Формат	Кодирование	Схема	Сжатие	Поатрибутное хранение
XML	текстовое	в документе	нет	нет
JSON	текстовое	в документе	нет	нет

Формат	Кодирование	Схема	Сжатие	Поатрибутное хранение
BSON	двоичное	в документе	нет	нет
MsgPack	двоичное	в документе	нет	нет
Parquet	двоичное	в файле	да	да
Avro	текстовое, двоичное	в файле	нет	нет
Protobuf	двоичное	вне документа	нет	нет
Thrift	двоичное	вне документа	нет	нет

Исторически первый стандарт **XML (eXtended Markup Language)**, опубликованный в 1998 году, является и самым сложным. XML-документ представляет собой текстовый файл, элементы которого размечены тегами — конструкциями, заключёнными в угловые скобки:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<person class="physical">
  <name>Иванов Василий Петрович</name>
  <contacts>
    <phone type="mobile">+79995551234</phone>
    <phone type="office">+78122128506</phone>
  </contacts>
</person>
```

Информация может содержаться как в самих элементах (например, «Иванов Василий Петрович»), так и в атрибутах тега (например, атрибут `type` тега `phone`, определяющий тип телефона).

К важнейшим достоинствам XML можно отнести:

- возможность хранения практически любой слабоструктурированной информации — например, договоров или публикаций;
- возможность ставить ссылки между документами;
- возможность описания схемы документа.

Схема XML-документа (XSD — XML Schema Definition) тоже записывается на языке XML и содержит названия элементов и атрибутов, отношения между элементами и типы данных.

Для иллюстрации приведём одну из возможных схем документа, рассмотренного выше:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="person">
    <xs:complexType>
      <xs:sequence>
        <xs:element type="xs:string" name="name" />
        <xs:element name="contacts">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="phone" maxOccurs="5" minOccurs="0">
                <xs:complexType>
                  <xs:simpleContent>
                    <xs:extension base="xs:string">
                      <xs:attribute name="type" use="optional">
                        <xs:simpleType>
                          <xs:restriction base="xs:string">
                            <xs:enumeration value="mobile" />
                            <xs:enumeration value="office" />
                            <xs:enumeration value="home" />
                          </xs:restriction>
                        </xs:simpleType>
                      </xs:attribute>
                    </xs:extension>
                  </xs:simpleContent>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute type="xs:string" name="class" use="required" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Очевидный недостаток XML — его большой объём и огромные затраты вычислительных ресурсов на разбор.

Современные приложения в основном используют XML как транспортный формат и хранят в базе данных не только образ исходного документа (в основном для разбора инцидентов), но и атомарные значения отдельных полей.

Многие реляционные СУБД, такие как Oracle, PostgreSQL, Db2, Microsoft SQL Server, позволяют хранить документы в XML и обрабатывать их непосредственно в базе данных.

Для приложений, основной задачей которых является хранение текстовых документов с относительно небольшим объёмом метаданных (например, база данных научных публикаций или судебных решений), существует подкласс документоориентированных платформ, использующих XML как основной формат хранения данных, — Native XML Databases. К ним относятся MarkLogic, Virtuoso, Oracle

Berkeley DB XML и другие[1]. Для обработки XML-документов существует мощный язык запросов XQuery.

Стандарт **JSON (JavaScript Object Notation)** практически вытеснил XML из целого ряда традиционных ниш. Этому способствовали относительная простота и компактность формата по сравнению с XML.

Как следует из названия, спецификация JSON — это подмножество синтаксиса языка JavaScript (точнее, его подмножества ECMAScript). Впервые JSON появился именно в веб-приложениях, потому что для разбора JSON достаточно передать его интерпретатору JavaScript. Сегодня библиотеки для работы с JSON есть во всех популярных языках программирования.

Объект, описанный выше в XML, в JSON выглядел бы так:

```
{
  "person": {
    "class": "physical",
    "name": "Иванов Василий Петрович",
    "contacts": {
      "phones": [
        { "type": "mobile", "number": "+79995551234" },
        { "type": "office", "number": "+78122128506" }
      ]
    }
  }
}
```

Размеры представлений этого конкретного объекта примерно равны, однако практика показывает, что JSON в среднем на 30 % компактнее.

С распространением JSON также появилась возможность описывать схемы JSON-документов и автоматически проверять соответствие документа схеме. В настоящее время стандарт JSON-схемы имеет статус черновика, но достаточно широко используется на практике.

Современные реляционные СУБД позволяют хранить документы в форматах JSON и обрабатывать их внутри БД.

В отличие от XQuery для XML, сегодня не существует общепризнанного стандарта для языка запросов к JSON-документам. Разные платформы используют не только разные языки, но и разную семантику сравнения данных. Группой экспертов, участвовавших в разработке XQuery, предпринята попытка создания

[1] Большинство таких платформ поддерживают и другие парадигмы хранения, так что правильнее назвать их не «XML-платформами», а многовариантными. Подробнее о многовариантности — далее в этом разделе.

языка JSONiq (jsoniq.org), претендующего на статус стандарта. Описание языка доступно под лицензией CC BY-SA[1]. Существует несколько движков для выполнения запросов на языке JSONiq, в том числе с открытым исходным кодом.

Формат **BSON (Binary JSON)** был разработан компанией MongoDB для хранения данных в своей СУБД. BSON очень похож на JSON, однако имеет ряд важных отличий:

- BSON — двоичный формат: в нём не используются отступы для удобочитаемости и кавычки для экранирования строк. Вместо этого строки завершаются нулевым байтом и дополнительно указывается их длина. Всё это позволяет уменьшить пространство, занимаемое объектом, и ускорить его обработку.
- Значения в BSON имеют тип, в отличие от JSON, где тип значения определяется исключительно его написанием (в кавычках — строка, без кавычек — число или логическое значение). BSON поддерживает типы «дата» и «набор байтов» (binary), которых нет в JSON, а также позволяет конкретизировать тип числа — integer, float, long и т. д.

Спецификация BSON опубликована, однако статуса стандарта у неё нет — фактически, это внутренний формат MongoDB. Тем не менее существуют библиотеки для многих популярных языков программирования, позволяющие работать с этим форматом. Сама MongoDB не предоставляет прямого доступа к своим данным как к BSON-объектам — вместо этого она предоставляет API, позволяющий манипулировать полями объектов. Кроме того, у этой СУБД есть утилиты, позволяющие выгружать и загружать данные в формате BSON, либо в JSON с некоторыми фирменными расширениями, обеспечивающими более строгую типизацию данных.

У PostgreSQL есть также формат JSONB, который не совместим с BSON, но зато обратно совместим с JSON. JSONB не является открытым стандартом, однако позволяет эффективно хранить и обрабатывать JSON-документы. К особенностям этого формата относятся сжатие незначащих пробелов, сортировка атрибутов и удаление дубликатов, хранение в разобранном виде с возможностью индексирования. Кроме того, компанией Postgres Professional разработано расширение zson, выполняющее словарное сжатие JSONB.

[1] Лицензия Creative Commons Attribution-ShareAlike позволяет другим перерабатывать, исправлять и развивать произведение даже в коммерческих целях при условии указания авторства и лицензирования производных работ на аналогичных условиях.

Формат **MsgPack (Message Pack)** также является заменой JSON и позволяет хранить документы в двоичном представлении. В отличие от BSON, MsgPack является открытым стандартом и используется, например, в Redis и Tarantool/Picodata.

Помимо двоичного кодирования, важным отличием MsgPack от JSON являются так называемые «паскалевские» строки и массивы: вместо того, чтобы определять конец строки или массива по нулевому символу или элементу, в MsgPack указывается длина строки или количество элементов массива. Такой подход сложнее в кодировании (сериализации), но проще в декодировании.

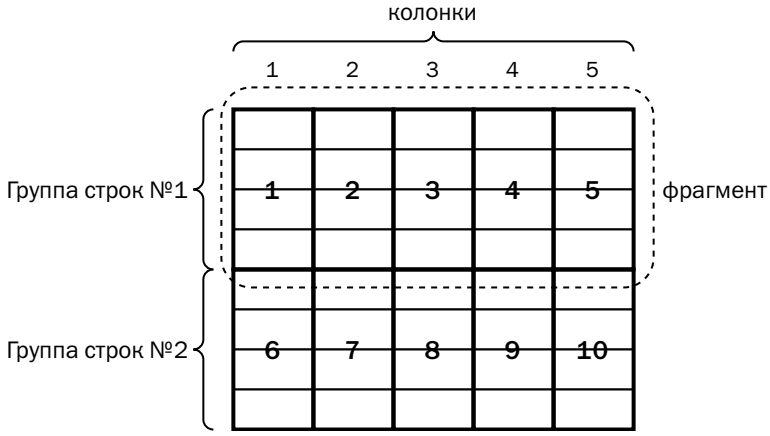
Представление данных в JSON, BSON или MsgPack хоть и компактно по сравнению с XML, всё же достаточно объёмно за счёт хранения схемы, то есть имён (а в BSON — и типов) атрибутов в каждом документе. Для экономии места и ускорения обработки применяются форматы, где схема данных хранится отдельно от самих данных.

Формат **Apache Avro** разработан в рамках экосистемы Hadoop и применяется как транспортный формат при удалённом вызове процедур и как формат для промежуточного хранилища необработанных данных («озеро данных», data lake). В Avro, как и в MsgPack, при сериализации записывается длина значения, а за ним — само значение. Порядок, название и типы полей описываются схемой, которая представляет собой документ в формате JSON или в специализированном формате Avro IDL.

При передаче большого объёма данных схема хранится в заголовке файла, за которым следует произвольное количество блоков данных. База данных может хранить каждую строку со своей схемой. При этом в начале записи помещается идентификатор схемы, а сами схемы хранятся отдельно от данных. Если Avro используется в качестве транспортного формата при удалённом вызове процедур, клиент и сервер могут договориться о схеме заранее.

Кроме двоичного кодирования, Avro поддерживает также кодирование в JSON, но эта возможность используется главным образом при отладке.

Формат **Apache Parquet** также разработан в экосистеме Hadoop и активно используется в Apache Spark. В отличие от всех рассмотренных ранее форматов, Parquet хранит данные не по строкам, а по колонкам. Вся таблица делится на группы строк (row groups), и в файл Parquet сохраняются сначала все значения одной колонки для группы строк, затем все значения другой колонки и т. д. Набор значений колонки, относящихся к одной группе строк, называется фрагментом (chunk). Каждый фрагмент описывается своими метаданными — тип данных, количество значений, алгоритм сериализации, сжатый и несжатый размеры.



Такой формат хранения имеет массу преимуществ перед традиционными:

- при выполнении аналитических запросов можно читать не весь файл, а только те колонки, которые нужны для выполнения запроса;
- благодаря разбиению данных на фрагменты чтение хорошо распараллеливается — в частности, набор данных можно обрабатывать одновременно на нескольких узлах;
- Parquet — наиболее компактный формат, поскольку для каждой колонки может быть выбран свой формат сериализации, зависящий от типа данных; кроме того, значения в колонке, как правило, похожи друг на друга и благодаря этому хорошо сжимаются.

Недостатки формата также очевидны:

- формат не поддерживает разные схемы — скорее, структура файла близка к таблице реляционной БД;
- единственная возможность изменения схемы — добавление колонки;
- любое изменение данных требует перезаписи файла, поэтому формат Parquet эффективен для аналитических систем, но совершенно непригоден для часто изменяемых данных.

Ещё два формата, заслуживающих внимания, — **Protobuf** и **Apache Thrift**. Первый разработан и поддерживается Google, второй разработан Facebook[1] и передан в Apache Software Foundation. Протоколы различаются некоторыми деталями — наборами поддерживаемых типов и языков программирования, возможностями

[1] Социальная сеть Facebook принадлежит компании Meta, признанной в России экстремистской организацией. Здесь и далее торговая марка Facebook употребляется исключительно в контексте созданных компанией технологий.

расширения и лицензией. В обоих форматах используется двоичная сериализация (Thrift в отладочных целях допускает также сериализацию в JSON), где данные состоят из тега (идентификатора) поля, его типа и значения. Структура сообщения описывается при помощи специальных языков (IDL), а затем транслируется в код, формирующий и читающий сообщения. Основное назначение обоих форматов — передача параметров при удалённом вызове процедур; для постоянного хранения данных эти форматы не используются.

Хранилища семейств колонок

Другой путь расширения возможностей хранилищ «ключ—значение» привёл к появлению ещё одного класса СУБД. Идея, впервые реализованная в СУБД Google BigTable, заключается в том, что значение, соответствующее ключу, представляет собой множество пар «ключ—значение» или «название колонки — значение колонки». Такие хранилища назвали wide column store, то есть широкое хранилище колонок. Как и следовало ожидать, в таких хранилищах стало появляться много ключей с одинаковыми наборами значений, и для повышения производительности описание наборов значений стали хранить отдельно от значений и назвали семействами колонок (column families). В англоязычной литературе термины wide column store и column family store используются как синонимы, а в русском языке закрепился более поздний термин «хранилища семейств колонок».

Наиболее известные платформы этого типа — Google BigTable, Apache HBase, Apache Cassandra и её коммерческая версия DataStax Enterprise, а также ScyllaDB.

Чтобы начать работу с хранилищем семейств колонок, нужно создать пространство ключей (keyspace — в Cassandra, table — в HBase) — аналог базы данных в реляционных БД. Пространство определяет параметры хранения данных, — такие как количество реплик каждого элемента и набор серверов, где они будут храниться. В пространстве создаются семейства колонок (column families) — наборы колонок, каждая из которых имеет определённый тип.

Легче всего представить себе семейство колонок как электронную таблицу — например, книгу Microsoft Excel®. Пространство соответствует книге, а семейство колонок — листу. Лист очень похож на таблицу — настолько, что в СУБД Cassandra с 2011 года используется термин table вместо термина column family. Если в HBase используется Java API и REST API, основанный на JSON, то в платформах Cassandra и ScyllaDB используется язык CQL (Cassandra Query Language), очень похожий на язык SQL. Тем не менее хранилище семейств колонок не является реляционной базой данных:

- В отличие от таблицы в реляционной базе, семейство колонок обязано иметь первичный ключ. Эта модель представления данных проектировалась с прицелом на распределённое хранение, и первичный ключ (точнее, его часть) определяет узлы, где будут храниться соответствующие ему данные.
- Как и в документоориентированных БД, хранилище семейств колонок не предусматривает операций над несколькими таблицами, поэтому для хранения сложных структур данных широко используются композитные типы — множества, списки, ассоциативные массивы. Если использовать терминологию реляционных БД, то семейство колонок не находится в первой нормальной форме.
- Данные в хранилище упорядочены. Если в реляционных БД основным инструментом ускорения доступа к данным является построение вторичных индексов, то в семействах колонок в первую очередь нужно определить именно порядок строк.
- В отличие от реляционных БД, колонки в семействе упорядочены. Это отличие скорее теоретическое, потому что, несмотря на отсутствие порядка атрибутов в отношении, колонки в таблице реальной реляционной БД обычно располагаются в порядке их перечисления в команде `CREATE TABLE`.

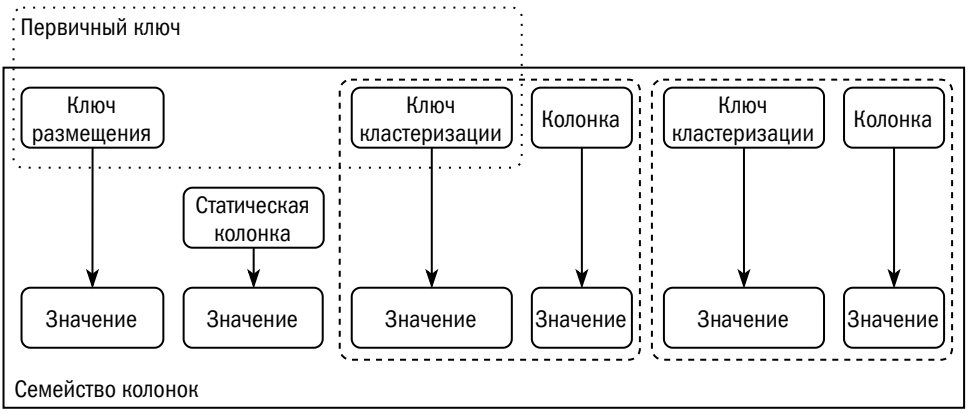
Если в HBase первичный ключ представляет собой набор байтов, который приложение должно как-то интерпретировать, то в таких платформах, как Cassandra и ScyllaDB, использующих CQL, первичный ключ — это типизированная колонка или набор колонок.

Ключ из единственного поля ничем не отличается от обычного ключа в «чистой» платформе класса «ключ—значение» или в документоориентированной платформе. Но если ключ составной, то разные части ключа выполняют разные функции.

Первое поле (или несколько полей) составляет ключ размещения (partitioning key). Платформа гарантирует, что все данные, у которых значения этого ключа совпадают, будут храниться на одном узле[1]. Остальные поля, входящие в первичный ключ, составляют ключ кластеризации (clustering key). Данные упорядочены в соответствии со значениями ключа кластеризации.

При использовании составного ключа у семейств колонок появляется возможность использовать статические колонки (static columns), значение которых

[1] Корректнее будет сказать «попадут в один шард». Подробнее о шардировании — в главе 3 «Структуры хранения данных».



одинаково для всех значений ключа размещения. Когда ключ состоит из единственного поля, все колонки можно считать статическими. По сути семейство колонок представляет собой соединение двух таблиц, одна из которых подчинена другой (master-detail).

Пусть, например, у нас есть база данных торговой компании, и в этой базе хранятся накладные. Идентификатором (ключом) накладной может служить набор (идентификатор клиента; номер накладной), причём идентификатор клиента будет ключом размещения, а номер накладной — ключом кластеризации. Контактная информация клиента может храниться в статических колонках.

Языки запросов к семействам колонок предоставляют те же возможности, что и интерфейсы документоориентированных БД, — возврат части атрибутов, поиск по значениям неключевых атрибутов и поддержка вторичных индексов. Так же, как и документоориентированные платформы, семейства колонок не поддерживают операции с несколькими наборами данных — соединения, объединения, пересечения и др.

БД временны́х рядов

Ещё одна ветвь эволюции хранилищ «ключ—значение» — базы данных временны́х рядов (time series databases). Это специализированные платформы, предназначенные для хранения событий — например, данных системы мониторинга или результатов опроса датчиков. У таких наборов данных и запросов к ним есть особенности, которые учитываются при разработке:

- Ключ может содержать несколько полей — например, номер объекта и номер датчика. Как и в семействах колонок, пользователь имеет возможность

на уровне описания данных указать, какие из этих полей влияют на место хранения событий (записей).
- Ключ обязательно содержит временну́ю метку.
- Данные делятся на два типа: собственно метрики событий и дополнительная информация о ключе — метки (tags). Например, вместе с номером объекта мы можем хранить его адрес, чтобы впоследствии анализировать информацию не по одному объекту, а по группе соседних объектов. Для выполнения таких запросов платформа предоставляет возможность индексирования меток.
- В целях экономии ресурсов данные со временем будут агрегированы. Следовательно, в качестве измерений необходимо уметь хранить не только скалярные значения, но и разного рода агрегаты — от количества измерений и среднего значения до гистограмм.
- Для БД временны́х рядов характерны запросы истории изменения набора параметров в течение заданного времени с определённым шагом, не обязательно совпадающим с шагом измерения значений. Следовательно, платформа должна быстро и эффективно выполнять пересчёт и сопоставление временны́х меток, агрегацию и ранжирование показателей. И, разумеется, предоставлять удобный интерфейс для этих операций.

Кроме того, есть и другие особенности работы этих платформ, влияющие не столько на модель данных, сколько на механизм хранения. Например, данные очень быстро записываются, но никогда не обновляются, а по истечении некоторого времени все данные со старыми временны́ми метками удаляются.

Наиболее известные платформы этого класса — InfluxDB, Prometheus, Graphite, TimescaleDB, OpenTSDB.

1.3. Другие модели данных

Объектные БД

В основе объектных баз данных лежит идея построить платформу для хранения и обработки данных на принципах объектно-ориентированного программирования.

Объектная модель имеет много общего с реляционной, и между терминами объектной и реляционной моделей есть прямое соответствие:

В объектной модели (английский термин)	В объектной модели (русский термин)	В реляционной модели (английский термин)	В реляционной модели (русский термин)
Package	Пакет	Schema	Схема
Class	Класс	Table	Таблица
Object instance	Объект (экземпляр класса)	Row	Строка (запись)
Property	Свойство	Column	Колонка (поле)
Relationship	Отношение (ссылка)	Foreign key	Внешний ключ
Embedded object	Встроенный объект	Multiple columns	Набор колонок
Method	Метод	Stored procedure	Хранимая процедура

По замыслу создателей объектных БД объектный интерфейс должен был упростить интерфейс между базой данных и приложением за счёт отказа от библиотек, представляющих реляционные данные как объекты (ORM, object-relationnal mapping), а кроме того — повысить производительность выполнения запросов за счёт унификации моделей данных при обработке (в приложении) и хранении (в базе).

На практике широкого распространения объектные базы данных не получили: в первой сотне рейтинга, рассчитываемого сайтом db-engines.com, находится всего одна такая платформа — InterSystems Caché. Библиотеки же класса ORM, наоборот, используются весьма широко: для Java существуют Hibernate, EclipseLink и более десятка менее распространённых библиотек; для .NET, кроме библиотеки Entity Framework, включенной в .NET Framework, существует более десятка библиотек, поддерживаемых независимыми разработчиками. Существуют также ORM-библиотеки для PHP (включая те, что входят в состав Yii и Zend Framework), для Python (включая ту, что входит в Django), для Perl, Ruby и других языков.

Одна из главных проблем объектной модели состоит в том, что **инкапсуляция,** один из принципов объектной парадигмы, подразумевает, что приложение вместо полного доступа к структуре данных может выполнять только набор разрешённых операций над объектами. А это в свою очередь подразумевает активное использование хранимого кода в БД или разработку для каждого языка про-

граммирования библиотек, учитывающих особенности объектной модели языка и совмещающих модель языка с моделью конкретной базы данных.

Представляет интерес история публикаций, посвящённых объектным платформам. В 1989 году на конференции в Киото группой учёных под руководством профессора Школы информатики Эдинбургского университета Малколма Аткинсона (Malcolm Atkinson) был сделан доклад под названием «Манифест систем объектно-ориентированных баз данных» («The Object-Oriented Database System Manifesto»). Но уже в 1990 году был опубликован «Манифест СУБД третьего поколения» («Third-Generation Data Base System Manifesto»), где одним из требований к новым платформам было сохранение всех возможностей старых, то есть реляционных, которые отнесены ко второму поколению. Второй манифест был написан группой исследователей под руководством великого Майкла Стоунбрейкера (Michael Stonebraker). Его можно считать реакцией индустрии на революционные предложения первого манифеста. Вкратце, по мнению авторов второго манифеста, все новые возможности можно получить, не производя революцию в области технологии баз данных, а эволюционно развивая традиционные реляционные платформы.

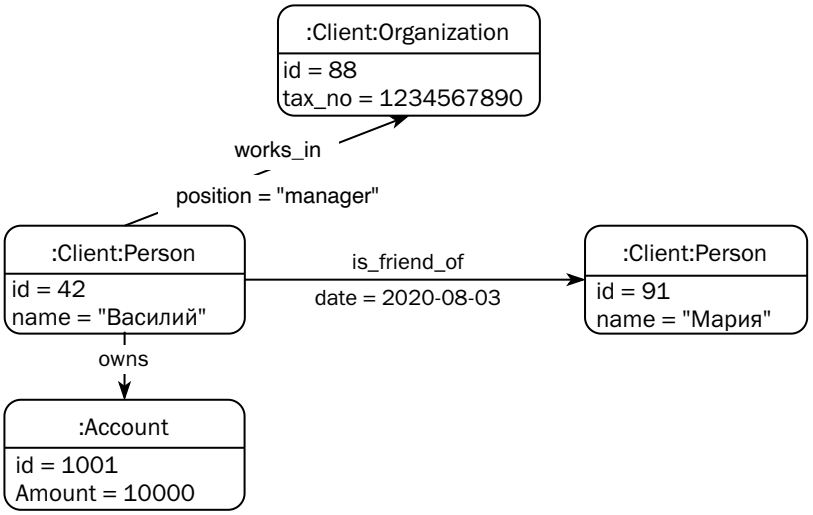
Графовые БД

Графовые базы данных — прямые наследники сетевых баз. Данные здесь представлены в виде графов: объекты являются узлами, а связи между ними — рёбрами. Графовые БД, как и реляционные, имеют развитые средства декларативного описания предметной области.

Каждый **узел** (объект) может иметь метки. Например, на рисунке ниже все узлы, обозначающие клиентов, имеют метку :Client и дополнительную метку, :Person или :Organization, определяющую тип клиента. Метки не фиксированы, их можно ставить и убирать. Так, например, если обслуживание клиента приостановлено, ему можно поставить метку :Suspended, а при возобновлении обслуживания метку можно будет убрать. Метки нужны для выделения классов узлов.

Между объектами существуют **отношения**[1] (рёбра). У отношения обязательно есть направление и название. На рисунке ниже мы видим отношения «владеет» (owns), «дружит_с» (is_friend_of), «работает_в» (works_in). Запросы могут

[1] Термин «отношения» в графовых СУБД не имеет ничего общего с отношениями в реляционных платформах. Но с точки зрения теории множеств множество рёбер в графовой базе является отношением третьей (или больше) степени.



учитывать или не учитывать направление отношения: например, мы всегда считаем Василия другом Марии, но в зависимости от логики запроса можем считать или не считать Марию подругой Василия.

У узлов и отношений могут быть **атрибуты** — наборы пар «ключ—значение», представляющие данные, связанные с узлом или отношением.

Графовые БД, как и реляционные, позволяют определить декларативные ограничения целостности:

- **Существование** атрибута. Атрибут с заданным именем должен существовать у всех узлов с заданной меткой или у всех отношений (рёбер) заданного типа.
- **Уникальность** атрибута. Значение атрибута должно быть уникальным среди всех узлов с заданной меткой. При этом атрибут не обязан существовать.
- **Первичный ключ** узла. Атрибут (или набор атрибутов) узла обязан существовать, а его значения (или наборы значений) обязаны быть уникальными.

Графовые базы позволяют создавать индексы по атрибутам узлов.

В основном графовые базы используются для разного рода аналитики, в том числе в реальном времени — анализа социальных сетей, рекомендации товаров и контента, предотвращения мошеннических операций, поддержки баз знаний.

Для написания запросов к графовым базам созданы декларативные языки. В отличие от SQL, который де-юре и де-факто является стандартом для реляционных платформ, для графовых баз создано несколько языков, отличающихся друг от друга синтаксисом и набором возможностей:

Язык	В каких платформах используется
Cypher	Neo4j, Agens Graph, AnzoGraph, CAPS: Cypher for Apache Spark, Memgraph
Gremlin	Neo4j, Microsoft Azure CosmosDB, OrientDB, JanusGraph, Amazon Neptune, Giraph, InfiniteGraph
SPARQL	Amazon Neptune, Virtuoso, AllegroGraph, Blazegraph

В 2019 году комитет ISO/IEC принял предложение о создании стандарта GQL — языка запросов к графовым БД, во многом основанном на Cypher.

Важное отличие графовых БД от реляционных состоит в том, что результатом выполнения запроса не обязательно является граф: это может быть и граф, и набор узлов, и таблица с набором атрибутов.

1.4. Сравнение моделей данных

После рассмотрения такого множества подходов к представлению данных возникает вопрос: какая же модель лучше? В таблице ниже приведены ключевые особенности рассмотренных моделей:

	Реляционная	Ключ—значение	Документная	Семейство колонок	Временны́е ряды	Объектная	Графовая
Поиск по ключу	+	+	+	+	+	+	+
Хранение слабоструктурированных данных	+	+	+	+	–	–	–
Извлечение или обновление части атрибутов	+	–	+	+	+	+	+
Поиск по неключевым атрибутам	+	–	+	+	–	+	+
Составной ключ	+	–	–	+	+	+	+
Связи между сущностями	+	–	–	–	–	+	+
Стандартный интерфейс	+	+	–	–	–	–	±

Из таблицы очевидно, что с точки зрения возможностей манипулирования данными реляционная модель — наиболее универсальная среди всех рассмотренных. Специализированные возможности, которых в реляционных СУБД нет «из коробки», могут быть реализованы при помощи языка SQL или его процедурных расширений, которые есть во всех современных платформах. Так, например, базы временны́х рядов в тестах регулярно сравниваются с реляционными и не всегда побеждают даже на тех специфических задачах, для которых изначально созданы.

Важным преимуществом реляционной модели является компактность хранения: структура таблицы, описывающая названия и типы колонок, хранится один раз для таблицы, в отличие, например, от документоориентированной модели, где имена полей хранятся в каждом документе.

В статьях и обсуждениях на форумах в качестве преимущества реляционной модели данных называется поддержка транзакций. Как мы увидим в главе 5 «Гарантии корректности данных», транзакции не связаны с моделью данных, и упомянутые заявления попросту безграмотны. Корни этого заблуждения растут из рекламных материалов ранних NoSQL баз данных, где отсутствие транзакций подавалось как преимущество. Рынок такого «преимущества» не оценил, что вынудило разработчиков пересматривать гарантии целостности, предоставляемые их платформами.

Тем не менее нереляционные базы данных используются широко, и с каждым годом их популярность растёт. В чём же недостатки реляционной модели?

О первом недостатке мы уже упоминали: реляционная модель плохо приспособлена для горизонтального масштабирования, поскольку язык SQL, интерфейс реляционной базы, не предусматривает обязательного указания первичного ключа строки, а потому клиент не может предсказать, где находятся данные, которые потребуются запросу, и какой узел должен выполнить этот запрос.

Второй недостаток — сложность изменения схемы данных. В те годы, когда реляционные базы данных только создавались, это не было проблемой, поскольку программное обеспечение создавалось по «водопадной» модели, где процесс проектирования предшествует процессу разработки, а весь цикл выпуска новой версии приложения растянут во времени. Современные методики разработки предполагают постоянное перепроектирование, которое может потребовать изменения модели данных. В хранилищах «ключ—значение» изменение модели не является событием — просто с какого-то момента в базе появляются объекты новой структуры. В реляционной же базе изменение структуры таблицы —

отдельная команда, выполнение которой может привести даже к приостановке работы приложения. Правда, современные платформы научились выполнять эти команды достаточно эффективно.

Литература

- Comparison of hierarchical and relational databases.
  www.ibm.com/docs/en/ims/15.1.0?topic=ims-comparison-hierarchical-relational-databases
- Эдгар Кодд. Реляционная модель данных для больших совместно используемых банков данных.
  citforum.ru/database/classics/codd/
- Кристофер Дж. Дейт. Введение в системы баз данных (Introduction to Database Systems). 8-е изд. — М.: Вильямс, 2005. — 1328 с. — ISBN 5-8459-0788-8 (рус.) 0-321-19784-4 (англ.).
- Java Temporary Caching API Compatible Implementations.
  jcp.org/aboutJava/communityprocess/implementations/jsr107/index.html
- William Zola. 6 Rules of Thumb for MongoDB Schema.
  www.mongodb.com/blog/post/6-rules-of-thumb-for-mongodb-schema-design-part-1
- Ирина Блажина. JSON Schema. Быть или не быть?
  habr.com/ru/post/495766/
- JSON and BSON.
  www.mongodb.com/json-and-bson
- Rahul Bhatiar. Форматы файлов в больших данных: краткий ликбез / пер. Андрея Пшеничнова.
  habr.com/ru/company/mailru/blog/504952/
- Shilpi Gupta. Двоичное кодирование вместо JSON (перевод выполнен специалистами школы SkillFactory).
  habr.com/ru/company/skillfactory/blog/509902/
- Sylvain Lebresne. A thrift to CQL3 upgrade guide.
  www.datastax.com/blog/thrift-cql3-upgrade-guide
- Jonathan Ellis. Does CQL support dynamic columns / wide rows?
  www.datastax.com/blog/does-cql-support-dynamic-columns-wide-rows
- Jim Wilson. Understanding HBase and BigTable.
  dzone.com/articles/understanding-hbase-and-bigtab

- Apache HBase™ Reference Guide.
  hbase.apache.org/book.html
- Александр Петров. Big Data от А до Я. Часть 4: Hbase.
  habr.com/ru/company/dca/blog/280700/
- Klara Oswald. Знакомство с InfluxDB и базами данных временных рядов.
  tproger.ru/translations/influxdb-guide/
- Полное руководство по Prometheus в 2019 году.
  habr.com/ru/company/southbridge/blog/455290/
- Baron Schwartz. Time-Series Database Requirements.
  www.xaprb.com/blog/2014/06/08/time-series-database-requirements/
- Introduction to Caché.
  docs.intersystems.com/latest/csp/docbook/pdfs/pdfs/GIC.pdf
- Сергей Кузнецов. Три манифеста баз данных: ретроспектива и перспективы.
  citforum.ru/database/articles/manifests/
- The Neo4j Getting StartedGuide v4.2.
  neo4j.com/docs/pdf/neo4j-getting-started-4.2.pdf

Глава 2
Другие методы классификации баз данных

> Селянская классификация нежити несколько отличалась от общепринятой магической, включая всего три вида: «вупыр», «вомпэр» и «щось такэ зубасто».
>
> *Ольга Громыко, «Верховная ведьма»*

Модель данных, применяемая в какой-либо платформе, не является её единственной и исчерпывающей характеристикой. Не менее важными характеристиками являются:

- тип нагрузки, на которую рассчитана платформа;
- возможность работы на нескольких серверах одновременно;
- среда хранения данных — диск или оперативная память.

2.1. Аналитические и транзакционные БД

База данных может применяться либо для обработки транзакций, либо для анализа информации. Традиционно эти два типа нагрузки называются «транзакционный» (OLTP — online transaction processing) и «аналитический» (OLAP — online analytical processing). Некоторые производители пытаются позиционировать свои платформы как «гибридные» (HTAP — hybrid transactional/analytical processing), однако практика показывает, что специализированные решения всегда выигрывают у универсальных.

Различия между типами нагрузки можно представить в виде таблицы:

Показатель	Обработка транзакций	Аналитика
Интенсивность операций	от десятков до десятков тысяч операций в секунду	от единиц до сотен операций в секунду
Время исполнения операции	от десятков миллисекунд до секунд	от секунд до часов
Количество объектов, обрабатываемых операцией	от единиц до тысяч	от десятков тысяч до миллиардов
Соотношение записи и чтения	от 100:1 в пользу чтения до 100:1 в пользу записи	от 1000:1 до 10^6:1 в пользу чтения
Использование колонок при выполнении операции	все	небольшое подмножество
Сложность запросов	низкая	высокая
Разнообразие запросов	низкое (типовые запросы)	высокое (специализированные запросы)

Перечисленные различия требуют различных архитектурных решений, оптимизированных для разных задач.

Транзакционные БД кешируют часто используемые объекты. При обработке аналитических запросов кеширование бесполезно, т. к. объём данных, читаемых за одну операцию, может превышать размеры кеша; вместо этого оперативную память целесообразно использовать для сортировки данных.

Операция фиксации транзакции (commit) в транзакционной СУБД должна быть быстрой и недорогой, т. к. количество транзакций сравнимо с количеством обновляемых объектов. В аналитических БД каждая пишущая транзакция обновляет тысячи или даже миллионы записей, поэтому такие платформы могут позволить себе долгую фиксацию, если это уменьшит общее время записи данных.

Структуры данных в транзакционных БД должны поддерживать быстрое чтение и изменение отдельных объектов с минимальным влиянием на остальные объекты, пусть даже ценой увеличения общего объёма чтения и хранения данных (read/space amplification). Аналитическая БД, наоборот, должна хранить данные наиболее компактно и обеспечивать быстрое считывание всего массива данных, но может позволить существенное удорожание операций записи (write amplification). Транзакционная СУБД вынуждена оставлять свободное место в файле для записи новых версий объекта при увеличении размера полей или хранить несколько версий объекта. Аналитическая СУБД, напротив, может сор-

тировать и сжимать данные при записи. Некоторые аналитические СУБД не поддерживают операцию обновления (update) данных — для обновления необходимо явно удалить (delete) старые данные и записать (insert) новые.

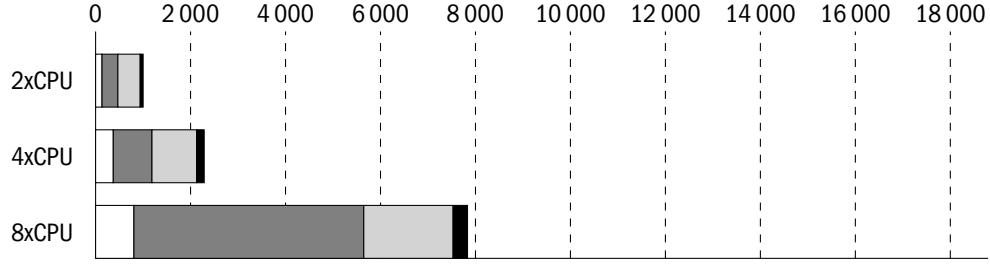
2.2. Монолитные и распределённые БД

Первые системы управления базами данных были рассчитаны на работу на одном сервере. Единственным способом увеличения производительности такой платформы является увеличение производительности сервера. Очевидно, что она не бесконечна.

Сейчас, когда обыкновенный типовой (commodity) сервер является настоящим суперкомпьютером по меркам двадцатилетней давности, есть другая проблема: стоимость сервера зависит от его мощности нелинейно. Если открыть конфигуратор на сайте одного из ведущих мировых производителей серверного оборудования и попробовать сконфигурировать серверы разной вычислительной мощности, то получим такую картину (цены приведены в «условных единицах» с тем расчётом, чтобы стоимость сервера начального уровня составляла 1000 у. е.):

	2-сокетный сервер	4-сокетный сервер	8-сокетный сервер
Количество ядер	32	64	128
Объём оперативной памяти, ГБ	256	512	1 024
Объём дисков, ГБ	480	960	1 920
Стоимость, у. е.	1 000	2 286	7 834

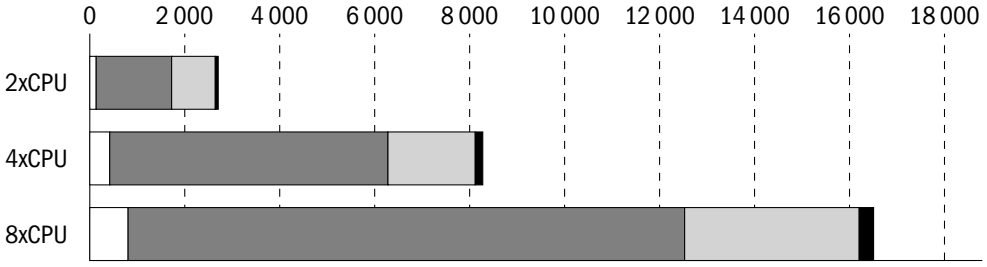
Наглядно соотношение стоимости представлено на диаграмме ниже (цветом выделен вклад в общую стоимость шасси, процессоров, памяти и дисков):



Если вместо 16-ядерных процессоров использовать 28-ядерные[1], картина будет следующая:

	2-сокетный сервер	4-сокетный сервер	8-сокетный сервер
Количество ядер	56	112	224
Объём оперативной памяти, ГБ	512	1 024	2 048
Объём дисков, ГБ	480	960	1 920
Стоимость, у. е.	2 705	8 277	16 508

Диаграмма вклада шасси, процессоров, памяти и дисков выглядит так:



Помимо маркетинговых соображений, у нелинейной зависимости стоимости сервера от максимального количества процессоров есть и технологические причины: от количества сокетов зависит сложность и стоимость как чипсета, так и самого процессора. На второй диаграмме восьмисокетный сервер всего вдвое дороже четырёхсокетного, поскольку в обеих конфигурациях использованы одинаковые процессоры из-за более бедной номенклатуры 28-ядерных чипов. Для четырёхсокетного сервера можно было бы использовать, например, более дешёвые 26-ядерные процессоры, но тогда сравнение было бы не совсем корректным.

Не удивительно, что стали появляться системы, способные к горизонтальному масштабированию, то есть к увеличению мощности за счёт одновременной работы на нескольких серверах. Одними из первых распределённых систем являются Tandem NonStop[2] (1976) и Teradata (1979).

[1] Количество ядер выбрано с тем расчётом, чтобы была возможность собрать любую конфигурацию от двух до восьми сокетов.

[2] Куплен компанией Hewlett-Packard.

Возможность масштабирования на дешёвом оборудовании весьма привлекательна, однако она требует совершенно иного подхода к разработке СУБД, т. к. распределённая система сталкивается с целым рядом проблем, нехарактерных для монолитной системы:

- межпроцессные коммуникации дороги, т. к. сообщение может идти по сети и для обеспечения высокой производительности количество сообщений должно быть минимизировано;
- из-за сетевых отказов межпроцессные коммуникации ненадёжны, поэтому система должна быть готова к потере части сообщений;
- у разных процессов, обслуживающих одну и ту же систему, может быть разное локальное время;
- часть узлов, а вместе с ними и часть процессов (а иногда и часть данных) могут исчезнуть, и система должна штатно реагировать на подобные ситуации;
- узлы могут разделиться на два (или больше) несвязанных подмножества, внутри которых узлы сохраняют связь между собой; получается два независимых кластера, каждый из которых может продолжить работу, считая, что часть узлов потеряна.

Перечисленные проблемы более или менее успешно решаются, и позже мы поговорим об используемых при этом алгоритмах. Однако позиции монолитных СУБД по-прежнему прочны, и нет никаких предпосылок к их исчезновению.

Отдельным подклассом распределённых платформ можно считать облачные базы данных (cloud-only database). Дистрибутивы или исходный код таких платформ нельзя скачать и развернуть на своём оборудовании; можно получить только базу данных как услугу в облаке.

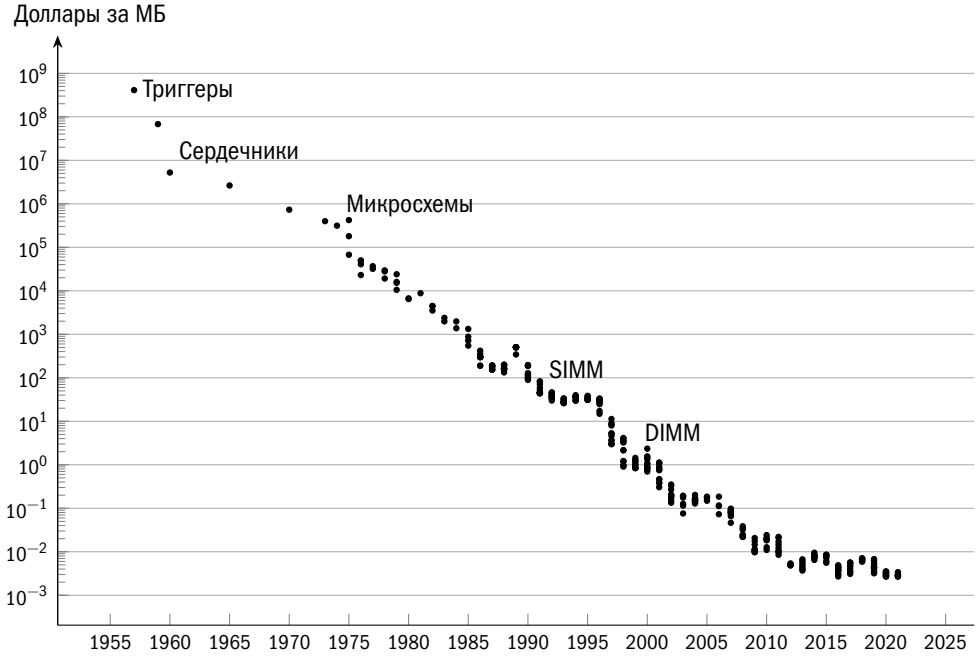
Облачные базы данных обладают рядом технических преимуществ. Во-первых, они могут быть сколь угодно сложными, поскольку рассчитаны на эксплуатацию не администраторами средней квалификации, а людьми, имеющими непосредственный контакт с разработчиками платформы. Во-вторых, облачная база данных может рассчитывать на любые особенности инфраструктуры — особые сетевые топологии и нестандартные протоколы, специализированные конфигурации серверов или наличие уникального оборудования — например, атомных часов.

Разработкой облачных баз занимаются крупные провайдеры облачных услуг. Примерами таких платформ могут служить Google BigTable, Google Spanner, Amazon DynamoDB, Amazon Aurora, Azure CosmosDB, YDB[1].

[1] Исходный код YDB доступен с 2022 года под лицензией Apache 2.0.

2.3. БД на диске и в памяти

Традиционные платформы хранения и обработки данных появились в то время, когда оперативная память была дорога, а объёмы памяти были весьма скромными. Так, например, объём памяти флагманского мейнфрейма Model 195 серии IBM System/360, выпущенного в 1969 году, мог достигать 4 МБ. Естественно, СУБД того времени могли хранить данные только на дисках, загружая их в память небольшими порциями.



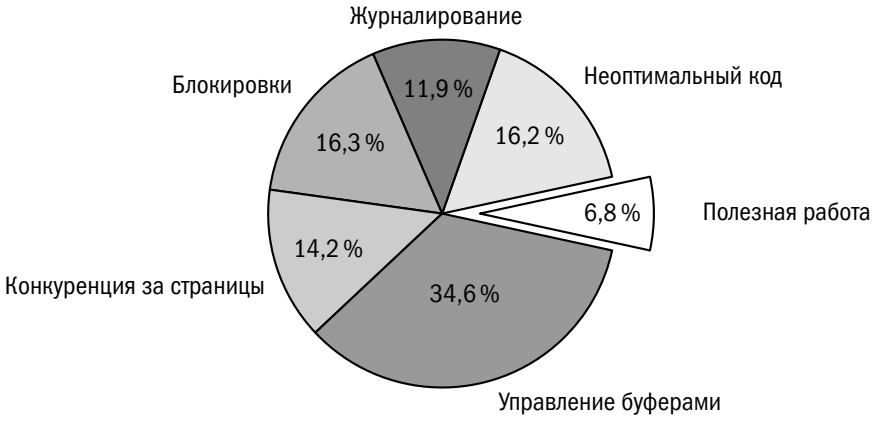
За 50 лет объём памяти в серверах вырос на шесть порядков, а на графике, построенном по данным сайта jcmit.net (шкала логарифмическая), видно, что стоимость памяти за это же время упала на восемь порядков. Естественным образом возникло желание создать платформы, хранящие и обрабатывающие данные непосредственно в оперативной памяти и использующие диск исключительно для восстановления данных после аварий.

Идея «взять обычную СУБД и просто дать ей больше памяти» выглядит заманчиво, но работает плохо. Дело в том, что в дисковых СУБД вся архитектура подчинена тому, чтобы оптимизировать работу с диском, и когда производительность дисковой подсистемы перестаёт быть узким местом, система упирается в другие

ограничения, которые не снимаются увеличением мощности оборудования. Обратим внимание лишь на некоторые проблемы.

- Дисковое представление данных может содержать пустое пространство: во-первых, данные выравниваются по границе блока, но могут не занимать блок целиком, а во-вторых, СУБД может намеренно оставлять запас пространства на случай увеличения размеров полей, хранящихся в блоке. Очевидно, что и в оперативной памяти это место будет пустым, хотя его можно было бы использовать.
- Дисковые СУБД используют оперативную память как кеш, то есть адрес блока в памяти никак не связан с его положением в базе данных. Вместо этого СУБД вынуждена хранить таблицу соответствия между физическим адресом блока и его положением в кеше. Практика показывает, что начиная с некоторого объёма[1] эффективность кеша падает, т. к. выигрыш от уменьшения обмена с диском уравновешивается накладными расходами на поиск данных в кеше.

Группа учёных под руководством Майкла Стоунбрейкера провела исследование, на что дисковая СУБД тратит ресурсы процессора. Результаты получились ошеломительными: полезная нагрузка по обработке данных составила чуть меньше 7 %! Остальные ресурсы процессора расходуются на управление буферами (buffer management), обеспечение конкурентного доступа к дисковым структурам (latching) и логическим записям (blocking), журналирование (logging)[2].

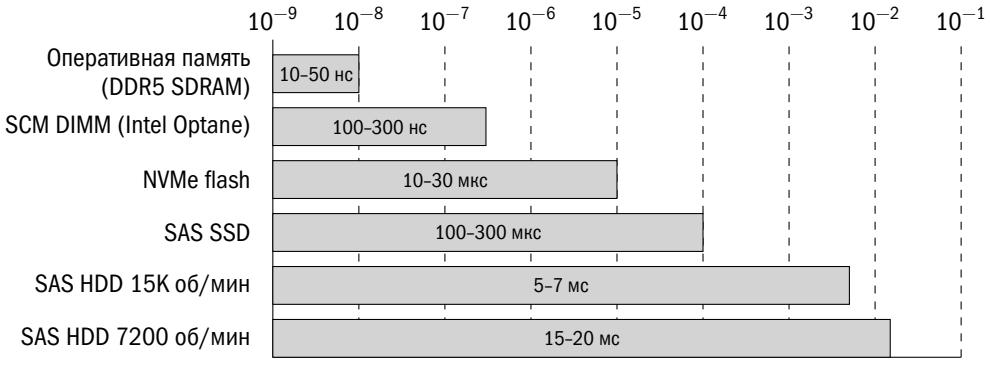


[1] Конкретный объём зависит от СУБД. Например, для Oracle он составляет 1–2 ТБ.

[2] Эксперимент проводился на модельной СУБД с открытым исходным кодом. На промышленных платформах распределение, вероятно, иное.

Первая коммерческая СУБД, содержавшая движок для хранения и обработки данных в памяти, solidDB, появилась в 1992 году, и совсем немного отстал стартап TimesTen, основанный в 1996 году и купленный в 2005 году корпорацией Oracle. В первой декаде нового тысячелетия появилось множество платформ, предоставляющих распределённый кеш в памяти (Memcached, Redis), а также встраиваемых библиотек для организации распределённого кеша внутри приложения (GigaSpaces XAP, Terracotta Ehcache, Hazelcast, GridGain). В 2010-х годах были созданы полноценные СУБД в памяти — VoltDB, SAP HANA, Tarantool, Picodata.

Оперативная память по-прежнему на три порядка дороже, чем диски, поэтому уход с рынка дисковых СУБД в обозримом будущем вряд ли произойдёт. Однако возможно, что нас ждёт очередная революция, связанная с расширением номенклатуры систем хранения данных. Если раньше разработчикам были доступны только диски и оперативная память, то сегодня появился как широкий выбор дисковых устройств, существенно различающихся по своим характеристикам, так и энергонезависимая память (SCM, storage-class memory), например Intel Optane. Энергонезависимая память занимает промежуточное положение по стоимости и производительности между оперативной памятью и дисками. На диаграмме ниже показана задержка (latency) при доступе к разным видам памяти.



2.4. Карта баз данных

Попытаемся классифицировать широко известные платформы по перечисленным признакам: модель данных, возможность горизонтального масштабирования, тип нагрузки, среда хранения:

	Монолитные дисковые	**Монолитные в памяти**	**Распределённые дисковые**	**Распределённые в памяти**
Реляционные	Oracle, PostgreSQL, Microsoft SQL Server, Db2, MySQL, SQLite	Oracle TimesTen, SAP HANA, solidDB	**Teradata, Greenplum, SAP IQ**, **Vertica**, **ClickHouse,** Google Spanner, Oracle Real Application Cluster, Oracle with Sharding Option, IBM PureData System for Transactions, HPE NonStop SQL, CockroachDB, YugabyteDB, YDB	**SAP HANA, Exasol,** Picodata, SingleStore (бывший MemSQL), Oracle TimesTen Scaleout IMDB, VoltDB
Ключ–значение	GT.M, RocksDB, Berkeley DB		Riak KV, Project Voldemort	Redis, Hazelcast, Memcached, etcd, Terracotta Ehcache, Aerospike, GridGain/Apache Ignite, Infinispan, Oracle Coherence, Pivotal GemFire/ Apache Geode, Tarantool, WebSphere eXtreme Scale
Документо-ориентиро-ванные			MongoDB, Elasticsearch, Amazon DynamoDB, Couchbase, CouchDB, MarkLogic, Oracle NoSQL	
Семейства колонок			Cassandra, HBase, Google BigTable, ScyllaDB, Apache Accumulo	
Временны́е ряды	Graphite, TimescaleDB	kdb+	InfluxDB, Prometheus, VictoriaMetrics	
Графовые	Amazon Neptune	Memgraph	Neo4j, ArangoDB, OrientDB, JanusGraph	

На полученной карте видно несколько аномалий:

- практически все платформы класса «ключ—значение» распределённые; более бедная (относительно реляционной) модель данных является платой за относительную простоту разработки распределённой системы;
- все аналитические системы (выделены жирным шрифтом) — реляционные; это прямое следствие того, что реляционная модель даёт наиболее мощные инструменты анализа данных.

Карту полезно держать перед глазами, чтобы исключить сравнение несравнимых платформ — например, «PostgreSQL vs. Redis» или ещё лучше «БД в памяти vs. реляционные БД» (дословная цитата из статьи в одном из блогов на сайте habr.com).

Тем не менее необходимо помнить, что эта карта, как и любая классификация, достаточно условна.

Условно деление по модели данных.

В 2011 году Мартин Фаулер (Martin Fowler) опубликовал статью «Многовариантное хранение» («Polyglot persistence»). По словам Фаулера, будущее — за теми платформами, которые могут предложить пользователям несколько разных интерфейсов для хранения разных данных в рамках комплексного решения сложной задачи.

Многие популярные современные платформы поддерживают несколько моделей данных. Например, Aerospike и Tarantool, которые позиционируются как хранилища «ключ—значение», поддерживают обращение к отдельным полям. Документоориентированная Couchbase наряду с Aerospike поддерживает вторичные индексы. Такие платформы, как GridGain, Pivotal GemFire, даже поддерживают язык SQL, а Picodata, бывшая поначалу форком Tarantool, стала полноценной реляционной платформой. Реляционные Oracle и PostgreSQL поддерживают хранение XML и JSON с обработкой документов на сервере, а также работу с графами[1].

Условно деление по типу нагрузки. Некоторые платформы (например, SAP HANA или SingleStore) содержат несколько движков разного назначения, а Oracle в зависимости от настроек экземпляра может эффективно работать и как транзакционная, и как аналитическая платформа. Графовые СУБД используются преимущественно для анализа данных, но при этом выполняется множество мелких читающих запросов — как при обработке транзакций.

[1] Работа с графами реализована в СУБД AgensGraph, базирующейся на PostgreSQL.

Условно деление по возможности горизонтального масштабирования. В категорию распределённых попадают как классические платформы, работающие на полностью независимых узлах (архитектура «shared nothing»), так и платформы с общим дисковым пространством (архитектура «shared disk» — Oracle RAC, HPE NonStop, Vertica).

Условно деление по среде хранения. Многие классические дисковые СУБД (Oracle, Microsoft SQL Server) получили движки для обработки данных в памяти, в то время как платформы, работающие в памяти (solidDB, GridGain, Tarantool, MemSQL/SingleStore), обзавелись подсистемами хранения на дисках. Есть и платформы, которые с самого начала разрабатывались как гибридные — например, kdb+.

Даже само понятие «система управления базами данных» весьма условно. Помимо платформ, реализующих полный набор механизмов хранения и обработки данных, в карту попали встраиваемые системы (SQLite, Berkeley DB, RocksDB), а также надстройки над существующими платформами, обеспечивающие дополнительные модели обработки данных (JanusGraph, Prometheus, Graphite, TimescaleDB, Project Voldemort).

Для тех, кто желает самостоятельно классифицировать СУБД и исследовать родственные взаимоотношения разных платформ, настоящей находкой станет сайт «База баз» (Database of Databases) dbdb.io.

Литература

- Владислав Бурмистров. Поколоночное и гибридное хранение записей в СУБД Teradata.
  habr.com/ru/company/teradata/blog/170321/
- British Isles Tandem Users Group. About HPE NonStop.
  www.bitug.com/about-hp-nonstop
- The Teradata Company History.
  www.teradata.ru/About-Us/Our-History
- Memory Prices 1957+.
  jcmit.net/memoryprice.htm
- Константин Осипов. Что особенного в СУБД для данных в оперативной памяти.
  habr.com/ru/company/oleg-bunin/blog/310560/

- Сергей Кузнецов. В ожидании нативных архитектур СУБД на основе энергонезависимой основной памяти.
  www.ispras.ru/proceedings/docs/2020/32/1/isp_32_2020_1_153.pdf
- Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos. OLTP Through the Looking Glass, and What We Found There.
  nms.csail.mit.edu/~stavros/pubs/OLTP_sigmod08.pdf
- Martin Fowler. Polyglot Persistence.
  martinfowler.com/bliki/PolyglotPersistence.html

Часть II

Доступ к данным

Глава 3
Структуры хранения данных

> Плохие программисты беспокоятся о коде. Хорошие программисты в первую очередь думают о структурах данных и отношениях между ними.
>
> *Линус Торвальдс*

В этом разделе поговорим о структурах хранения данных. Разумеется, СУБД скрывает их от разработчика, но знание внутренней структуры позволит понять, почему те или иные операции выполняются быстро или, наоборот, медленно, а также какую платформу лучше выбрать, исходя из предполагаемой нагрузки.

3.1. Общая информация о дисковых структурах

Изменяемые и неизменяемые структуры

Все дисковые структуры делятся на две большие группы — изменяемые (mutable) и неизменяемые (immutable). В первых новые версии данных записываются поверх старых, а во вторых сохранённые данные не меняются, а при изменении объектов их новые версии записываются в другое место.

Системы, использующие изменяемые структуры (а это все традиционные реляционные СУБД, такие как Oracle, Db2, Microsoft SQL Server, PostgreSQL, MySQL), должны иметь возможность в любой момент прочитать и перезаписать любую область данных. Поскольку при любой операции с диском читается или записывается не менее одного блока информации, данные делятся на страницы (pages), размер которых кратен размеру блока системы хранения данных (СХД) и составляет от 2 до 64 КБ — в зависимости от характеристик используемой СХД и настроек конкретной базы данных. Страница является минимальной единицей обмена СУБД с диском. К сожалению, производители СУБД не выработали единую терминологию, и иногда страницы называют блоками (blocks)[1].

[1] Например, в документации к СУБД Oracle.

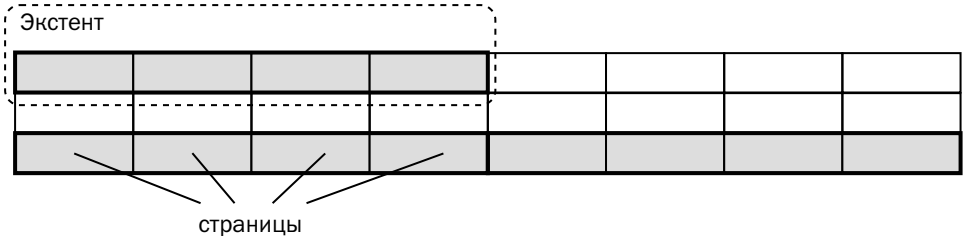
Важным преимуществом страничной структуры является то, что она может быть реализована без файловой системы, прямо поверх «сырого» блочного устройства (raw device) — диска. Отказ от файловой системы способствует повышению надёжности и производительности платформы за счёт исключения лишнего уровня абстракции, но усложняет сопровождение. Работа с блочными устройствами реализована в Oracle и Db2.

В современных СУБД используется два подхода к управлению страницами.

Первый подход (Oracle, Microsoft SQL Server, Db2 Database Managed Tablespaces, MySQL InnoDB General Tablespaces) заключается в создании дополнительного уровня абстракции — табличного пространства (tablespace), или файловой группы (filegroup) в Microsoft SQL Server. Табличное пространство состоит из одного или нескольких файлов (или блочных устройств), а внутри табличного пространства могут храниться данные разных объектов — например, таблиц или индексов.

Место выделяется группами идущих подряд страниц — экстентами (extent). Каждый экстент определяется файлом, в котором он расположен, номером первой страницы и количеством страниц. Поскольку базы данных год от года становятся всё больше, постраничный учёт дискового пространства неэффективен. Для уменьшения накладных расходов на учёт в современных платформах количество страниц во всех экстентах табличного пространства одинаково, что позволяет резко сократить объём метаданных.

Все страницы, принадлежащие одному объекту, образуют сегмент (segment)[1]. На рисунке ниже сегмент показан затемнением. Он состоит из 12 страниц в трёх экстентах.



Альтернативный подход к управлению страницами заключается в выделении файла операционной системы под каждый объект. Такой подход используется в PostgreSQL, Db2 System Managed Tablespaces и InnoDB File-per-table Tablespaces.

[1] Секционированные объекты — исключение из этого правила.

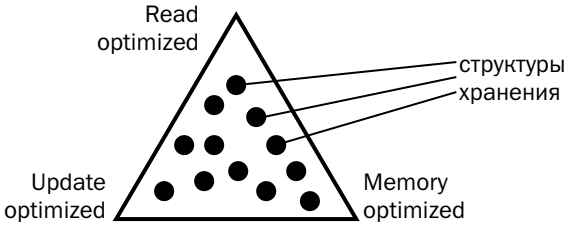
Системы, использующие неизменяемые структуры (а это практически все NoSQL-платформы — MongoDB, Cassandra, Riak KV и др.), при необходимости изменить данные просто сохраняют новое значение в другом месте, не изменяя старого. В отличие от страничной организации изменяемых структур, внутренняя организация неизменяемых структур проста: данные представляют собой сплошную последовательность байтов. Поскольку в таких структурах файлы часто создаются и удаляются, все движки с неизменяемыми структурами используют файловые системы.

Движкам с неизменяемыми структурами необходимы фоновые процессы, вычищающие неактуальные копии данных.

Эффективность операций

Одна из важнейших характеристик любой структуры данных — эффективность, которую оценивают по трём параметрам: скорости чтения (read), скорости записи (write, или update) и занимаемому пространству (memory, или space). Эффективность определяется тем, во сколько раз прочитанный (записанный, сохранённый) объём больше, чем реальный объём данных.

Существует гипотеза, что структуры, в которых наиболее эффективно реализована одна из операций, наименее эффективно реализуют остальные две (RUM conjecture):



Это именно гипотеза, строгого её доказательства не существует, но на практике пока никому не удалось создать структуру данных, которая позволяла бы одинаково эффективно выполнять чтение и запись и при этом данные занимали бы ровно столько дискового пространства, сколько необходимо для их хранения. Вместо этого современные платформы используют несколько структур данных одновременно, и одни компенсируют недостатки других.

Разумеется, у каждой структуры свои коэффициенты усложнения (amplification), но можно вывести некоторые общие принципы для изменяемых (страничных) и неизменяемых структур:

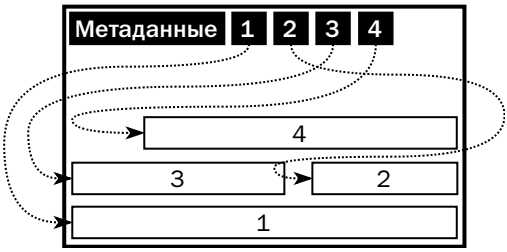
- Минимальная единица обмена с диском базы со страничной структурой — одна страница, следовательно, мультипликатор чтения (и записи) равен или больше отношения размера страницы к размеру извлекаемых (записываемых) данных. Разумеется, утверждение верно только в том случае, если размер данных меньше размера страницы, но в подавляющем большинстве случаев это именно так. Для уменьшения обмена с диском платформы со страничной структурой всегда кешируют страницы.
- Страничная СУБД хранит в каждой странице некоторую метаинформацию. Кроме того, размер данных в странице не всегда идеально совпадает с её размером, из-за чего в каждой странице остаётся неиспользуемое пространство. Таким образом, даже при идеально заполненных страницах мультипликаторы хранения и чтения могут достигать 1,05–1,10 (то есть данные занимают на диске на 5–10 % больше их реального объёма).
- В некоторых случаях строка, не помещающаяся в страницу, распределяется по нескольким страницам, и её мультипликаторы чтения и записи становятся ещё больше. Администратор БД имеет возможность оставить в каждой странице запас пространства на случай увеличения размера строки — этим потенциально уменьшаются мультипликаторы чтения и записи, но гарантированно увеличивается мультипликатор пространства.
- Платформы с неизменяемыми структурами пишут ровно столько, сколько занимают изменённые объекты, поэтому операции записи в них выполняются крайне эффективно. На самом деле есть ещё фоновые процессы, вычищающие неактуальные копии данных, и с учётом фоновых процессов мультипликатор записи может достигать нескольких десятков. Однако процессы очистки работают асинхронно, и их влиянием на производительность можно управлять.
- Мультипликатор пространства в неизменяемой структуре зависит от того, насколько часто изменяются конкретные объекты. Если данные приложения, работающего с базой, никогда не меняются (например, это журнал бизнес-операций), то мультипликатор пространства равен единице. Если данные изменчивы (например, текущее состояние регистра), то мультипликатор пространства может достигать нескольких десятков.

В большинстве движков баз данных помимо основной структуры присутствует и журнал, который увеличивает общий объём записи, но позволяет растянуть его во времени. О структуре и назначении журнала поговорим в главе 5 «Гарантии корректности данных».

3.2. Изменяемые структуры

Неупорядоченная таблица

Самая простая изменяемая структура данных — неупорядоченная таблица (heap-organized table). Каждая страница хранит несколько строк таблицы, и сама по себе структура не требует упорядоченности строк.



В начале каждой страницы находятся метаданные, за ними — заголовки строк, указывающие на начало данных соответствующей строки, а сами данные находятся в конце страницы. Между заголовками строк и данными находится свободное пространство. Комбинация «номер страницы + номер строки в странице» указывает на физическое положение строки и позволяет мгновенно её отыскать.

Данные в странице хранятся максимально компактно, без промежутков между строками. При изменении значений полей может изменяться и размер строки — например, меняется количество разрядов в числе или количество символов в текстовом поле. При изменении размера строки положение изменившейся строки и всех строк, хранящихся ближе к началу страницы, изменится, но физический адрес останется неизменным.

Стандарт SQL никак не регламентирует работу с физическим идентификатором строки, поэтому название, формат, возможности доступа и правила работы с такими идентификаторами отличаются от платформы к платформе:

СУБД	Название	Как получить	Действителен
Oracle	rowid	псевдостолбец `ROWID`	Всё время жизни строки
PostgreSQL	ctid	псевдостолбец `CTID`	До выполнения команды `vacuum`[1]

[1] Строго говоря, ctid уникально идентифицирует не строку, а версию строки.

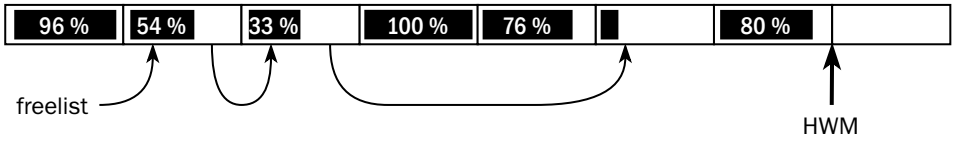
	Название	Как получить	Действителен
Microsoft SQL Server	rid	псевдостолбец %%physloc%%	Всё время жизни строки
Db2 for Linux, Unix, Windows Db2/i	rid	функция RID_BIT()	Всё время жизни строки
Db2 for z/OS	rid	функция RID_BIT()	В пределах транзакции

Некоторые платформы (PostgreSQL, Db2 for z/OS) записывают каждую версию строки в новое место. Другие платформы при обновлении строки сохраняют её физический адрес[1], а если строка не помещается в страницу, то в старой странице остаётся её заголовок, а данные переезжают в свободную страницу. Даже в платформах, сохраняющих физический адрес, администратор БД может выполнить реорганизацию таблицы, и тогда все строки переедут на новые места, и их физические адреса изменятся.

Несмотря на кажущуюся простоту неупорядоченной таблицы, есть два вопроса, на которые разработчики разных платформ отвечают по-разному.

Вопрос номер один: куда записывать новые строки?

Возможных ответа на него два: в конец таблицы или в страницы, где есть свободное пространство. В свою очередь, информацию о свободных страницах можно хранить в виде списка или в виде битовых карт.



Отметка максимального заполнения (highwatermark, или HWM) — место в таблице, после которого гарантированно нет никаких данных. В зависимости от реализации отметка может быть адресом в выделенном заранее пространстве или совпадать с физическим концом файла. Простейший алгоритм поиска свободного пространства заключается в том, чтобы записывать данные в последнюю страницу перед отметкой максимального заполнения, а после исчерпания места в странице сдвигать отметку на одну страницу вправо (или добавить к файлу новую страницу). Такая стратегия применяется в Db2, если таблица создана с опцией ORGANIZE BY ROW USING INSERT TIME. Алгоритм прост, но неэффективен, если

[1] Исключение из этого правила — обновление ключа секционирования, о котором речь в разделе 3.5 «Локализация данных».

данные из таблицы удаляются — в этом случае слева от отметки максимального заполнения образуются частично заполненные страницы, свободное пространство в которых не будет использовано до физической реорганизации таблицы.

Oracle предлагает быстрый метод загрузки данных в таблицу — прямую загрузку (direct path load). При использовании этого метода клиентское приложение (например, фирменная утилита SQL*Loader) формирует готовые образы страниц и передаёт их серверу, который добавляет страницы в таблицу справа от отметки максимального заполнения.

Если принято решение повторно использовать страницы со свободным пространством, то в базе данных может храниться список свободных страниц (freelist). Первая страница сегмента содержит указатель на страницу, в которой есть свободное пространство; та, в свою очередь, содержит указатель на следующую страницу, куда можно записать данные, и так далее. Такой подход используется в Oracle — до версии 8i включительно он был единственным, а в текущих версиях сохраняется для совместимости. Такой же подход используется в MySQL; правда, там в списки входят не страницы, а экстенты. Недостаток списка в том, что множество транзакций, работающих с одной и той же таблицей, конкурируют за него, и для уменьшения конкуренции Oracle позволяет создавать в сегменте несколько списков.

Самый распространённый подход — хранение битовых карт (bitmaps), где каждой странице отводится несколько битов, показывающих степень её заполнения. Эта структура допускает гораздо более эффективное совместное использование, чем список свободных блоков. В том или ином виде битовые карты применяются в PostgreSQL, Microsoft SQL Server и в Oracle начиная с версии 9i в табличных пространствах с автоматическим управлением пространством (automatic segment space management, ASSM). Название «битовая карта» не подразумевает плоскую структуру — для оптимизации поиска свободных страниц структура карт может быть, например, древовидной.

Таблица с более сложными правилами размещения строк, подразумевающими сортировку, называется кластеризованной таблицей (clustered table). Строки в ней упорядочены в соответствии со значением одного или нескольких полей. Кластеризованные таблицы поддерживаются всеми основными реляционными СУБД, но детали реализации несколько отличаются:

- Oracle упорядочивает строки кластеризованной таблицы (attribute-clustered table) только во время прямой загрузки, а дальше работает с ней точно так же, как с обычной таблицей;

- в PostgreSQL нет отдельного понятия «кластеризованная таблица», но есть оператор `CLUSTER`, который переупорядочивает строки в таблице в соответствии с одним из индексов на этой таблице; в дальнейшем PostgreSQL работает с таблицей как обычно, не сохраняя порядок вновь созданных или изменённых строк;
- Db2 позволяет определить в таблице набор полей, по которым будет проводиться кластеризация (таблицы с опцией `ORGANIZE BY DIMENSIONS`); в момент определения такого набора полей СУБД автоматически создаёт по ним индексы и в дальнейшем по возможности поддерживает порядок строк в таблице;
- Db2 позволяет также определить ключ кластеризации (таблицы с опцией `ORGANIZE BY KEY SEQUENCE`) и диапазон значений ключа. В этом случае при создании таблицы выделяется пространство сразу под все возможные строки — в соответствии с возможными значениями ключа. Если строка имеет переменную длину (например, в ней есть колонки типа `VARCHAR`), то пространство резервируется с расчётом на максимальную длину. Каждая новая строка попадает на отведённое для неё место.

Второй важный вопрос: сколько свободного пространства следует оставлять на странице на случай увеличения объёма данных в строках, например расширения текстового поля.

- Microsoft SQL Server скрывает ответ на этот вопрос от пользователя, управляя свободным пространством самостоятельно.
- Db2 при создании таблиц позволяет задать параметр `PCTFREE` — объём свободного пространства в процентах от размера страницы. Новые строки записываются в страницу до тех пор, пока объём свободного пространства не уменьшится до `PCTFREE`. После этого страница считается заполненной, и свободное пространство в ней используется только для изменения данных в уже записанных строках. Если объём свободного пространства превысит `PCTFREE`, то страница вновь считается свободной, и СУБД может записать в неё новую строку.
- Логика PostgreSQL такая же, как у Db2, но там параметр создания таблицы называется `FILLFACTOR` и обозначает максимальный объём занятого пространства в таблице, после которого страница считается заполненной. Другими словами, `FILLFACTOR 80` — это то же самое, что `PCTFREE 20`.
- Oracle предоставляет пользователю два параметра — `PCTFREE` и `PCTUSED`. Смысл параметра `PCTFREE` тот же, что у Db2, но после заполнения выше

PCTFREE страница будет вновь считаться свободной только тогда, когда объём свободного пространства превысит PCTUSED.

Очевидно, что если данные в таблицу загружаются один раз, то имеет смысл задать параметр PCTFREE 0, но если приложение предполагает изменение данных, то лучше оставить в каждой странице запас свободного пространства.

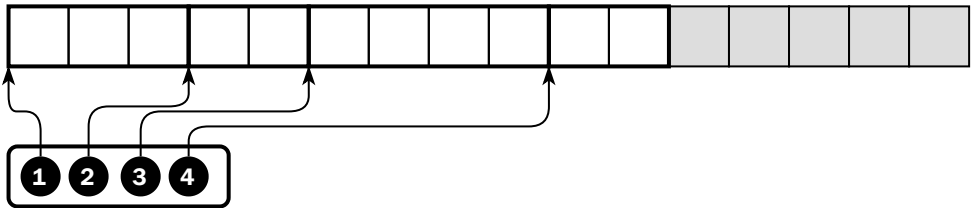
Сжатие данных

У пользователей базы данных возникает естественное желание уменьшить объём занимаемого пространства, особенно если узким местом в работе являются операции ввода-вывода. Существует два подхода к сжатию данных:

- сжатие блоков стандартными алгоритмами (например, из библиотеки zlib);
- словарное сжатие данных внутри блока с использованием алгоритмов семейства LZ.

Разные реализации первого подхода используются в MySQL и Postgres Pro Enterprise — версии PostgreSQL, поставляемой компанией Postgres Professional.

Postgres Pro[1] записывает сжатые страницы вплотную друг к другу, выравнивая каждую страницу по границе блока СХД. Теперь для того, чтобы найти страницу, платформа вынуждена поддерживать карту страниц и при чтении страницы сначала вычислять её физический адрес по карте:



Этот подход добавляет накладные расходы при чтении страниц вразбивку, но существенно ускоряет чтение страниц подряд.

MySQL с движком InnoDB[2] поступает по-другому. Каждая страница записывается на предназначенную ей позицию, но у страницы может быть неиспользуемый «хвост», и БД сообщает операционной системе, что выделять пространство

[1] СУБД Postgres Pro поставляется в двух версиях — Standard и Enterprise. Здесь и далее торговая марка Postgres Pro может означать как одну, так и другую версию. За точным списком функций обратитесь к фирменной документации.

[2] InnoDB не поддерживает неупорядоченные таблицы, но описанный алгоритм позволяет сжимать любые страницы вне зависимости от их содержимого.

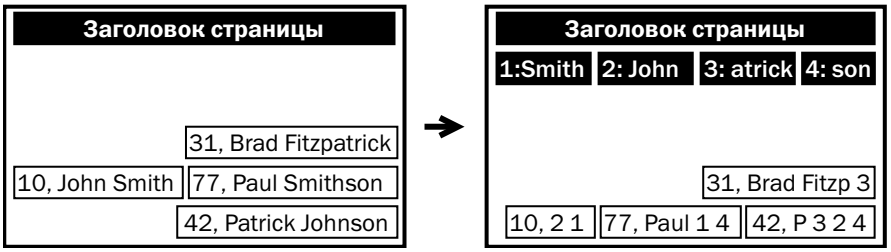
под «хвост» не надо, его можно использовать для других файлов. MySQL использует механизм разреженных файлов (sparse files), поддерживаемый NTFS в Windows Server и большинством файловых систем Linux. На рисунке ниже размер страницы БД равен четырём блокам СХД, а серым показаны свободные блоки СХД.

Подход MySQL, в отличие от подхода Postgres Pro, позволяет быстрее читать отдельные страницы, но при чтении страниц подряд ускорение не столь существенно.

Интересен механизм хранения сжатых страниц в памяти:

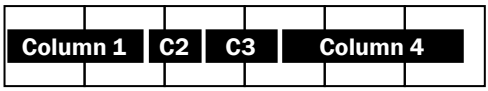
- сжатая страница загружается в память, занимая предназначенный ей буфер, а затем её данные разжимаются, занимая ещё один буфер, то есть сжатая и несжатая копии страницы хранятся в памяти одновременно;
- изменения вносятся и в сжатую, и в несжатую копии страницы, причём сжатая страница не разжимается — вместо этого в конец записывается журнал изменений;
- если места в буферном кеше не хватает, то несжатая страница может быть выброшена из памяти без записи на диск;
- если журнал изменений не помещается в страницу, данные страницы разжимаются, к ним применяются изменения, и затем данные снова сжимаются;
- если даже сжатые данные не помещаются в страницу, страница делится на две в соответствии с правилами платформы и соответствующей структуры данных.

Microsoft SQL Server, Oracle и Db2 используют второй подход — словарное сжатие. В данных страницы находятся повторяющиеся значения, которые заменяются ссылками на словарь. Сам словарь находится в той же самой странице. Такой подход сохраняет размер страницы, но позволяет поместить в одну страницу больше данных:



Есть и другие методы сжатия данных, но они уже выходят за рамки страничной структуры. Таково, например, гибридное сжатие (hybrid columnar compression) в Oracle:

- СУБД берёт группу строк и переупорядочивает данные так, что сначала идут значения первого поля, затем значения второго поля в том же порядке и т. д.;
- данные полей сжимаются;
- сжатые данные записываются в выделенный для этой группы строк набор страниц.

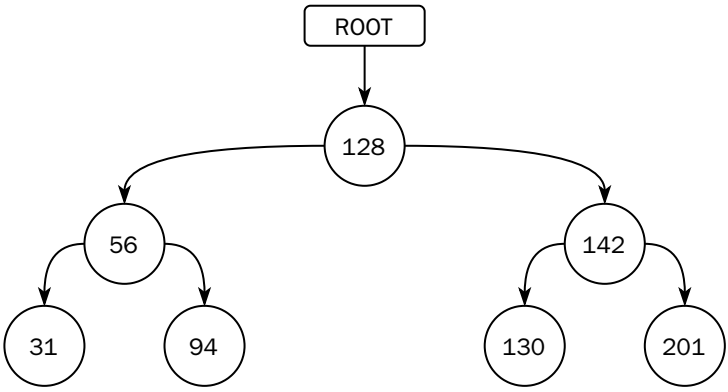


Такое сжатие очень эффективно, однако получившаяся структура ближе к хранилищу колонок, чем к неупорядоченной страничной таблице.

B-дерево

Неупорядоченная таблица весьма проста, однако поиск данных в ней неэффективен. Для поиска предпочтительнее упорядоченная структура, допускающая не только полный просмотр данных, но и более эффективные алгоритмы.

Пожалуй, самая известная из упорядоченных структур — двоичное дерево (binary tree), то есть направленный ациклический граф, каждый узел которого имеет максимум двух потомков. Узел, не имеющий предков, называется корнем (root), а узел, не имеющий потомков, — листом (leaf):



В каждом узле двоичного дерева находится какой-либо ключ. Ключи, находящиеся в узлах левого поддерева, всегда меньше, а ключи в узлах правого поддерева — всегда больше, чем ключ в узле-предке. Таким образом, если дерево

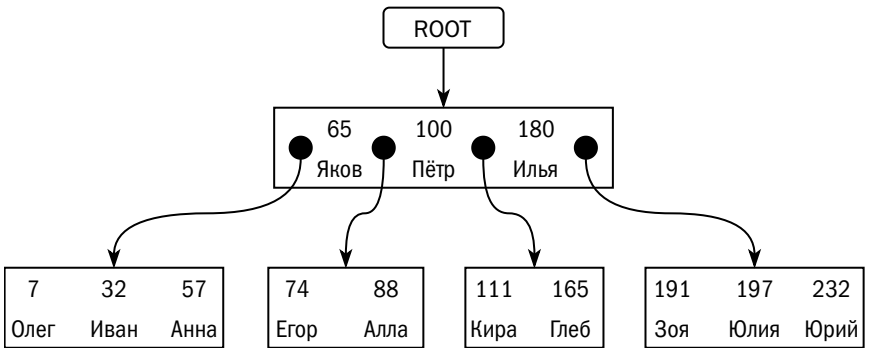
сбалансировано, то есть расстояния от корня до листовых узлов отличаются не более, чем на единицу, то любой ключ в дереве находится за время, пропорциональное двоичному логарифму от количества ключей (узлов).

При всех достоинствах двоичного дерева у него есть несколько существенных недостатков:

- по умолчанию двоичное дерево не сбалансировано, то есть разные поддеревья могут иметь разную длину; поддержка баланса требует применения специальных алгоритмов (например, АВЛ-дерево), но дерево приходится перестраивать почти при каждой операции вставки или удаления;
- накладные расходы в двоичном дереве очень велики: для каждого значения нужно хранить два указателя — на левое и правое поддеревья;
- нет гарантии, что связанные узлы находятся в памяти рядом друг с другом; это значит, что при обходе дерева придётся хаотично перемещаться по памяти, а если каждое перемещение будет сопровождаться считыванием страницы с диска, то производительность такой структуры будет крайне низка. Кроме того, и механизм упаковки узлов двоичного дерева в страницы не очевиден.

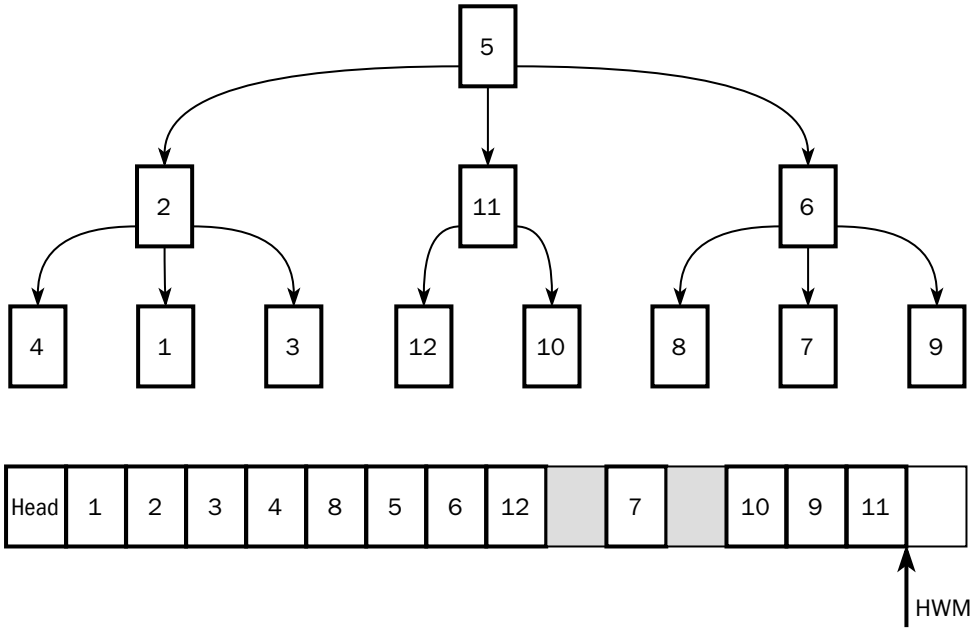
В 1970 году инженеры компании Боинг Рудольф Бэйер (Rudolf Bayer) и Эдвард Мак-Крейт (Edward M. McCreight) предложили структуру под названием B-tree (по-русски — B-дерево или «би-дерево»). Буква «B» может означать Boeing, Bayer, «balance» (сбалансированное), «broad» (широкое), «bushy» (кустистое), а также любое другое слово, которое читатель придумает для описания этой структуры.

Узел B-дерева представляет собой список пар «ключ—значение», перемежающихся ссылками на поддеревья. Все ключи в дереве уникальны. Ключи в узле упорядочены, а значения всех ключей в поддереве, на которое указывает ссылка между ключами K_n и K_{n+1}, лежат в промежутке $(K_n; K_{n+1})$:



Сложность поиска в B-дереве по-прежнему логарифмическая, но узел такого дерева можно сделать достаточно большим, а именно — размером со страницу базы данных.

Страницы в базе данных естественным образом упорядочены, но положение страницы в файле базы данных никак не связано с положением узла B-дерева, размещённого в этой странице:

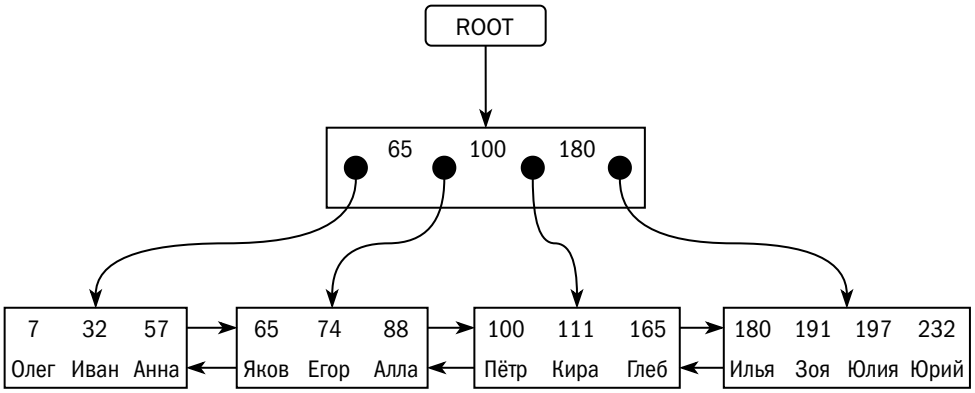


В современных СУБД обычно применяют так называемые B^+-деревья, которые представляют собой B-деревья со следующими модификациями:

- значения содержатся только в листах дерева — в корневом узле и в промежуточных узлах содержатся лишь ключи;
- листовые узлы образуют двунаправленный список, что позволяет перебирать диапазоны ключей — найти минимальный (максимальный) и двигаться по списку вправо (влево);
- ключи в поддереве, на которое ссылается указатель между K_n и K_{n+1}, лежат в промежутке $[K_n; K_{n+1})$, то есть ключ может дублироваться в дочернем узле.

B^+-дерево очень сильно ветвится: количество потомков у каждого узла на единицу больше, чем N — количество ключей, помещающихся в страницу. Глубина дерева, то есть количество узлов, которые требуется обойти в поисках нужного

значения, равна логарифму по основанию *N* от количества ключей и на практике редко превышает четыре[1].



B-дерево (здесь и далее под этим термином будем понимать все модификации, включая B^+-дерево, B^*-дерево и др.) можно использовать двумя разными способами:

- в качестве ключа дерева используется первичный ключ или любое другое индексируемое поле, а в качестве значения — указатель на данные;
- в качестве ключа дерева используется первичный ключ, а в качестве значения — все остальные поля таблицы.

При первом подходе данные хранятся в неупорядоченных таблицах, B-деревья используются для построения индексов, а в качестве указателя на данные используется физический адрес (идентификатор) строки в неупорядоченной таблице. Этот подход применяется в Oracle, Db2, Microsoft SQL Server, PostgreSQL.

Индекс по первичному ключу строится на неупорядоченной таблице всегда, поскольку при вставке новой записи СУБД обязана проверить уникальность значения ключевых полей, а сделать это быстро можно, только имея индекс.

Второй подход используется в SQLite и MySQL с движком InnoDB. Каждая таблица в InnoDB имеет кластеризованный индекс, в котором и хранятся все данные таблицы. Если его не создать явно, уникальное поле и кластеризованный индекс по этому полю будут созданы автоматически.

---

[1] Если принять, что размер идентификатора блока (указателя на поддерево) равен 8 байт, размер ключа — 30 байт, а размер страницы — 8 КБ, то на втором уровне B-дерева может разместиться 200 страниц (узлов), на третьем — 40 000 страниц, на четвёртом — 8 000 000 страниц, то есть около 60 ГБ или более 300 млн объектов размером 200 байт.

Возможен второй подход и в других платформах:

- Oracle позволяет создать таблицу, организованную как B-дерево (index-organized table);
- Microsoft SQL Server позволяет определить на таблице кластеризованный индекс; в этом случае значения всех полей таблицы хранятся непосредственно в B-дереве, как в InnoDB.

На таблице, хранимой в B-дереве, также можно строить вторичные индексы, то есть индексы по полям, не входящим в первичный ключ. В этом случае во вторичном индексе в качестве указателя на данные будет использован первичный ключ, т. к. в B-дереве нет понятия физического адреса строки[1].

Несмотря на то что значение ключа в B-дереве уникально, индекс на основе B-дерева может быть и неуникальным. Физически он ничем не отличается от уникального, просто идентификатор строки является в нём частью ключа.

B-дерево всегда сбалансировано благодаря особенностям алгоритмов вставки и удаления.

При вставке ключа алгоритм спускается до листового узла, в который должен попасть ключ. Если после вставки узел оказывается заполнен, то алгоритм делит его пополам: половину ключей оставляет в старом узле, а половину переносит в новый узел. Ключ, по которому происходит разделение узлов, переносится в узел-предок. Если узел-предок заполнен, то операция повторяется — узел-предок делится пополам, а ключ, по которому произошло разделение, переносится на уровень выше. Если в конце концов оказывается, что заполнен корневой узел, то он тоже делится пополам, а у дерева появляется новый корень, в котором оказывается единственный ключ.

Некоторые реализации B-дерева (т. н. B^*-деревья) вместо безусловного разделения узла пробуют переносить часть ключей в соседний узел. За счёт этого уменьшается производительность обновления, но увеличивается плотность хранения данных и, соответственно, производительность поиска.

При удалении ключа из дерева узел может оказаться полупустым, и в этом случае алгоритм может попытаться объединить узел с соседним и удалить разделяющий ключ из узла-предка — и так далее до корня.

Узел B-дерева делится только после того, как в него будет записано максимально возможное количество ключей. Авторы B-дерева называют «полупустым»

[1] Такое понятие можно ввести, но оно бессмысленно, т. к. этот адрес может измениться при любой операции, даже не затрагивающей данный ключ.

узел, заполненный менее чем наполовину (или менее чем на $^2/_3$ для B*-дерева). Промышленные СУБД позволяют более гибко управлять наполнением узлов (каждый из которых размещается в выделенной дисковой странице):

- Oracle позволяет задать параметры `PCTFREE` и `PCTUSED`. При первоначальном построении индекса в узлах будет оставлено `PCTFREE` процентов свободного пространства, чтобы вставки не приводили сразу же к делению узлов. Узел будет объединяться с соседними только после того, как его заполнение опустится ниже `PCTUSED`.
- Db2 позволяет задать отдельно параметры `PCTFREE` и `LEVEL2 PCTFREE` — первый распространяется на все узлы, а второй — на промежуточные узлы, то есть на узлы, кроме корня и листьев. Параметр, показывающий минимальный уровень заполнения, после которого узел будет объединён с предыдущим, называется `MINPCTUSED`.
- PostgreSQL и Microsoft SQL Server позволяют задать параметр `FILLFACTOR`. При построении нового индекса и при расширении индекса вправо (то есть при добавлении максимальных значений) узлы будут заполняться, пока в них не будет занята указанная (в процентах) доля пространства. Порог заполнения, после которого полупустые узлы будут объединены, эти СУБД определяют самостоятельно[1].

СУБД выделяет для деревьев сегменты так же, как и для неупорядоченных таблиц, и точно так же управляет свободным пространством в этих сегментах.

Для B-деревьев, как и для неупорядоченных таблиц, может применяться словарное сжатие, при котором вместо пар «ключ—значение» в листьях дерева хранятся пары «ключ — массив значений».

Все рассуждения про параметры хранения неупорядоченных таблиц справедливы и для B-деревьев: запас свободного пространства в узлах положительно сказывается на производительности вставки и обновления, а плотно заполненные узлы и сжатие повышают скорость поиска и экономят дисковое пространство, но отрицательно сказываются на скорости изменений.

Другие страничные структуры

Структура подавляющего большинства объектов в страничных СУБД — это именно неупорядоченные таблицы и B-деревья. Во всяком случае, автору неизвестны случаи, когда в качестве основной структуры хранения используется что-то

[1] Например, PostgreSQL вообще не объединяет узлы.

другое. Однако есть и несколько менее распространённых структур, используемых для индексирования.

СУБД PostgreSQL весьма богата индексными структурами на базе **сильно ветвящихся деревьев**, где под каждый узел выделяется страница:

- GiST (Generalized Search Tree) — «каркас» сильно ветвящихся деревьев, позволяющий создавать любые индексы, задавая типы индексируемых значений и правила их сравнения.
- R-дерево (R-tree) — дерево, где элементом хранения является не скалярное значение, а точка в n-мерном пространстве. Применяется для индексирования геопространственных данных.
- GIN (Generalized Inverted Index) и RUM (игра слов, при которой GIN превращается в напиток джин, а тот, в свою очередь, — в ром). Инвертированные индексы, в основе которых лежит B-дерево, а в листовых узлах вместо одного идентификатора строки — список идентификаторов, который при большом объёме сам может быть представлен в виде дерева.

Детальное описание этих структур выходит за рамки книги.

Индексы на основе битовых карт (bitmap indexes) используются в Oracle. Индекс представляет собой дерево, в корневом узле которого размещены ключи и ссылки на дочерние узлы, как и в B-дереве, а вот устройство листовых узлов отличается. В листовых узлах содержится информация вида «значение, начало диапазона rowid, конец диапазона rowid, карта».

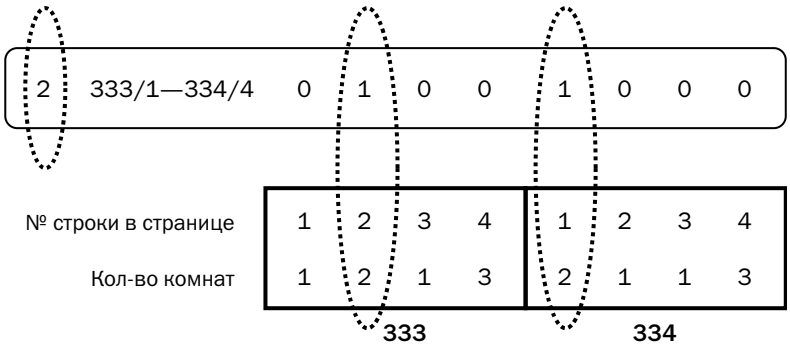
Пусть, например, у нас есть неупорядоченная таблица, состоящая из двух следующих страниц:

Страница	Номер в странице	Id	Адрес	Кол-во комнат
333	1	56124	ул. Лесная, д. 13, кв. 916	1
333	2	56209	ул. Садовая, д. 13, кв. 1531	2
333	3	57001	ул. Огородная, д. 13, кв. 2048	1
333	4	56328	ул. Полевая, д. 13, кв. 1014	3
334	1	63421	ул. Грядочная, д. 13, кв. 1311	2
334	2	58651	ул. Бахчевая, д. 13, кв. 2752	1
334	3	56434	ул. Луговая, д. 13, кв. 1001	1
334	4	56811	ул. Болотная, д. 13, кв. 988	3

В этом случае индекс, построенный по полю «количество комнат», будет выглядеть так:

Значение	Rowid от	Rowid до	Карта
1	333/1	334/4	10100110
2	333/1	334/4	01001000
3	333/1	334/4	00010001

В каждой карте установленный бит соответствует строке, в которой ключ (индексированное поле) равен значению, соответствующему карте:



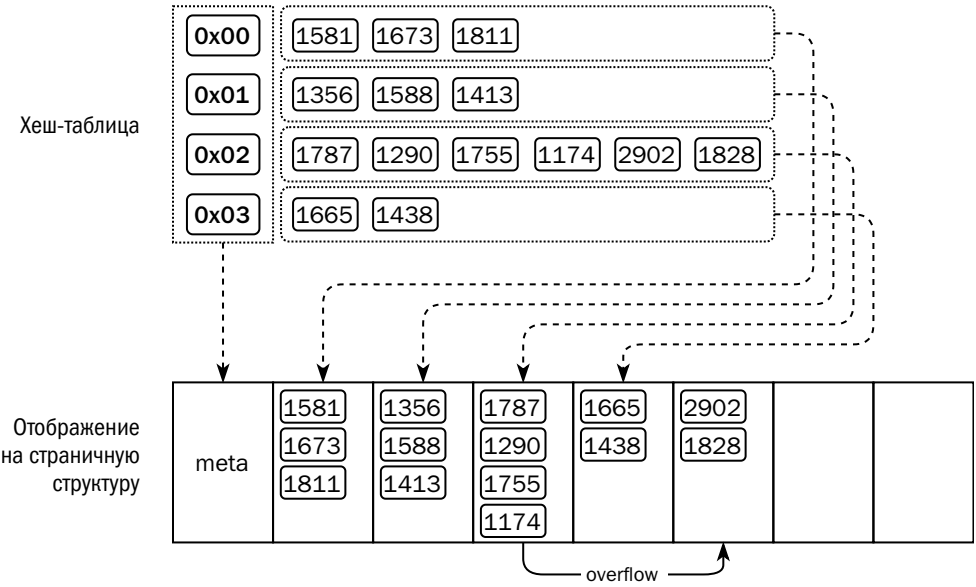
Чем больше уникальных значений в индексируемом поле, тем больше карт в индексе. Обновление данных приводит к обновлению всех карт, относящихся к диапазону, в который попадает обновляемая запись, а удаление приводит к перестройке карт. Поэтому битовый индекс подходит для индексирования полей с низкой кардинальностью, таких как пол или цвет глаз, и только в базах, где обновления крайне редки или отсутствуют, — в основном это аналитические базы.

Несколько битовых индексов на одной таблице позволяют очень эффективно искать строки по набору признаков: чтобы найти набор строк, удовлетворяющих сразу нескольким условиям, надо выполнить логические операции «и» или «или» над битовыми картами соответствующих индексов.

Индексы на основе хеш-таблицы (hash indexes) используются в PostgreSQL. Как известно, хеш-таблица представляет собой массив значений хеш-функции, каждое из которых ссылается на список ключей, хеш-функция от которых принимает соответствующее значение.

Вот как эта структура данных ложится на страничную структуру дискового хранилища:

- первая страница содержит метаинформацию об индексе, в частности количество различаемых значений хеш-функции — корзин (buckets);
- под каждую корзину выделяется страница, содержащая ключи и физические идентификаторы строк, в которых индексируемое поле принимает нужное значение;
- если корзина не помещается в страницу, то часть ключей переносится в дополнительные страницы (overflow pages).



Описанная структура сохраняет все достоинства и недостатки хеш-таблицы: с одной стороны, — постоянное время поиска, не зависящее от количества строк в индексированной таблице, с другой — невозможность поиска по диапазону ключей.

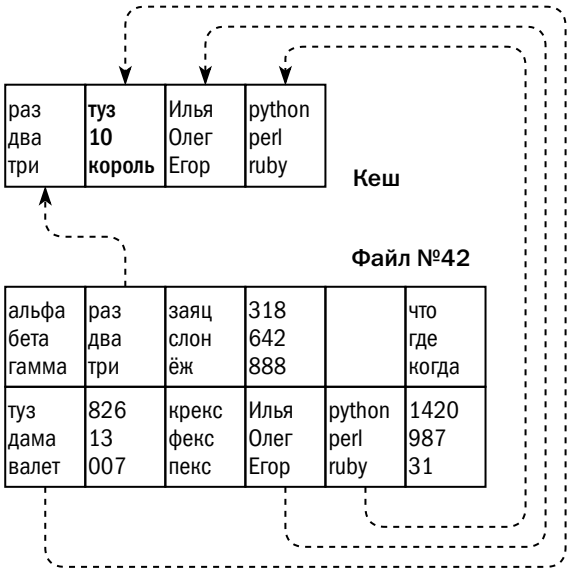
Буферный кеш

Как мы уже говорили выше, минимальный объём обмена СУБД с диском — страница. Если бы для извлечения данных страницы каждый раз читались с диска, а при изменении — немедленно записывались на диск, работа с такой базой данных была бы невозможна из-за длительного ожидания ввода-вывода,

а мультипликатор чтения и записи достигал бы нескольких сотен (например, при размере записи 50 байт и размере страницы 8 КБ). Для повышения эффективности все платформы со страничной организацией памяти используют буферный кеш (buffer cache, или buffer pool).

Кеш является частью экземпляра и располагается в разделяемой области памяти, доступной всем процессам, образующим экземпляр.

Кеш разделён на буферы, размер которых равен размеру страницы БД. В служебной области хранится информация о том, какие страницы данных хранятся в кеше: физический адрес страницы, время последнего доступа к странице и признак «грязной» (dirty) страницы, показывающий, что страница была изменена в памяти, но ещё не сохранена на диск.



Служебная область

buffer #	page	access	dirty
0	42.001	128391	
1	42.006	125313	yes
2	42.009	127837	
3	42.010	126530	

Если пользовательскому запросу необходимо обратиться к странице с данными, он действует так:

- ищет страницу в кеше;
- если страница найдена, то блокирует её, иначе:
  - находит в кеше свободный буфер или буфер, который можно освободить;
  - считывает страницу с диска, помещает её в найденный буфер, записывает идентификатор таблицы в служебную область и тут же блокирует страницу;

- считывает или изменяет данные — в зависимости от запроса;
- если страница в кеше была изменена, то помечает её как грязную;
- снимает блокировку со страницы, разрешая другим запросам доступ к ней.

Частота попадания запросов в кеш (cache hit ratio) считается одним из важнейших показателей правильной настройки экземпляра. Интуитивно кажется, что чем больше кеш, тем быстрее будет работать система, однако это не всегда так: при современных объёмах оперативной памяти и скорости дисков возможна ситуация, когда поиск в кеше занимает столько же времени, сколько считывание страницы с диска, а дальнейшее увеличение объёма кеша приведёт к снижению производительности.

Периодически кеш переполняется, и системе приходится вытеснять страницы с наименьшим рейтингом. Логика расчёта рейтинга страницы отличается от платформы к платформе. Один из часто используемых алгоритмов — вытеснение страниц, которые дольше всего не использовались (Least Recently Used, LRU). Детальное описание алгоритмов кеширования выходит за рамки книги.

На рисунке выше чем больше значение в колонке access, тем нужнее страница в кеше. Если, к примеру, запросу понадобится страница 42.000, то будет освобождён буфер 3: наименьший рейтинг у буфера 1, но в нём находится грязная страница, которую нельзя вытеснить из памяти, не сохранив предварительно на диск.

Операция обращения к кешированной странице достаточно коротка и сравнима по времени выполнения с обращением к объектам синхронизации операционной системы (семафорам и критическим секциям). В связи с этим блокировку страницы в кеше реализуют как переменную в памяти, которую запросы изменяют при помощи атомарной операции test-and-set. Такую реализацию блокировки называют «защёлкой» (latch), или «циклической блокировкой» (spin-lock). Если запросу не удалось заблокировать страницу, изменив значение переменной, он повторяет попытку блокировки несколько раз, потом засыпает на некоторое время и вновь повторяет попытку блокировки. Таким образом конкуренция нескольких параллельных запросов за одну и ту же страницу может приводить к большим непродуктивным затратам процессорного времени.

В зависимости от платформы процессы, обслуживающие запросы пользователя, могут как самостоятельно записывать грязные страницы на диск, так и полностью переложить эту задачу на служебные процессы экземпляра, эффективно планирующие запись. Надёжность хранения изменений обеспечивается журналом базы данных.

3.3. Неизменяемые структуры

LSM-дерево

LSM-дерево (Log-Structured Merge-tree) — структура данных, предложенная в 1996 году группой учёных Массачусетского университета во главе с Патриком О'Нейлом (Patrick O'Neil).

В основе этой структуры лежат следующие принципы:

1. Данные записываются на диск один раз и впоследствии не изменяются. Это позволяет писать данные последовательно, что существенно повышает скорость записи. Кроме того, такой подход минимизирует накладные расходы на запись и хранение, т. к. данные могут быть упакованы максимально плотно и даже сжаты.
2. У каждого элемента данных есть ключ, и данные на диске отсортированы по значению ключа. Поскольку перезаписывать данные нельзя, первичное накопление и сортировка данных происходят в памяти.
3. Дерево состоит из множества файлов, каждый из которых содержит операции за определённый промежуток времени. Данные с одним и тем же ключом могут встречаться в нескольких файлах. Фоновые процессы объединяют файлы, выбрасывая из них неактуальные значения данных. В результате объединения получаются новые файлы, которые заменяют собой файлы, подвергшиеся объединению.

Рассмотрим работу LSM-дерева на примерах.

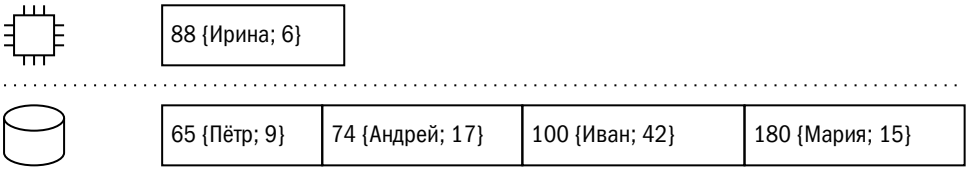
Под данные, сохраняемые в структуре на основе LSM-дерева, выделен буфер в оперативной памяти, и пока в буфере есть свободное место, данные сохраняются только в нём. Надёжность буфера в оперативной памяти обеспечивается журналом. Буфер образует нулевой уровень дерева L_0. Данные в буфере отсортированы — для этого может использоваться любая структура, например двоичное дерево или B-дерево:

65 {Пётр; 9}	74 {Андрей; 17}	100 {Иван; 42}	180 {Мэри; 14}

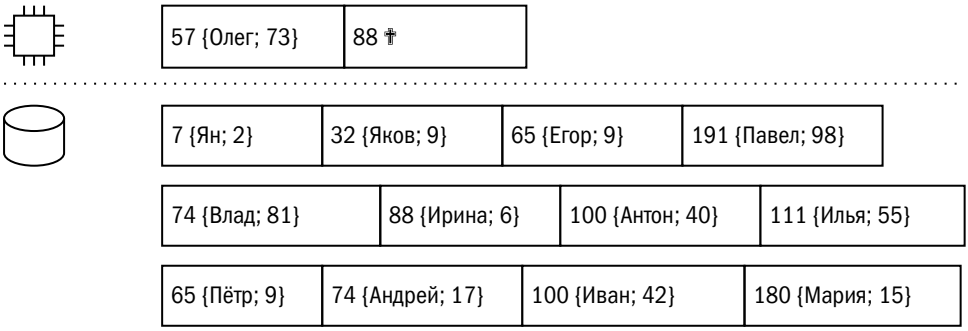
Если какое-то значение изменяется и соответствующий ключ есть в буфере, то изменения вносятся в содержимое буфера:

65 {Пётр; 9}	74 {Андрей; 17}	100 {Иван; 42}	**180 {Мария; 15}**

Как только буфер заполнен (или по прогнозам СУБД оставшегося места при текущей нагрузке не хватит на время, необходимое для записи буфера на диск), данные из буфера переносятся на диск, и файлы, содержащие эти данные, образуют первый уровень — L_1:

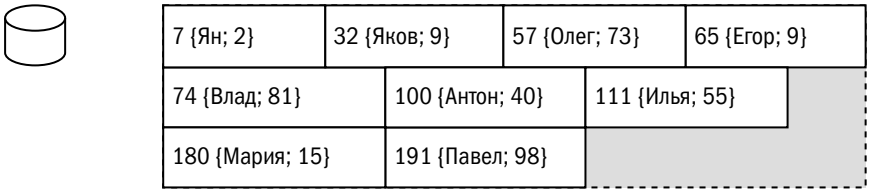


Любое изменение существующих данных, как и новые данные, пишется в буфер в памяти, и когда буфер в очередной раз переполняется, он вытесняется в новый файл. Чтобы получить значение по ключу, необходимо поискать его в памяти; если ключ не найден в памяти, СУБД ищет его сначала в последнем по порядку записи файле, затем, в случае неуспеха, — в предпоследнем, и так далее, пока ключ не будет найден или пока не будут прочитаны все файлы:



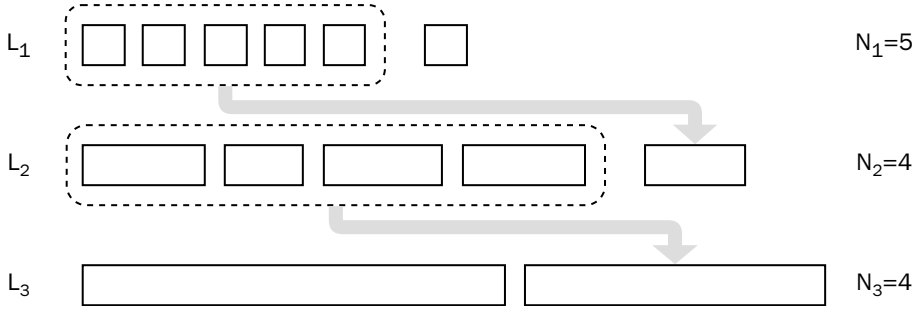
Удалять ключи из буфера нельзя, поскольку если читающий сеанс не найдёт ключ в буфере, он будет искать его в файле и может найти старое значение. Поэтому вместо удаления ключа создаётся специальный маркер удаления (tombstone).

Ключи в файлах могут повторяться, и для того чтобы уменьшить занимаемое пространство, файлы данных периодически объединяются: фоновый процесс считывает ключи из нескольких файлов и создаёт новый файл. Если у одного и того же ключа оказывается несколько значений, то остаётся только последнее, а если последним значением был маркер удаления и текущий уровень — самый нижний (последний), то ключ удаляется из нового файла.



Политика объединения файлов зависит от конкретной платформы.

Базовая политика объединения основывается на физическом размере файлов: когда на первом уровне создаётся N_1 файлов, эти файлы объединяются в файл второго уровня. И далее, когда на уровне L_i создаётся N_i файлов, они объединяются в файл уровня L_{i+1}:

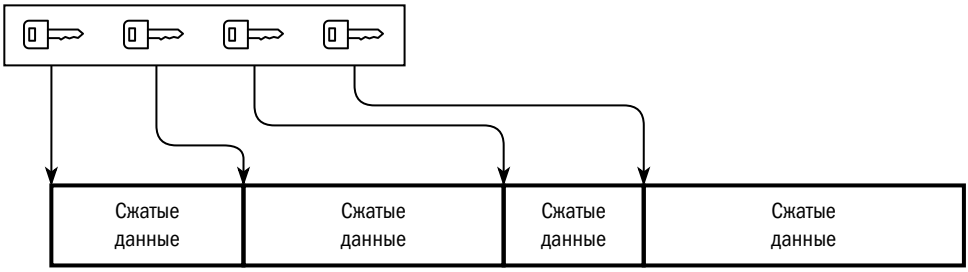


Уровень файла определяется его размером: чем больше файл, тем больше (ниже) его уровень. Если диапазоны ключей в объединяемых файлах пересекаются, то новый файл может остаться на том же уровне или даже переехать на уровень вверх.

Изменяя настройки LSM-дерева, можно управлять его формой. Чем больше соотношение между размерами файлов на уровнях, тем меньше количество уровней и быстрее поиск за счёт меньшего количества просматриваемых файлов, но больше расходы на объединение.

LSM-дерево оптимизировано для записи, а поиск данных в LSM-дереве — гораздо более медленная операция. Если повезёт, то ключ будет найден в оперативной памяти, но в худшем случае при поиске будут просмотрены все файлы. Худший случай встречается не так редко, как кажется: например, все файлы необходимо просмотреть, если запрашивается диапазон ключей или если необходимо убедиться в отсутствии ключа перед записью уникального значения.

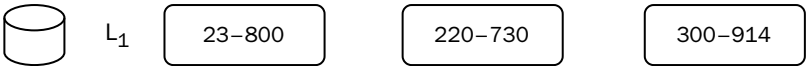
Для оптимизации поиска применяют индексные файлы. Индексный файл содержит ключи и смещения в основном файле, начиная с которого располагаются данные с такими ключами:



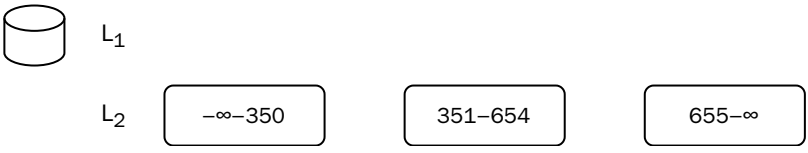
Большая проблема LSM–дерева — появление огромных файлов на нижних уровнях. Для объединения двух файлов необходимо иметь на диске свободное пространство, равное сумме размеров этих файлов, — на случай, если все ключи в двух файлах разные. То есть для объединения двух терабайтных файлов на диске должно быть 2 ТБ свободного пространства. Мало того, если все ключи в двух файлах разные, то объединение этих файлов лишено смысла.

Чтобы избежать бесполезной работы по объединению, данные на каждом уровне делят на диапазоны. Файлы объединяются только в том случае, если диапазоны ключей в этих файлах пересекаются, а если размер получившегося файла выходит за установленные рамки, то диапазон разбивается на два новых.

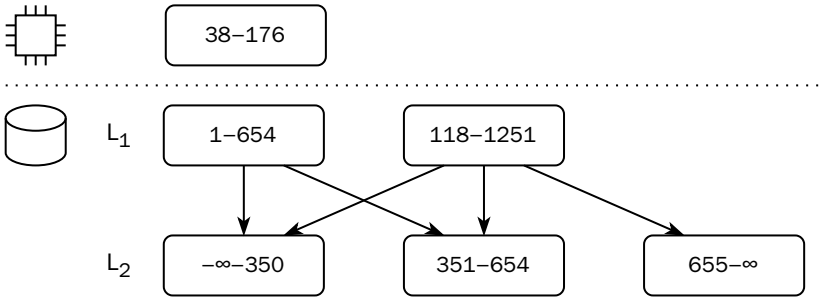
Пусть на первом уровне скопились файлы и пришло время их объединить:



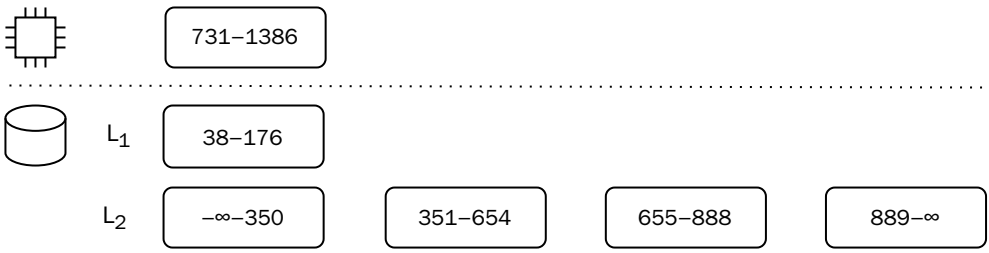
Приложение работает с ключами в произвольном порядке, поэтому файлы первого уровня могут содержать любые ключи. Фоновый процесс объединяет их и перемещает на второй уровень независимо от размера. Файлы второго уровня уже разбиты по диапазонам ключей:



Пусть теперь на первом уровне накопились новые файлы. Диапазоны ключей в этих файлах могут пересекаться с диапазонами произвольного количества файлов второго уровня. Вместо того чтобы объединять файлы первого уровня между собой, фоновый процесс вливает каждый файл первого уровня в каждый файл второго уровня с пересекающимся диапазоном ключей:



Если файл второго уровня становится слишком велик, он может быть разбит на два (например, в результате объединения последний файл был разделён по ключу 888):



Когда размер уровня превышает ограничение, файлы объединяются, образуя на следующем уровне файлы большего размера. Если следующий уровень не пуст, то файлы вливаются в файлы следующего уровня, с которыми у них пересекаются диапазоны ключей.

При такой стратегии объединения размер файлов уменьшается за счёт увеличения их количества. К счастью, современные файловые системы способны хранить десятки тысяч файлов в одном каталоге без заметного влияния на производительность.

Фильтр Блума

Фильтр Блума (Bloom filter) — вспомогательная структура, позволяющая не сканировать файл, если искомого ключа в нём гарантированно нет. Это самостоятельная структура данных, но она часто используется именно в связке с LSM-деревьями, где иногда в поисках ключа приходится просматривать множество файлов.

Чтобы построить фильтр, берётся хеш-функция небольшой мощности, и выделяется карта размером во столько битов, сколько разных значений может принимать функция. Так, например, для CRC16 необходим массив размером 8 КБ.

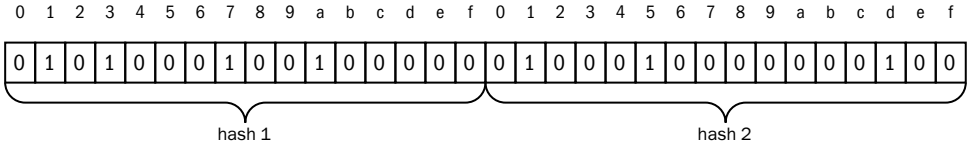
Для каждого ключа в файле рассчитывается значение хеш-функции, и в карте устанавливаются биты, номера которых равны полученным значениям. Как правило, чтобы уменьшить количество коллизий, берётся несколько хеш-функций (например, CRC16 с разными полиномами) и несколько карт (либо биты для значений всех хеш-функций устанавливаются в одной карте). При проверке наличия ключа в файле для этого ключа рассчитывается значение всех хеш-функций, и если хотя бы один из битов с полученными номерами не установлен, то ключа в файле гарантированно нет.

Возьмём для примера две 4-битные хеш-функции. Фильтр Блума с этими функциями будет занимать четыре байта (2×2^4 бит). Значения функций приведены в таблице:

Ключ	Хеш 1	Хеш 2
foo	3	5
bar	0	9
joe	1	d
ttt	6	1
wow	7	5
ham	4	9

Ключ	Хеш 1	Хеш 2
pty	5	d
gwm	a	1
bmw	b	5
vag	8	5
zaz	9	1
qty	a	d

Если в файле находятся ключи foo, joe, wow и gwm, то фильтр Блума будет выглядеть так:



Попытаемся найти, например, ключ pty. Мы видим, что бит, соответствующий значению первой хеш-функции, 0x05, не установлен в первом массиве, а значит, и ключа в файле нет. Но для ключа qty оба бита — и 0x0a в первом массиве, и 0x0d во втором — установлены, несмотря на то что самого ключа в файле нет. На практике обычно используется 3–5 массивов, что даёт крайне низкую вероятность ложноположительного срабатывания.

Такие структуры, как фильтр Блума, называются вероятностными (probabilistic data structure). Они, требуя сравнительно небольших объёмов памяти, позволяют быстро получить приблизительную, но достаточно точную оценку результата. Наиболее известные вероятностные структуры — HyperLogLog, позволяющий оценить количество уникальных элементов в больших наборах данных, и Count-Min Sketch, позволяющий оценить частоту вхождения элементов в потоке данных.

Механизмы хранения на основе LSM-деревьев

Полноценный механизм хранения данных (движок, storage engine) на основе LSM-деревьев включает в себя несколько компонентов:

- хранилище данных в памяти (memtable);
- собственно код управления LSM-деревом;
- фильтры Блума;
- кеш значений, то есть хранилище конкретных элементов данных, независимо от того, где находятся их актуальные копии;
- кеш блоков, то есть хранилище индексов и фрагментов файлов LSM-дерева.

Первой СУБД, основанной на LSM-деревьях, была Google BigTable, а затем компания Google выпустила под открытой лицензией движок LevelDB. Сегодня самый популярный движок на LSM-деревьях — RocksDB.

Движок	Авторы	Использование
BigTable	Google	BigTable
LevelDB	Google	Google Chrome, Riak KV
RocksDB	Facebook	YugabyteDB, Dgraph, Nebula Graph, ArangoDB Внутренние проекты LinkedIn, Facebook, AirBnB, Uber
WiredTiger[1]	WiredTiger Inc.	MongoDB
Cassandra engine	Facebook, DataStax	Apache Cassandra/DataStax Enterprise
Vinyl	Mail.Ru, picodata.io	Tarantool/Picodata
*MergeTree	Яндекс	ClickHouse
Pebble	Cockroach Labs.	CockroachDB
ScyllaDB engine	ScyllaDB Inc.	ScyllaDB

[1] WiredTiger поддерживает LSM-деревья в качестве основного хранилища и B-деревья для индексов.

У движков, перечисленных в таблице, есть ряд архитектурных различий:

- Гарантии синхронизации. Движок может гарантировать безопасный доступ из единственного потока или из нескольких потоков одного процесса.
- Структура данных для хранилища в памяти. Популярнее всего список с пропусками (skiplist), но возможны и другие реализации — например, несортированный массив (одна из опций RocksDB) или B^{+*}-дерево (vinyl).
- Алгоритмы сжатия данных. Наиболее популярны библиотеки zstd, zlib, snappy.
- Детали устройства вторичных индексов.

Прочие неизменяемые структуры

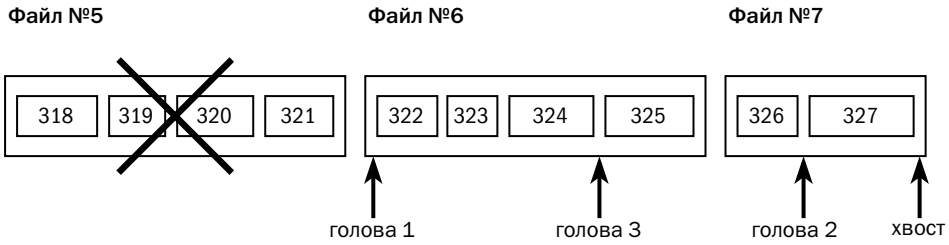
Движки на основе LSM-деревьев чрезвычайно распространены, однако есть и другие неизменяемые структуры хранения данных на диске.

Журнал (log) — структура, данные в которую пишутся без сортировки в порядке поступления. Журнал характеризуется наибольшей из возможных производительностью записи и применяется в случаях, когда нет необходимости поиска в сохранённых данных (то есть данные читаются целиком) или когда их объём невелик. Примерами использования могут служить движки семейства Log в СУБД ClickHouse или метод хранения «recno» в движке Berkeley DB, используемый наряду с неупорядоченной таблицей и B-деревьями.

Движки Bitcask (Riak KV) и Aerospike также построены на основе журнала: данные записываются в несортированные файлы, а фоновый процесс объединяет эти файлы, удаляя неактуальные версии записей. Для быстрого поиска записей СУБД поддерживает в памяти индекс на основе хеш-таблицы. По сравнению с движком на основе LSM-дерева у этого движка вдвое меньший мультипликатор записи, поскольку журнал есть и там, и там, но у самого журнала никаких дополнительных структур нет. При этом фоновый процесс объединения файлов создаёт гораздо большую нагрузку на процессор, чем процесс сжатия LSM-дерева, поскольку ему требуется сортировать записи для поиска дубликатов. Кроме того, индекс требует большого объёма оперативной памяти.

Очередь (queue) — структура, похожая на журнал, но предназначенная для последовательного доступа с удалением прочитанных элементов. Представляет собой последовательность несортированных файлов (журналов). Пишущие транзакции добавляют данные в «хвост», то есть в конец текущего файла, создавая новый файл при достижении какого-либо условия — превышения размера,

достижения определённого количества записей или истечения времени. Читающие транзакции считывают файлы из другого конца, с «головы», продвигая указатель на «голову» вперёд, причём этот указатель у каждого читающего процесса или группы процессов может быть свой. После того как все указатели «головы» уходят за пределы очередного файла, файл удаляется с диска:

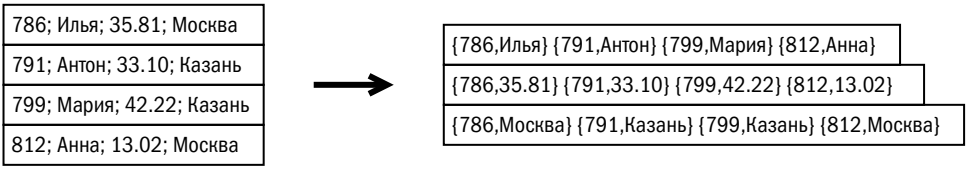


Очередь реализована в методе хранения «queue» движка Berkeley DB, а также в брокерах очередей сообщений — например, Apache Kafka.

3.4. Колоночное хранение

Концепция и история

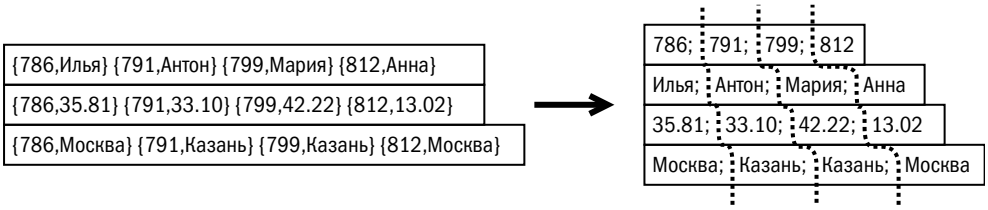
Во всех структурах, рассмотренных до этого момента, строка в БД хранилась как единое целое. В аналитических СУБД широко применяется другой подход к хранению данных — по колонкам:



Основная цель такого подхода — ускорение аналитических запросов, читающих отдельные колонки вместо всего массива. Очевидно, что значительное количество данных при выполнении таких запросов просто не нужно читать. Если в примере выше требуется посчитать средний рейтинг (третья колонка), то в колоночном представлении достаточно будет прочитать только одну из трёх частей.

В структуре, изображённой на рисунке, принадлежность значения колонки конкретной записи определяется значением первичного ключа. В современных

СУБД чаще применяется другой приём: значения атрибутов отсортированы в одном и том же порядке, и принадлежность значения конкретной записи определяется его индексом в массиве значений.



Первая система, хранящая данные по колонкам, TAXIR, была разработана в 1969 году, а первыми платформами общего назначения, использующими колоночную структуру данных, стали Sybase IQ (сейчас — SAP IQ) в 1995 году и SAP BW в 1997 году. Бурное развитие СУБД с колоночным хранением началось в середине 2000-х с началом университетских проектов C-Store и MonetDB. Из C-Store впоследствии выросла компания Vertica, которая поддерживает одноимённую платформу (ныне принадлежит компании Micro Focus). MonetDB дала начало платформе VectorWise, ныне известной как Actian Vector.

Сегодня на рынке присутствует множество платформ, использующих колоночный подход к хранению, — как дисковые (Vertica, SAP IQ, MonetDB, ClickHouse, Apache Druid), так и СУБД в памяти (Exasol, SAP HANA). Кроме того, движки для колоночного хранения появились в платформах, изначально ориентированных на строчное хранение (Greenplum, Teradata, Db2, Microsoft SQL Server, SingleStore).

На русских форумах, посвящённых базам данных, встречается термин «колоночная БД», под которым авторы могут иметь в виду как «реляционную БД с колоночным хранением» (column-oriented RDBMS), про которые мы говорим здесь, так и «хранилища семейств колонок» (column family store). Будьте внимательны!

Для хранения фрагментов, представляющих собой данные отдельных колонок, могут использоваться любые структуры:

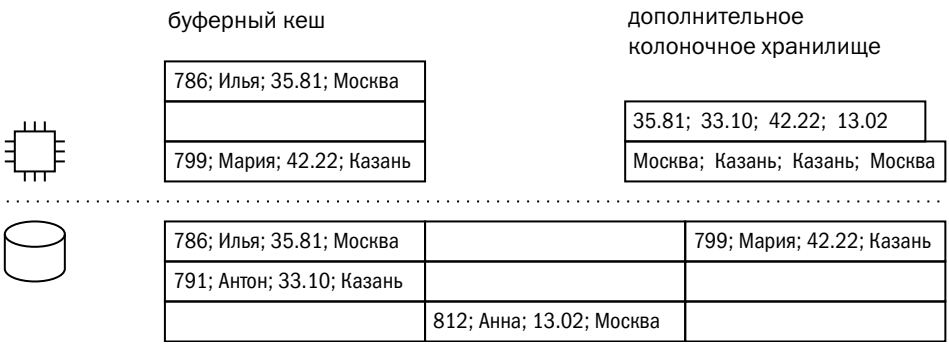
- СУБД ClickHouse, разработанная компанией Яндекс, использует для хранения колоночных данных структуры на основе журнала или LSM-дерева;
- Vertica использует набор файлов для каждой проекции (набора колонок), причём эти файлы могут храниться в объектном хранилище, поддерживающем интерфейс S3[1];

[1] Simple Storage Service, объектный API доступа к дискам поверх http, разработанный компанией Amazon.

- в проекте C-Store использовался движок Berkeley DB на основе LSM-дерева;
- Microsoft SQL Server (для индексов columnstore), Db2 (для таблиц с опцией `ORGANIZE BY COLUMN`) и SAP IQ используют блочные структуры, где под данные каждой колонки выделяются собственные экстенты (при этом экстент рассматривается как сплошная область хранения, не разделённая на страницы):

- в MonetDB используются массивы в памяти, отображаемые на файлы при помощи механизма memory-mapped files;
- экспериментальный движок Zedstore, обеспечивающий колоночное хранение для PostgreSQL, хранит колонки в виде B-деревьев: каждой записи при вставке присваивается последовательно возрастающий внутренний идентификатор, по которому упорядочены значения атрибутов в каждом дереве.

Некоторые СУБД используют колоночные структуры не как основное хранилище, а как дополнительное представление данных, хранящихся построчно. Так, например, дополнение Oracle In-memory Option, доступное начиная с версии 12cR1, создаёт отдельную область в памяти, вне буферного кеша (in-memory columnar storage). В этой области данные, загружаемые из обычных таблиц, сохраняются в колоночном формате. Система не отказывается и от буферного кеша, где страницы по-прежнему хранятся в том же формате, что и на диске:



Microsoft SQL Server начиная с версии 2016 поддерживает «колоночные индексы» (columnstore index). В этих индексах сохраняется часть колонок обычной таблицы, хранимой построчно в виде неупорядоченной таблицы или B-дерева. Можно объявить колоночный индекс кластеризованным — в этом случае он станет основным местом хранения данных.

Ещё одно любопытное решение применено в программно-аппаратном комплексе Oracle Exadata. В качестве системы хранения данных в нём использованы «ячейки хранения» (Exadata storage cells) — серверы с внутренними дисками и специальным программным обеспечением — Exadata software. Данные на дисках хранятся в традиционном для Oracle строчном формате, но при определённых видах запросов ячейки могут возвращать серверу БД значения только требуемых колонок.

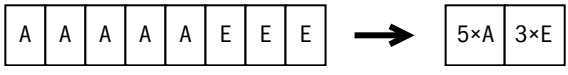
Оптимизация доступа

Значения, хранимые в колонке, имеют одинаковый тип. Если речь идёт о числах, то и размер значений одинаков, что позволяет использовать для их обработки SIMD-инструкции[1] современных процессоров. Так, например, в SAP IQ до версии 16, вышедшей в 2013 году, каждая страница данных была построена в виде массива ячеек фиксированного размера, и все значения всех колонок имели один и тот же тип данных — «число».

Очевидно, что данные одной колонки гораздо более однородны, чем данные разных колонок одной и той же строки, поэтому колоночное представление гораздо эффективнее сжимается. В движках применяются различные механизмы сжатия, часть из которых сохраняет возможность обработки данных без разжатия, а у части механизмов сжатые значения имеют одинаковый размер. Фиксированный размер важен для ускорения обработки данных за счёт использования SIMD-инструкций процессора и упрощения поиска элемента по номеру позиции.

Рассмотрим наиболее популярные алгоритмы сжатия.

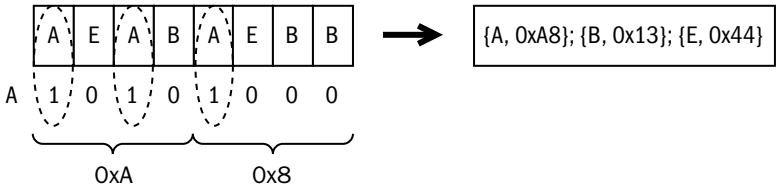
- **Кодирование повторов (RLE — run-length encoding)**.Если встречается серия одинаковых значений, то она заменяется на длину серии и значение:



[1] Single instruction, multiple data — инструкции, одновременно обрабатывающие несколько однотипных значений. В процессорах семейства x86 реализованы SIMD-расширения MMX, SSE, AVX, 3DNow! и др.

При вычислении агрегатных функций (sum, avg и других) можно использовать эти данные непосредственно, без распаковки.

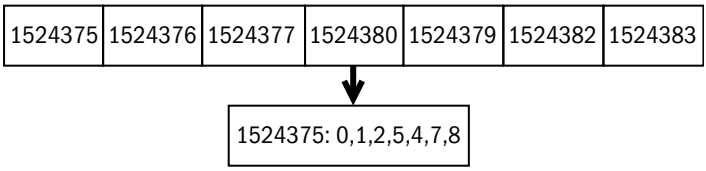
- **Битовый вектор (bit-vector)**. Этот метод кодирования похож на битовые индексы в Oracle и используется для колонок, содержащих малое количество различных значений. В файле данных помещается список значений, а затем для каждого значения создаётся битовая карта, где единицей отмечены позиции, в которых встречается это значение:



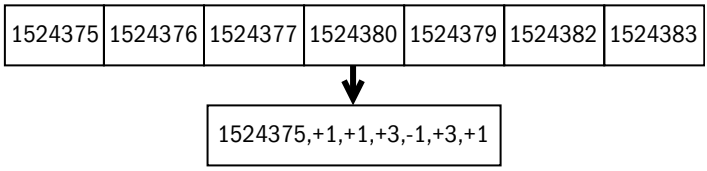
- **Словарное кодирование с фиксированным размером (fixed width dictionary)**. Для колонок с повторяющимися значениями создаётся словарь, и значения заменяются на позицию в словаре:

Код Хаффмана более компактен (на рисунке выше значение «Москва» с точки зрения объёма хранения было бы выгоднее закодировать одним битом), но нередко компактность приносится в жертву фиксированному размеру.

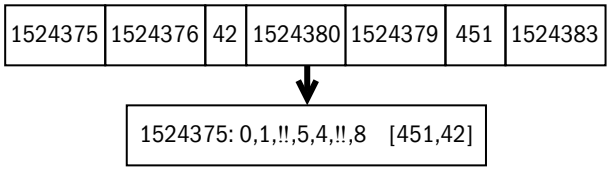
- **Подвижная система отсчёта (frame-of-reference)**. Если значения в колонке близки, система сохраняет базовое значение («точку отсчёта») и разницу текущего значения с базовым:



- Вариант подвижной системы отсчёта — **разностное кодирование (delta coding)**, когда вместо числа сохраняется его разница с предыдущим числом:



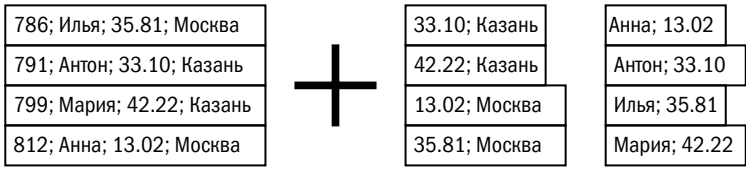
- **Заплатки (patching)**. Это приём, применяемый совместно с другими схемами. Если какое-то значение резко выбивается из общего ряда, то вместо него ставится признак исключения, а в конце в обратном порядке сохраняется список исключений. Порядок обратный, т. к. список заполняется начиная с конца блока данных:



Разумеется, описанные приёмы не отменяют применения в некоторых случаях алгоритмов общего назначения — например, алгоритмов семейства LZ или кодирования Хаффмана. Решение о выборе алгоритма принимается на основании множества параметров — типа данных колонки, количества различных значений и порядка сортировки данных в представлении. Сжатие может применяться как на уровне всей колонки, так и на уровне элемента хранения — файла или страницы.

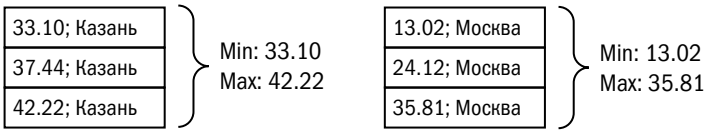
Если преимущества колоночного хранения при чтении небольшого количества колонок очевидны, то чтение всей таблицы может быть существенно медленнее, чем чтение такой же таблицы, хранимой по строкам. Замеры, сделанные в начале 2000-х годов, показывали отставание на полной таблице от двух до шести раз за счёт накладных расходов на восстановление записей из нескольких частей. Современные колоночные платформы даже при столь неблагоприятных условиях тестирования проигрывают традиционным строчным базам не более 20–30 %.

Один из методов ускорения чтения данных — проекции, используемые, например, в СУБД Vertica. Проекцией называется файл, содержащий набор колонок. У одной и той же таблицы может быть несколько проекций, причём по-разному отсортированных. Как правило, есть и «основная» проекция, содержащая все колонки.



На рисунке выше мы видим три проекции, первая из которых отсортирована по ключу, вторая — по местоположению и затем по сумме, а третья — по сумме. Хоть значения нескольких колонок логически хранятся вместе, на физическом уровне данные всё равно могуть быть разделены для повышения эффективности хранения.

Обычные индексы в виде деревьев и хеш-таблиц неэффективны для колоночных хранилищ, вообще плохо приспособленных к извлечению единичных записей. Для ускорения поиска используют метаданные[1], в которых указываются минимальные и максимальные значения колонок в части хранилища — странице, файле и т. п.:



Если, например, по проекции на рисунке выше будет выполняться запрос всех значений до 30, то первый блок вообще не будет прочитан, т. к. минимальное значение в его метаданных больше запрашиваемого.

Метаданные с обобщёнными значениями используются во многих платформах, однако единого термина для их названия не существует — даже один производитель может использовать несколько названий. Вот лишь некоторые из них:

- storage indexes — Oracle Exadata и Oracle In-memory Option;
- zonemaps — Netezza, Oracle[2];
- MinMax indices — VectorWise;
- segment elimination — SingleStore;
- synopsis tables — Db2;
- засечки (marks) — ClickHouse;

[1] Минимальные и максимальные значения могут использоваться и при построчном хранении — например, индексы BRIN в PostgreSQL.

[2] В отличие от «storage indexes», «zonemaps» не требуют ни комплекса Exadata, ни Oracle In-memory Option.

- block range index (BRIN) — PostgreSQL;
- zone map, storage index, data skipping, constraint exclusion — SAP IQ.

В отличие от традиционных индексов, минимальные и максимальные значения можно вычислять на лету — как результат выполнения аналитических запросов. Впервые такая техника была применена в MonetDB и называлась расщеплением базы (database cracking).

Обновление данных

Очевидно, что задача обновления данных при колоночном хранении существенно сложнее, чем при построчном: каждое обновление требует поиска, распаковки, изменения и упаковки нескольких наборов данных — по количеству обновляемых колонок.

В платформах, использующих колоночное хранение, используется два подхода к обновлению данных.

Первый подход заключается в поддержке **буферной области**, куда записываются изменения данных. Данные в этой области хранятся построчно, а сама область зачастую располагается в памяти. Читающие запросы объединяют данные из колоночного хранилища с данными из буфера во время чтения, а фоновые процессы время от времени сохраняют накопившиеся изменения из буферной области в основное колоночное хранилище.

Как и в случае с картами минимальных и максимальных значений, единого термина для обозначения этого буфера не существует:

- разностные группы строк (deltastore) — Microsoft SQL Server;
- RLV store (row-level versioning) — SAP IQ;
- write-optimized storage (в противовес основному колоночному хранилищу read-optimized storage) — C-Store и Vertica.

MonetDB хранит обновлённые строки в самом журнале транзакций, который сохраняется до выполнения первой операции чтения изменённых данных. Читающая операция объединяет данные и записывает их в основное хранилище.

На первый взгляд кажется, что изменения стоит идентифицировать ключом изменённой строки. При таком подходе слияние изменений становится ресурсоёмкой операцией: во-первых, для слияния обязательно читать ключ, даже если он не изменяется; во-вторых, данные в таблице могут быть отсортированы не по ключу, что не позволяет использовать алгоритм слияния сортированных списков (merge join).

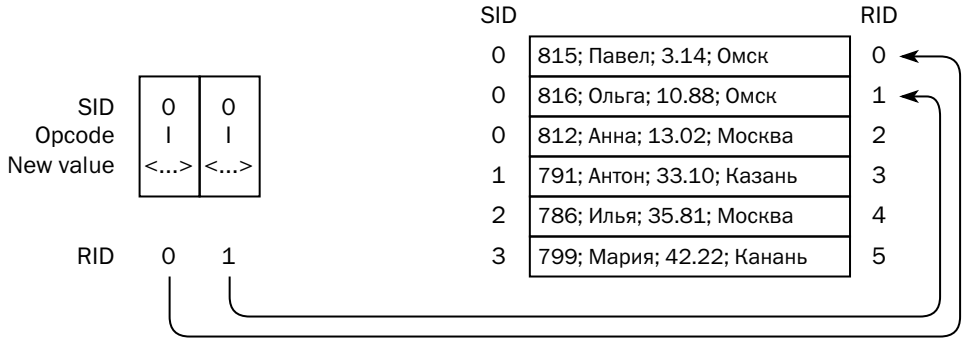
Выходом является идентификация изменений не ключом, а позицией, в которой произошло обновление. Проблема лишь в том, что при вставке или удалении строки́ позиции всех следующих строк изменяются. Для решения этой проблемы в 2010 году учёные из Амстердамского университета Шандор Хеман (Sándor Héman) и Марчин Жуковски (Marcin Zukowski) предложили **дерево позиционных изменений (positional delta tree, PDT)**.

Строки в таблице идентифицируются двумя числами: SID (stable identifier) — позиция строки в «стабильной» таблице, то есть в постоянном хранилище, и RID (row identifier) — позиция строки в текущем состоянии таблицы. RID = SID + Δ, где Δ (дельта) — изменение позиции строки за счёт вставок и удалений в предыдущих позициях.

Нумерация обоих идентификаторов начинается с 0, оба идентификатора виртуальны, то есть в постоянном хранилище их нет. Для строк из постоянного хранилища оба идентификатора равны позиции строки в таблице (в порядке сортировки):

SID		RID
0	812; Анна; 13.02; Москва	0
1	791; Антон; 33.10; Казань	1
2	786; Илья; 35.81; Москва	2
3	799; Мария; 42.22; Казань	3

Вставим в таблицу несколько строк. Для хранения обновлений будем использовать структуру, в которой сохраняются SID, тип операции и новое значение, если оно есть, то есть если это не операция удаления:



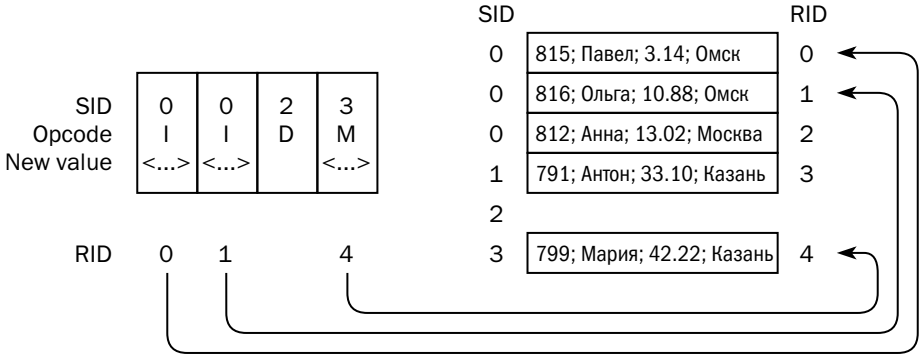
Слева на рисунке изображена область изменений, а справа — общий вид таблицы с учётом и основного хранилища, и изменений. Если строка добавляется,

то в области изменений ей приписывается SID, который бы оказался у этой строки, если бы она уже была в таблице.

Чтобы вычислить позицию строки в обновлённой таблице (RID), нужно к SID прибавить дельту. Изначальна дельта равна нулю, а в процессе движения по области изменений дельта изменяется:

- увеличивается для следующего элемента на 1, если обработана операция вставки;
- уменьшается для следующего элемента на 1, если обработана операция удаления;
- остаётся неизменной, если была обработана операция обновления.

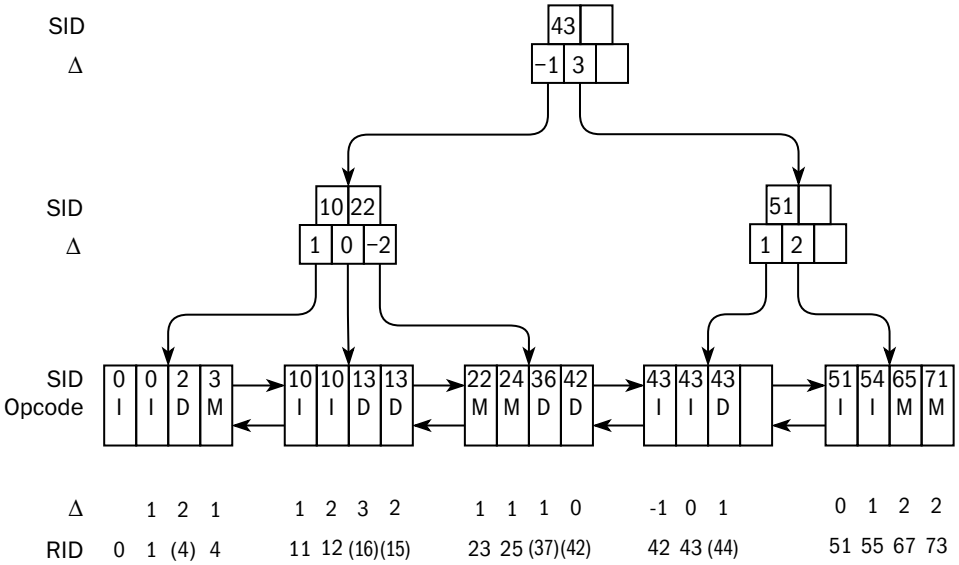
Проведём ещё несколько операций над таблицей:



Структура хранения обновлений в виде простого списка хороша для последовательного доступа, а для возможности произвольного доступа структура должна быть проиндексирована.

Если бы мы продолжили операции, то могли бы получить последовательность изменений, первые четыре из которых подробно разобраны в предыдущем абзаце. Эти изменения составляют самый первый (левый) лист дерева.

В нелистовых узлах расположены значения SID, чередующиеся с указателями на поддеревья, SID в которых лежат в интервале между значениями. Каждому указателю приписано изменение дельты, соответствующее проходу по поддереву, на которое он указывает. Например, если мы начнём обход дерева с начала, то дельта будет равна нулю, а когда пройдём первые три листа, составляющие левое поддерево, дельта будет равна −1. Соответственно, указателю на левое поддерево в корне приписано значение −1.



Пусть теперь условие выборки данных из таблицы сформулировано так, что требуется извлечь данные, начиная с SID = 42. Процесс чтения должен найти те изменения, которые влияют на данные, начиная с этого SID. Для этого он спускается вниз по дереву, попутно вычисляя дельту, которую надо добавлять к SID, чтобы получить актуальный RID.

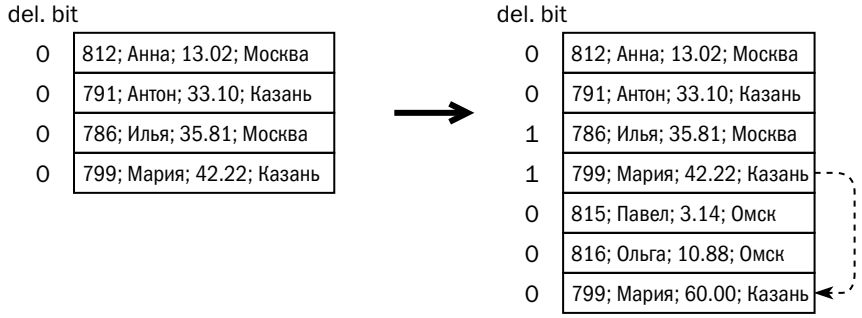
Считаем дельту равной нулю и начинаем путь вниз по дереву с корневого узла:

- На первом уровне дерево делится по SID = 43. Поскольку 42 < 43, идём в левое поддерево. Дельта при этом не меняется.
- На втором уровне дерево делится на три поддерева по SID = 10 и SID = 22. Поскольку 42 > 22, идём в третье поддерево, а к дельте прибавляем изменения, которые вносят первые два поддерева, — 1 и 0, то есть теперь дельта равна единице.
- На третьем уровне попадаем в лист. Зная, что для первого элемента этого листа дельта равна единице, двигаемся вправо до нужного ключа, изменяя дельту по правилам. Поскольку до ключа 42 встречается две операции изменения, не меняющие дельту, и одна операция удаления, уменьшающая дельту на 1, для SID = 42 дельта становится равной 0.

Второй подход к хранению изменений в системах с колоночным хранением данных заключается в поддержке **карт удалённых строк (deletion bitmaps)**. Каждой строке ставится в соответствие флаг актуальности. При изменении строки её старая версия помечается удалённой, а новая версия просто вставляется

в конец таблицы. При реорганизации таблицы удалённые версии строк будут из неё исключены. Такая схема применяется, например, в Db2.

Очевидно, что эта схема существенно проще, но она не позволяет воспользоваться порядком сортировки данных в таблице при обработке.



Несмотря на все ухищрения, применяемые разработчиками колоночных СУБД для обеспечения возможности изменения данных, рекомендация по эффективному использованию таких платформ остаётся крайне консервативной:

- загружать данные в систему большими порциями;
- загружать данные как можно реже;
- никогда не изменять данные.

3.5. Локализация данных

Секционирование

Секционированием (partitioning, или реже fragmentation) называется физическое разделение данных в базе на части (секции) с разными параметрами физического хранения в зависимости от значений каких-либо полей.

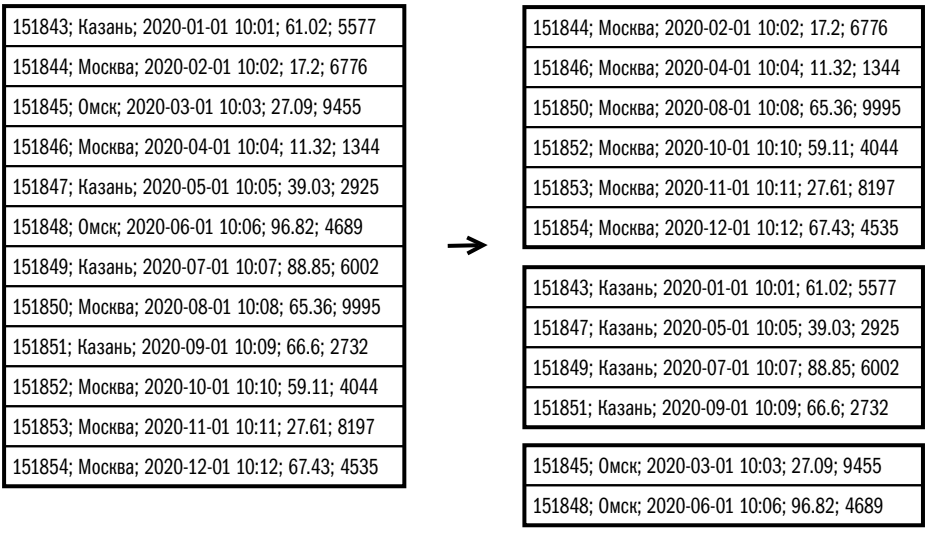
Пусть, например, приложение, использующее базу данных, обрабатывает транзакции торговой компании, имеющей филиалы в Москве, Казани и Омске. Записи о транзакциях хранятся в базе данных вперемежку. Однако если в транзакции есть колонка-идентификатор филиала, в котором совершена транзакция, то современные реляционные СУБД позволяют физически отделить строки друг от друга.

Каждая секция хранится на диске независимо и представляет собой отдельный объект. В разных СУБД этот объект называется по-разному: в Oracle это сегмент

(segment), в Db2 — объект хранения (storage object), в PostgreSQL секция — обычная таблица, а в Microsoft SQL Server у объекта хранения собственное название отсутствует.

При обсуждении LSM-дерева мы говорили о делении данных на файлы по диапазонам ключей, которое некоторые источники также могут называть секционированием. Здесь же мы обсуждаем деление данных на фрагменты в зависимости от значений любой колонки, которая может быть первичным ключом или частью первичного ключа, а может и не быть — как в приведённом примере. Колонка, на основании значения которой принимается решение о помещении строки в ту или иную секцию, называется **ключом секционирования** (partitioning key, или иногда partition key).

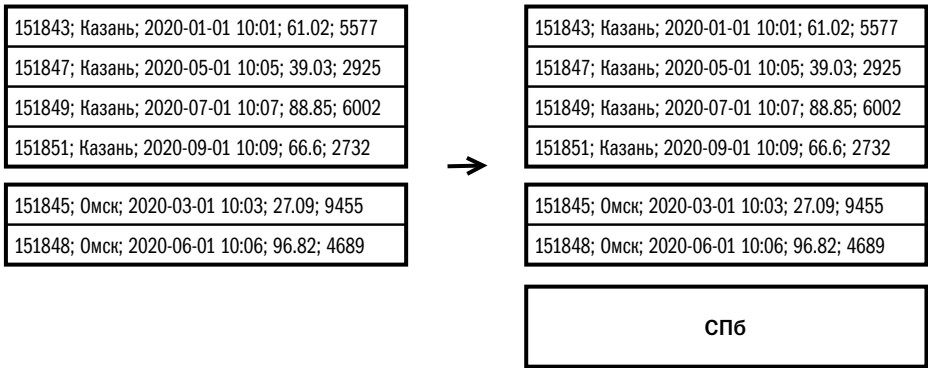
На рисунке ниже секция выделена для каждого города:



Запись в секционированную таблицу выполняется медленнее, чем в несекционированную, т. к. при записи СУБД должна проанализировать данные строки́ и принять решение, в какую из секций её вставлять. Почти во всех платформах есть возможность вставить стро́ки в конкретную секцию, однако делать так рекомендуется только при импорте данных из внешних источников под контролем оператора: если при записи, например, в московскую секцию в колонке «Филиал» окажется значение «Омск», операция остановится с ошибкой.

Со временем количество секций в таблице может меняться. Администратор может как создавать новые секции для новых значений ключа секционирования,

так и удалять существующие. При удалении секции все данные, которые в ней хранились, исчезают. При создании новой секции изменяется описание таблицы и создаётся новый объект хранения:

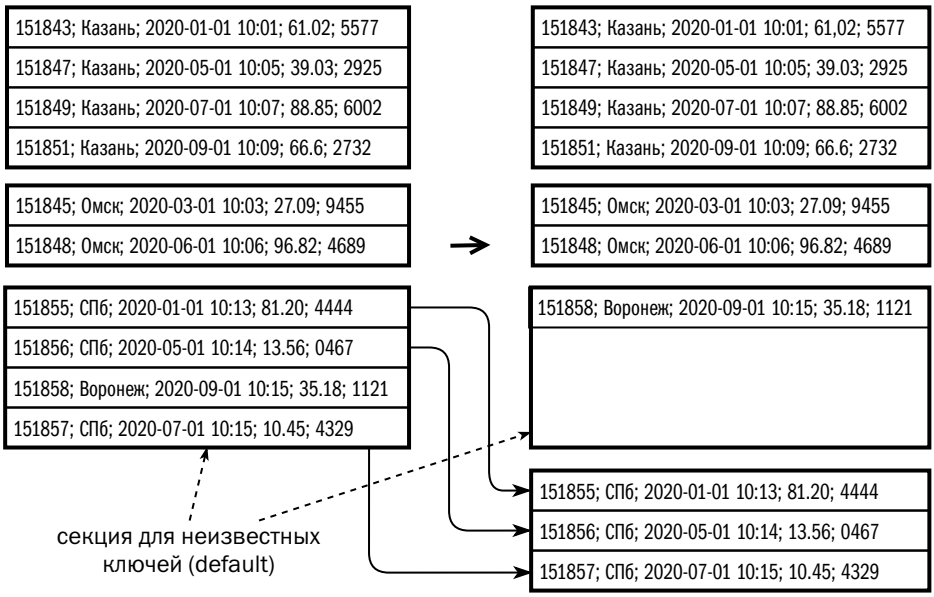


Одной и той же секции может соответствовать несколько значений ключа. Например, в обсуждаемом примере мы могли бы создать общую секцию для Омска и Казани.

Если при вставке значение ключа секционирования не соответствует ни одному из допустимых значений (например, в обсуждаемую таблицу вставляется строка со значением «Воронеж» в колонке «Филиал»), операция остановится с ошибкой[1]. Администратор может создать секцию для неизвестных ключей (default, «по умолчанию»), в которую будут попадать строки со значениями, для которых нет выделенных секций. Правда, создание новой выделенной секции при наличии секции «по умолчанию» будет уже не таким быстрым, т. к. помимо создания нового объекта хранения потребуется просмотр этой секции и, возможно, физическое перемещение части строк в новую секцию.

Если приложение изменяет значение ключа секционирования, то строка может переместиться из одной секции в другую. В большинстве СУБД (Db2, PostgreSQL, Microsoft SQL Server, MySQL) такое поведение реализовано по умолчанию, но Oracle при попытке изменить ключ секционирования выдаёт ошибку, а разрешать перемещение строк нужно явно, создав таблицу с опцией `ENABLE ROW MOVEMENT`.

[1] Oracle версии 12c и более поздних поддерживает автоматическое создание секции при вставке строки с ранее не встречавшимся значением колонки.



Секционирование в общем случае не гарантирует, что размеры секций будут одинаковы. Так, в одном из рассмотренных примеров размер секции со значением «Москва» составляет половину размера всей таблицы. При выборе ключа секционирования следует это учитывать.

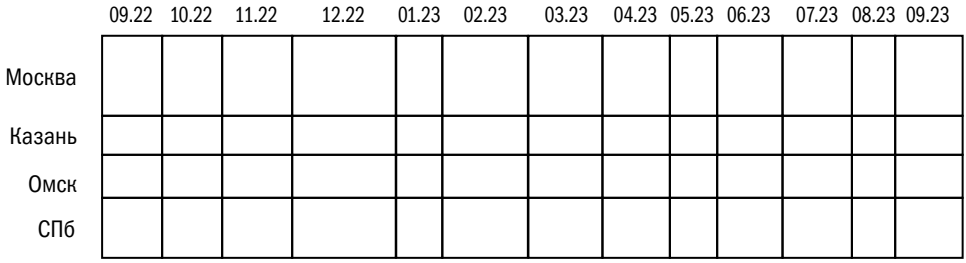
В примере выше мы рассмотрели таблицу, где значения колонки для каждой секции заданы явно. Такой подход называется **секционирование по списку значений (partition by list)**, и он применяется для деления таблиц на части по колонкам, имеющим ограниченное количество значений, — филиалам, странам и т. п. Но есть и другие способы секционирования.

Секционирование по диапазонам (partition by range) — подход, при котором для каждой секции задаются границы значений ключа. Широко применяется для деления таблицы на временны́е интервалы — например, каждая секция может соответствовать месяцу или дню.

Секционирование по хеш-сумме (partition by hash). Создавая таблицу, администратор указывает ключ секционирования и количество секций. Как правило, в качестве ключа секционирования по хеш-сумме выбирается первичный ключ. При записи данных в таблицу для каждой строки рассчитывается значение хеш-функции от ключа секционирования, затем вычисляется остаток от деления на количество секций, и строка помещается в соответствующую секцию. Если значение ключа секционирования достаточно разнообразно (например, это

уникальное число, генерируемое в качестве суррогатного ключа), то распределение данных между секциями будет равномерным. Секционирование по хеш-сумме применяется, чтобы ускорить параллельную обработку данных в таблице — например, каждую секцию можно поместить на выделенный диск.

Композитное секционирование (composite partitioning). Каждая секция делится на подсекции (subpartitions), и при этом каждая подсекция является отдельным объектом хранения. Применяется для больших баз данных, где, например, данные за определённый временной интервал имеет смысл поделить также и по филиалам — в этом случае секции определяются по интервалам, а подсекции — по списку. Возможность выбора параметров разбиения на секции и на подсекции зависит от конкретной платформы и от её версии. На рисунке ниже показана таблица, состоящая из 52 объектов хранения — тринадцать секций по четыре подсекции в каждой:

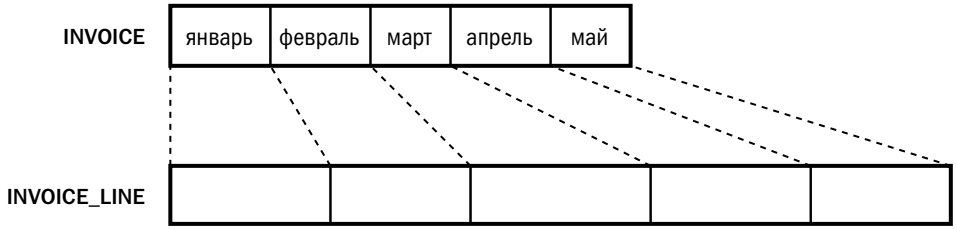


На рисунке виден типичный подход к созданию секций: регламент эксплуатации приложения требует, чтобы система хранила данные за определённый период. Как только данные уходят за горизонт хранения, секции с этими данными удаляются, и создаются новые секции для новых данных. Ситуация на рисунке соответствует сентябрю 2023 года и глубине хранения, равной году.

Ссылочное секционирование (partition by reference) — подход, при котором ключ секционирования находится в другой, родительской таблице, с которой данная таблица связана внешним ключом.

Пусть, например, у нас есть таблица накладных и таблица, содержащая товарные позиции, входящие в накладную:

Таблица накладных (invoice) секционирована по дате формирования накладной (inv_date). Логично было бы секционировать по этой же дате и таблицу товарных позиций (invoice_line), но для этого необходимо добавить в неё соответствующую колонку и, самое важное, — корректно заполнять эту колонку. Oracle предлагает ещё один способ: таблица объявляется секционированной по ссылке, и после этого СУБД сама поддерживает взаимно однозначное соответствие между секциями родительской и дочерней таблиц:



Многие платформы поддерживают лишь некоторые методы секционирования. Ниже представлена сводная таблица:

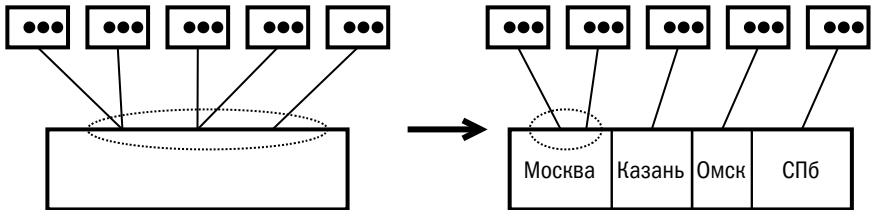
	Oracle	Db2	MS SQL	PostgreSQL	MySQL
По списку	+	–	–	+	+
По диапазонам	+	+	+	+	+
По хеш-сумме	+	–	+[1]	+	+
По ссылке	+	–	–	–	–
Композитное	+	–	–	+	+[2]

Какие же преимущества даёт секционирование?

В транзакционных базах данных секционирование может снизить конкуренцию за данные. Если, например, каждый сервер бизнес-логики обслуживает один филиал, то серверы разных филиалов не конкурируют друг с другом за страницы, хотя конкуренция серверов, обслуживающих один филиал, конечно же, сохраняется:

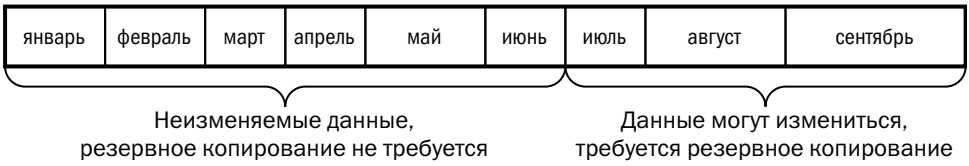
[1] Реализуется как секционирование по диапазонам вычисляемого поля, в которое записывается хеш-сумма ключа.

[2] Секция может быть разбита на подсекции только по хеш-сумме (range/hash или list/hash).



В аналитических системах секционирование позволяет существенно уменьшить объём читаемых данных. При построении отчёта по одному из филиалов (и при любых других запросах, где в параметрах указан филиал) сервер не будет обращаться к областям, где хранятся данные других филиалов.

Администратору секционирование по времени совершения транзакции позволяет упростить резервное копирование. Глубина изменения данных ограничена логикой приложения, поэтому данные, возраст которых превышает порог изменения, уже не подлежат резервному копированию, поскольку со времени последнего копирования они не изменялись:



Кроме того, секционирование позволяет организовать архивирование данных. Так, например, если принято решение ограничить глубину хранения шестью месяцами, то секции с января по март можно удалить. Операция удаления будет мгновенной, т. к. для удаления секции требуется лишь изменение словаря, где отмечается, что некоторые объекты хранения больше не принадлежат базе данных. Секции с апреля по июнь можно сжать и поместить на более дешёвые и медленные диски. При этом производительность чтения и изменения актуальных данных, находящихся в секциях с июля по сентябрь, не пострадает.

Изначально секционирование было привилегией больших баз данных, использовавшихся в крупных организациях, — впервые оно появилось как дополнение (опция) к СУБД Oracle версии 8.0, выпущенной в 1997 году. Однако с тех пор объёмы данных многократно выросли, и с середины 2000-х годов секционирование поддерживается всеми реляционными БД и может быть использовано для любых данных.

Шардирование

Шардирование (sharding), как и секционирование, — физическое разделение данных на фрагменты, которые могут храниться независимо и обрабатываться параллельно. Секционирование преследует цель упростить администрирование данных и ускорить чтение, но при этом размер секций зависит исключительно от профиля данных, хранимых в таблице. Главная задача шардирования — поделить данные между несколькими независимыми обработчиками, желательно приблизительно поровну[1].

Важно, что связанные данные (например, клиент и все его счета) должны храниться вместе, то есть в одном шарде, а это значит, что ключ шардирования у них должен быть один и тот же. Но если шардирование по первичному ключу (например, шардирование клиентов по идентификатору клиента) разобьёт данные на примерно равные части, то шардирование по другому полю (например, шардирование счетов по идентификатору клиента, которому выставлен счёт) может создать значительный перекос в размерах шардов.

Прообразом шардирования можно считать секционирование по хеш-сумме. Как правило, термин «шардирование» употребляется, если данные делятся между несколькими узлами, каждый из которых имеет прямой доступ только к своей части данных. При этом узлы даже не обязаны работать на разных серверах: для однопоточных платформ, таких как Tarantool/Picodata, Redis или VoltDB, запуск нескольких узлов на одном сервере — одна из рекомендуемых конфигураций.

Количество узлов в кластере может изменяться динамически в достаточно широких пределах: как правило, система запускается под небольшой нагрузкой, но планируется, что со временем нагрузка может вырасти в несколько раз или даже на несколько порядков. Это условие формирует целый ряд требований к механизму распределения данных:

- данные должны делиться примерно поровну между узлами при любом количестве узлов (разумеется, в некоторых пределах);

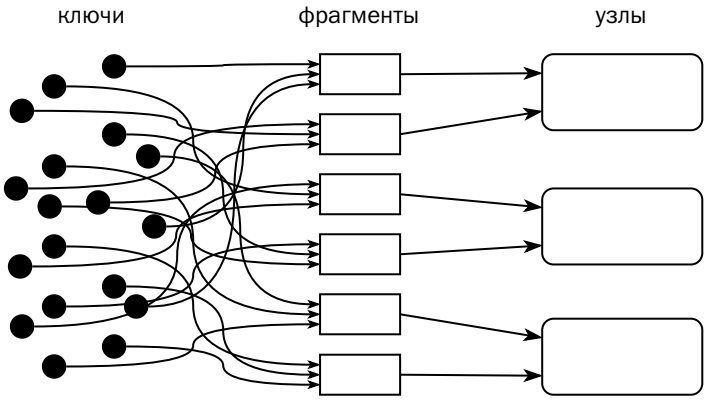
[1] Объём обрабатываемых данных должен быть пропорционален мощности узлов. В системах, поддерживающих узлы разной мощности, перераспределение данных происходит за счёт изменения количества фрагментов на узле. Если система не поддерживает узлы разной мощности, то производительность узлов кластера выравнивается по производительности наименее мощного узла.

- выход узла из строя не должен приводить к недоступности данных, обрабатываемых этим узлом;
- данные должны быстро и с минимальной нагрузкой перераспределяться между узлами, которые в любой момент могут быть добавлены в кластер или выведены из кластера.

Одной из первых распределённых систем без общих компонентов стала аналитическая платформа Teradata, вышедшая на рынок в 1984 году. Принципы деления данных между узлами, заложенные в Teradata, используются сегодня в большинстве распределённых систем, включая и те, что предназначены для обработки транзакций.

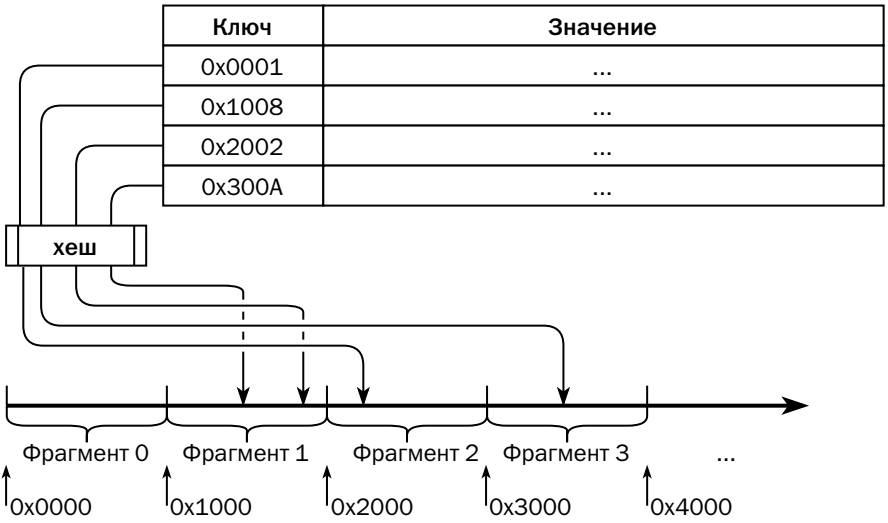
Вот эти принципы:

1. В подавляющем большинстве случаев ключ шардирования является первичным ключом или частью первичного ключа, хотя возможно и шардирование на основании произвольных полей (Teradata, MongoDB).
2. Данные делятся на фрагменты. Количество фрагментов больше, чем предполагаемое максимальное количество узлов в кластере. Каждый узел хранит и обрабатывает несколько фрагментов, и количество фрагментов на узле зависит от количества узлов в кластере. Разные платформы дают разные рекомендации по оптимальному количеству фрагментов на узле: 2–8 (Greenplum), 20–200 (HBase), 30–100 (Apache Ignite), 1000 (Tarantool/Picodata).
3. Поиск объекта по ключу осуществляется в два этапа: сначала вычисляется номер фрагмента, а затем находится узел, который хранит соответствующий фрагмент.



Есть два основных подхода к делению данных на фрагменты.

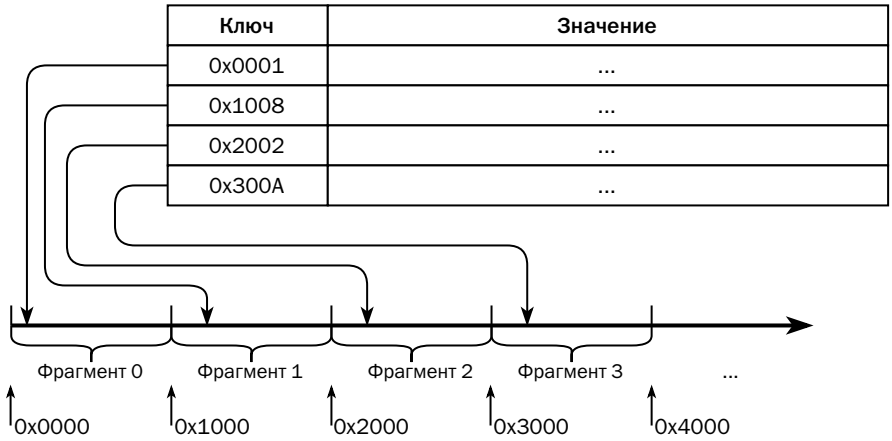
Подход, основанный на хешировании, зародился вместе с первыми распределёнными базами, которые предоставляли программисту только интерфейс «ключ—значение». При этом подходе количество фрагментов определяется перед началом записи данных в фрагментированное хранилище (таблицу) и впоследствии не может быть изменено. Объект относится к тому или иному фрагменту на основании значения хеш-суммы ключа шардирования. Номер фрагмента вычисляется как остаток от деления хеш-суммы на количество фрагментов:



Некоторые алгоритмы распределения ставят в соответствие каждому фрагменту диапазон значений хеш-суммы. Сначала вычисляется размер фрагмента как частное от деления максимального значения хеш-функции на количество фрагментов, а затем для каждого ключа вычисляется номер фрагмента как частное от деления хеш-суммы на размер фрагмента.

Распределение, основанное на хешировании, используется в таких платформах, как Teradata, Apache Cassandra, Apache Ignite/GridGain, Tarantool, Amazon DynamoDB, YugabyteDB.

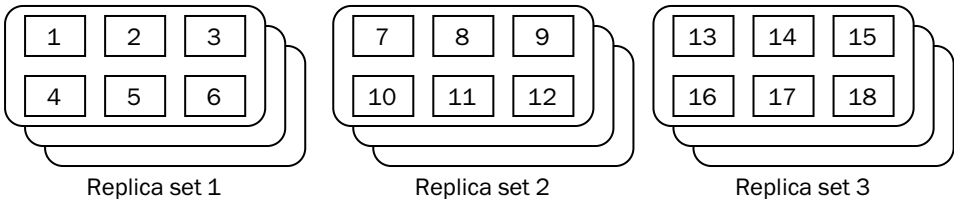
Второй подход основан на делении ключей по диапазонам. Пустая таблица состоит из единственного фрагмента, куда попадают все записи. Когда фрагмент достигает максимального размера, он делится на два, и часть ключей мигрирует в новый фрагмент:



Распределение по диапазонам используется в MongoDB, HBase, Google Spanner, CockroachDB, YugabyteDB[1].

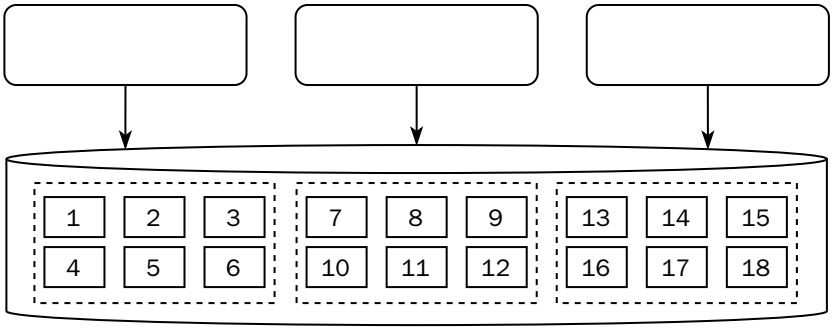
В большинстве случаев подход с хеш-суммой оказывается эффективнее: он равномерно распределяет данные между фрагментами, не требует перераспределения, при массивных операциях не создаёт «горячих» фрагментов, на которые приходится вся нагрузка. Преимущество распределения по значениям проявляется, когда приложение регулярно выбирает данные по диапазонам ключей. В случае распределения по хеш-сумме для выполнения таких запросов необходим полный просмотр всех фрагментов, а при распределении по значениям подавляющее большинство таких запросов будет выполнено одним узлом поиском по единственному фрагменту.

В ряде платформ (Redis, Tarantool/Picodata, Teradata, Oracle Sharding, MongoDB) каждый узел сам обеспечивает свою отказоустойчивость: у узла есть дублирующие узлы (реплики), которые хранят тот же набор фрагментов, что и основной узел. Узел вместе с дублирующими узлами называется набором (replica set):

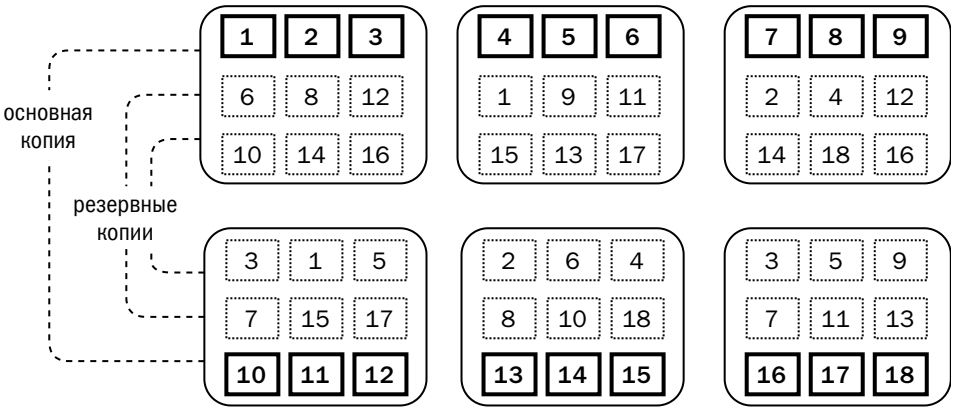


[1] YugabyteDB поддерживает оба подхода к делению на фрагменты. По умолчанию выбирается деление на основе хеширования.

В других платформах (HBase, SAP HANA) узлы подключены к общей системе хранения данных, которая должна быть надёжна, и при аварии оставшиеся узлы могут прочитать с дисков данные потерянных узлов:



Наиболее же популярный подход (Apache Cassandra, Apache Ignite, Hazelcast, Greenplum и др.) заключается в том, что узел считается ненадёжным, и каждый фрагмент должен храниться на нескольких узлах одновременно, а обрабатываться только на одном. Копия фрагмента на том узле, который его обрабатывает, называется основной (master, owner, primary, leader), а остальные копии этого фрагмента на других узлах — резервными (backup, mirror, fallback, replica). В случае потери узла для каждого фрагмента, основная копия которого находилась на этом узле, выбирается новая основная копия, а на одном из оставшихся узлов создаётся новая резервная копия. Новые основные копии фрагментов, находившихся на потерянном узле, могут оказаться на разных узлах. Такой подход предъявляет дополнительные требования к функции распределения — она должна для каждого фрагмента назначить не один узел, а несколько, по количеству копий:



Несмотря на то что распределённые системы существуют давно, единой терминологии для обозначения фрагментов и узлов не выработано, каждый производитель использует собственную. В таблице вы найдёте примеры терминов:

Платформа	Фрагмент	Узел
Apache Ignite/GridGain, Riak KV, Redis, VoltDB	partition	node
Apache Cassandra	virtual node	host
MongoDB, Oracle Sharding Option	chunk	shard
Tarantool/Picodata	bucket	shard
Teradata	access module processor (AMP)	node
Hazelcast	partition	member
SAP HANA	shard	host
HBase	region	regionserver
Greenplum	segment	host/node
CockroachDB	range	node
YugabyteDB	chunk/shard/tablet	server

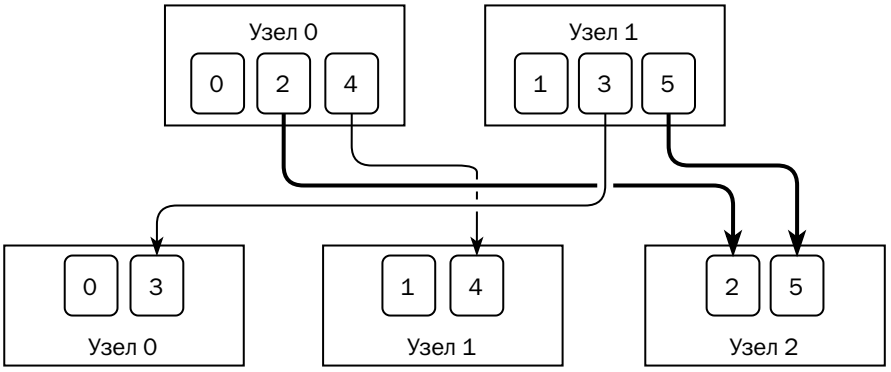
Если деление ключей на фрагменты устроено примерно одинаково (отличаться может только алгоритм хеширования), то для распределения фрагментов по узлам существует несколько алгоритмов. Некоторые платформы (например, Apache Ignite) позволяют переопределить функцию распределения.

Предопределённая карта распределения. Этот подход применяется в платформе Teradata. При инсталляции системы фиксируется количество фрагментов (AMPs) и генерируются карты, согласно которым эти фрагменты распределяются между узлами. На самом деле каждый узел Teradata — это небольшой кластер (клика, clique[1]), состоящий из нескольких серверов с общей системой хранения данных, и в случае сбоя сервера фрагменты могут переезжать с сервера на сервер внутри этого кластера.

В большинстве современных платформ предполагается, что количество узлов в кластере может динамически меняться, поэтому требуется алгоритм, позволяющий размещать данные на любом количестве узлов. Интуиция подсказывает,

[1] В английском языке, в отличие от русского, слово «клика» не несёт негативных коннотаций.

что при выборе узла для фрагмента достаточно взять остаток от деления номера фрагмента на количество узлов. Проблема такого подхода в том, что при изменении количества узлов многие фрагменты должны будут переехать с узла на узел:

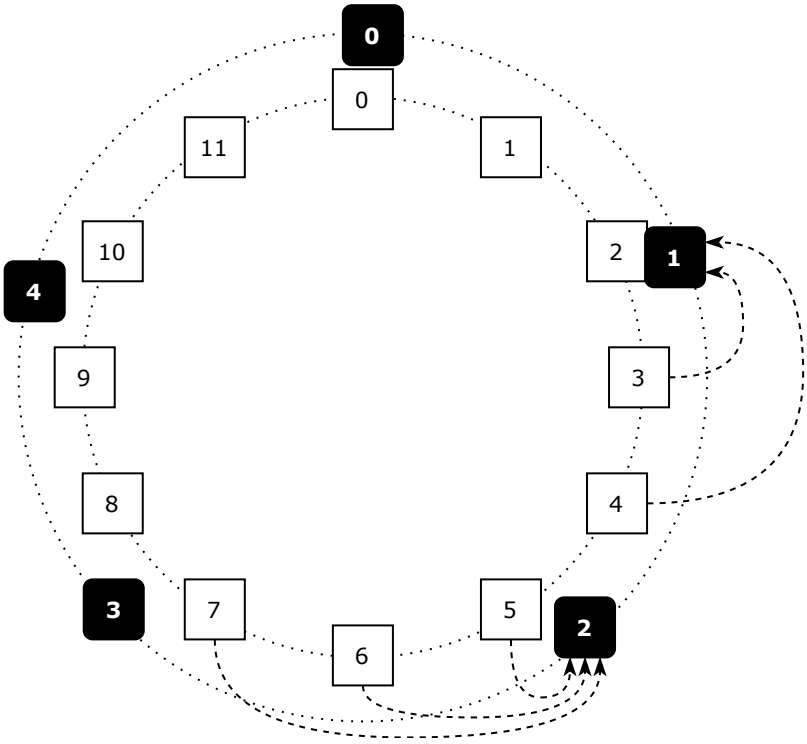


На рисунке выше при добавлении в кластер третьего узла придётся переместить четыре фрагмента, причём два фрагмента переезжают с узла 0 на узел 1 и обратно. Очевидно, что было бы достаточно перенести всего два фрагмента — по одному фрагменту с каждого старого узла на новый узел. Для минимизации переноса данных при изменении количества узлов придумано несколько алгоритмов.

Справедливое распределение (fair affinity) применяется в таких платформах, как Greenplum, Oracle Sharding, Hazelcast, Tarantool, MongoDB, а также в ранних версиях GridGain. При развёртывании кластера данные делятся поровну между узлами. При необходимости данные перераспределяются между узлами так, чтобы нагрузка на узлы была одинаковой — отсюда название «справедливое». Очевидное достоинство подхода в том, что критерии «справедливости» могут быть самые разные: при перераспределении данных можно учитывать не только количество фрагментов, но и размер фрагментов и количество транзакций, обрабатываемых каждым узлом. Недостаток заключается в том, что решение о перераспределении принимает один узел: если несколько узлов будут параллельно строить карты распределения данных, они могут получить разные результаты. После создания новой карты узел-создатель рассылает эту карту остальным узлам (GridGain, Hazelcast) или выступает в роли маршрутизатора запросов (Tarantool, Greenplum, MongoDB). Oracle Sharding поддерживает специальные узлы-маршрутизаторы (shard directors), отдельные от узлов хранения и обработки данных (shards).

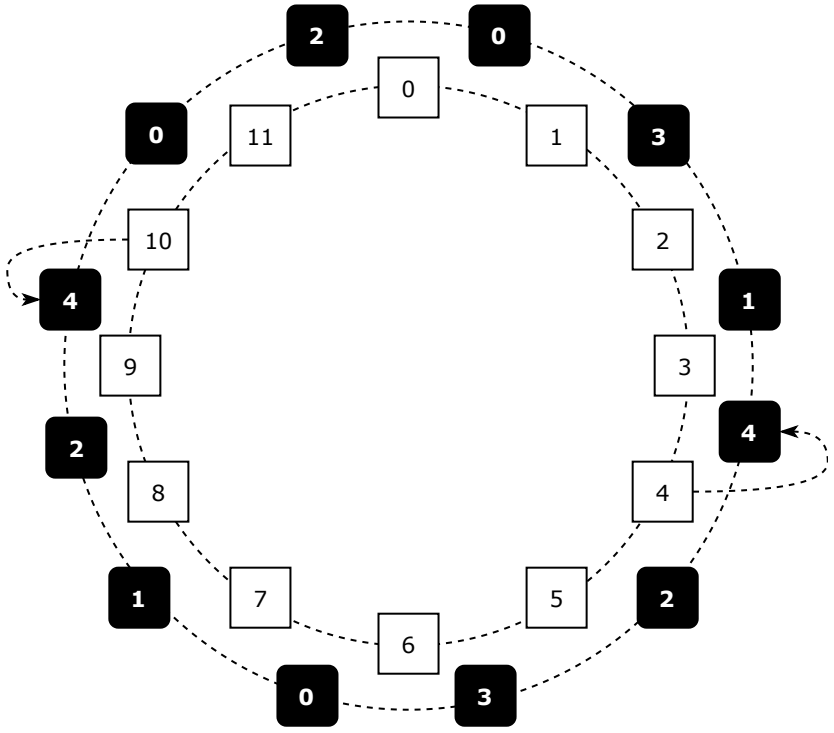
Согласованное хеширование (consistent hashing) — алгоритм, предложенный группой учёных Массачусетского Технологического института под руководством Дэвида Каргера (David R. Karger) в 1997 году. Согласованное хеширование используется в таких системах, как Riak KV, Redis, Cassandra, и других.

Фрагменты располагаются на виртуальной окружности через равные промежутки — каждому фрагменту соответствует угол $k \times 360° / n$, где k — номер фрагмента, а n — общее количество фрагментов. На той же окружности по тому же принципу располагаются и метки узлов. Каждый фрагмент размещается на узле, метка которого находится ближе всего к нему, если идти вдоль окружности против часовой стрелки:



На рисунке видно, что при удалении из кластера узла 2 к узлу 1 будет отнесено пять фрагментов, что в два с половиной раза больше, чем на узлах 0 и 3, а при добавлении узла новый узел заберёт часть данных с одного из узлов, никак не разгрузив остальные узлы. Чтобы распределять данные между узлами более равномерно, каждому узлу приписывается не одна метка, а несколько. Если кластер состоит из узлов разной мощности, то для каждого узла может быть выбрано

количество меток, соответствующее его мощности, — такой подход используется, например, в распределённой файловой системе Ceph:

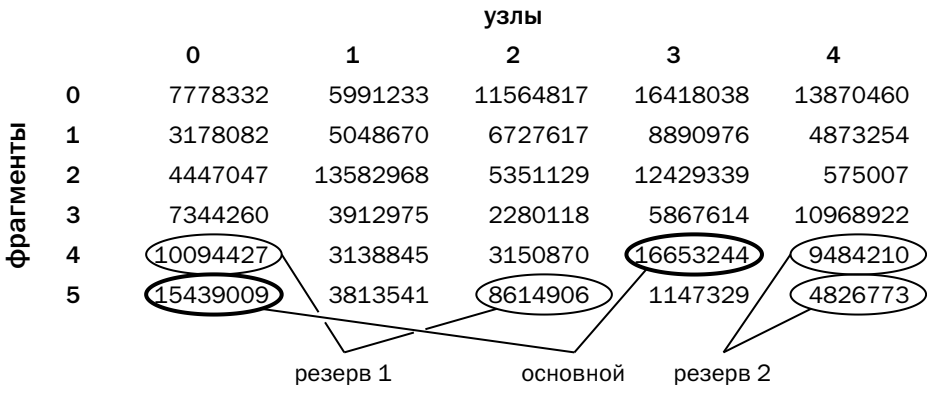


На рисунке видно, что при потере узла 4 его данные будут равномерно распределены между узлами 1 и 2, а остальные данные не потребуют перемещения. Метки узлов располагаются на окружности в псевдослучайном порядке — в противном случае независимо от количества меток у каждого узла все данные с потерянного узла переезжали бы на предыдущий, а добавление узла разгружало бы единственный узел.

В отличие от справедливого распределения, алгоритм согласованного хеширования позволяет перестроить карту меток независимо на каждом узле кластера. Для этого необходимо, чтобы генератор меток узлов кластера работал одинаково на всех узлах.

Если фрагменту нужно назначить не один узел, а несколько (основной и, например, два резервных), то в качестве резервных следует взять узлы, метки которых стоят на окружности следующими. Так, на рисунке выше фрагменту 4 основным назначен узел 4, а резервными узлами будут назначены 1 и 3.

Алгоритм **рандеву (rendezvous hashing)** разработали в 1996 году сотрудники Университета Мичигана Дэвид Талер (David Thaler) и Чиня Равишанкар (Chinya Ravishankar). Для работы этому алгоритму требуется хеш-функция $H(n, c)$, принимающая два значения — идентификатор узла и номер фрагмента. Для каждой пары, состоящей из номера фрагмента и номера узла, рассчитывается значение хеш-функции. Основным для фрагмента назначается узел, для которого значение функции максимально (наиболее удалённый узел, если интерпретировать вычисленное значение как расстояние). Если же необходимо дублирование, то резервными назначаются следующие узлы в порядке «приближения» (убывания значения хеш-функции) к фрагменту:



Как и согласованное хеширование, алгоритм рандеву позволяет перестроить карту фрагментов независимо на всех узлах кластера. В обоих случаях карта хранится на узлах на протяжении всего времени, пока состав кластера остаётся неизменным, и используется для маршрутизации запросов по ключам.

Несмотря на то что алгоритм рандеву появился на год раньше, чем алгоритм согласованного хеширования, и выглядит более простым для понимания, автору известна его реализация только в платформе Apache Ignite/GridGain.

В 2014 году был изобретён алгоритм «рандеву с весами» (weighted rendezvous), позволяющий работать с узлами разной мощности. «Расстояние» от фрагмента до узла вычисляется по формуле

$$L = \frac{w_i}{-\log \frac{H(n_i, c_j)}{H_{max}}},$$

где w_i — «вес» i-го узла, $H(n_i, c_j)$ — значение хеш-функции для i-го узла и j-го фрагмента, а H_{max} — максимальное значение хеш-функции.

3.6. СУБД в памяти

СУБД в памяти занимают относительно небольшую долю рынка: из крупнейших поставщиков коммерческих платформ лишь SAP получает основную долю выручки от продажи СУБД в памяти. В первом десятке рейтинга популярности портала db-engines.com СУБД в памяти всего одна, а среди первых 50 позиций — от трёх до пяти.

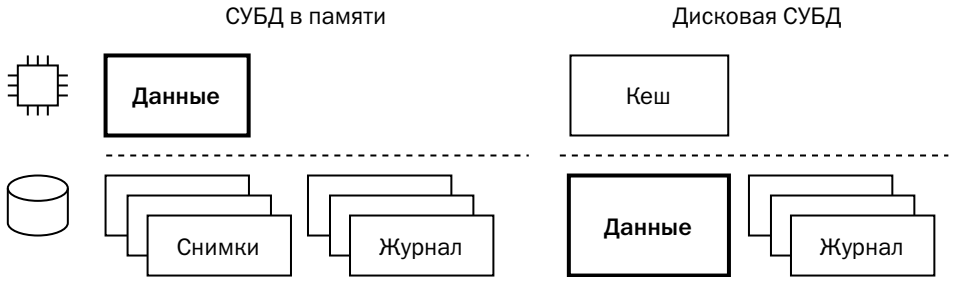
Тем не менее СУБД в памяти прочно заняли свою нишу, а именно нишу платформ для приложений, работающих под нагрузкой до нескольких сотен тысяч запросов в секунду: авторизация банковских карт, маршрутизация запросов между шардами, хранение клиентского профиля, хранение клиентских сеансов, кеширование данных перед традиционными дисковыми СУБД.

IMDG и IMDB

Говоря о платформах для обработки данных в памяти, обычно имеют в виду два родственных класса программного обеспечения: СУБД (in-memory database, IMDB) и кластеры хранения данных (in-memory data grid, IMDG).

Исторически первыми появились СУБД в памяти — solidDB и TimesTen.

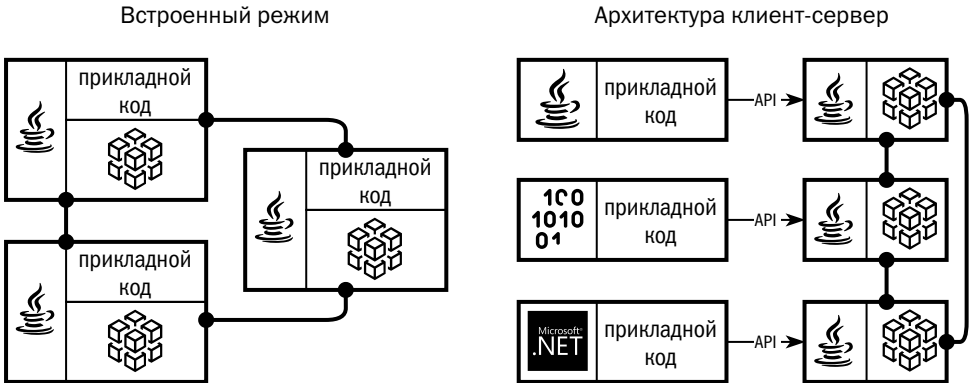
К СУБД в памяти, как и к традиционной дисковой СУБД, предъявляется требование надёжного хранения данных. Первичный образ данных располагается в памяти, а диск используется для восстановления после сбоев. На диск записываются журнал и периодические снимки (snapshot) данных:



Кластеры обработки данных решают задачу распределённого кеширования. Узел может быстро получить из памяти не только те данные, которые запрашивал сам, но и те, которые были запрошены другими узлами кластера. Кроме того, многие платформы включают в себя вычислительный кластер (compute grid), позволяющий выполнять вычисления над данными в памяти, отправляя

код функции на узлы, хранящие данные, вместо того чтобы переносить данные на узел, исполняющий код.

Все существующие IMDG написаны на языке Java и могут быть запущены как во встроенном режиме (embedded/library mode) в коде, выполняемом виртуальной машиной Java (JVM), так и на выделенном оборудовании в конфигурации клиент-сервер (dedicated/client-server mode):



Язык Java не имеет средств для явного освобождения занятой памяти; вместо этого виртуальная машина Java периодически возвращает неиспользуемые участки памяти в пул свободного пространства. Процедура сборки мусора (garbage collection) может вызывать паузы в работе Java-машины, что крайне негативно сказывается на производительности платформы. В связи с этим все современные кластеры минимизируют использование кучи, отдавая основную память для хранения данных под управление собственному менеджеру памяти, управляющему пространством вне кучи (off-heap memory). Пространство выделяется при помощи одного из механизмов JVM — класса `java.nio.ByteBuffer`, класса `Sun.misc.unsafe` или API в пакете `jdk.incubator.foreign`, появившегося в Java 14. Часть платформ (GridGain, Hazelcast, Infinispan) изначально разрабатывались с расчётом на использование памяти вне кучи, другие же получили такую возможность позже — в процессе эволюции.

Наиболее известные платформы класса in-memory data grid приведены в таблице ниже:

Название	Первый выпуск	Компания	Версия с открытым исходным кодом
XAP	2000	GigaSpaces	XAP Open Source
Coherence	2001	Tangosol/Oracle	Coherence Community Edition

Название	Первый выпуск	Компания	Версия с открытым исходным кодом
BigMemory	2003	Greg Luck/Terracotta/Software AG	Ehcache
GridGain	2007	GridGain Systems	Apache Ignite
WebSphere eXtreme Scale	2009	IBM	—
JBoss Data Grid	2009	Red Hat/IBM	Infinispan
Hazelcast	2009	Hazelcast	Hazelcast
GemFire	2013	EMC^2/Pivotal/VMware/Dell	Apache Geode

Сегодня разница между двумя классами платформ практически стёрлась: СУБД в памяти получили развитые средства обработки данных, а многие кластеры обработки получили функциональность сохранения данных на диск.

Обеспечение надёжного хранения

Первое, что приходит в голову разработчику СУБД в памяти, — использовать механизм отображения файла в память (memory-mapped files). Ряд СУБД работают с этим механизмом — например, MonetDB, LMDB и ранние версии MongoDB до перехода на движок WiredTiger. Однако у механизма отображения файлов есть несколько существенных недостатков:

- представление данных на диске совпадает с представлением в памяти;
- нет возможности неблокирующего доступа к памяти;
- СУБД не может знать, какие данные на самом деле находятся в памяти, а какие — нет.

В современных СУБД в памяти применяются собственные менеджеры памяти. Количество возможных структур в памяти велико, поэтому «золотого стандарта индустрии» не существует (возможно, пока), однако ряд приёмов, характерных для СУБД в памяти, уже сформировался.

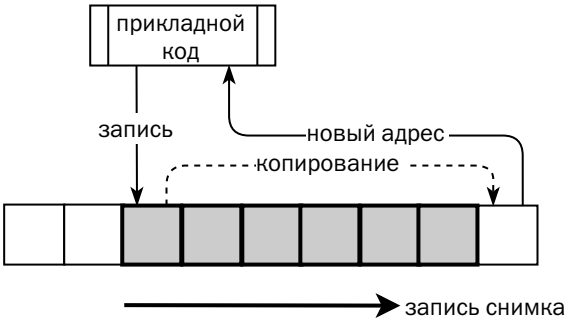
Буферизация записи журнала. Чтобы не потерять данные в случае сбоя сервера, СУБД в памяти, как и дисковая СУБД, обязана надёжно сохранять журнал транзакций. Сброс журнала на диск с использованием системного вызова `fsync()` (или `FlushFileBuffers()` в ОС Windows) — операция продолжительная, поэтому СУБД сбрасывает файловые буферы не после каждой транзакции, а периодически.

Есть вероятность, что сброс буфера не удастся и платформе придётся откатывать все транзакции, которые ждут записи в журнал. С учётом производительности СУБД в памяти, за время между сбросами в буфере может скопиться до нескольких тысяч транзакций, однако удачный сценарий гораздо вероятнее, и платформы проектируют так, чтобы оптимизировать именно наиболее частые сценарии.

Сохранение снимков на диск. Время восстановления данных при сбое прямо пропорционально объёму журналов, обработанных во время восстановления. Периодически СУБД в памяти сохраняет на диск снимок данных[1], и при следующем восстановлении журналы будут накатываться на последний снимок. Сигналом к сохранению снимка может быть тайм-аут или превышение количеством изменённых объектов некоторого порогового значения.

Простейший подход состоит в порождении нового процесса вызовом `fork()` и сохранении данных на диск в порождённом процессе. В операционных системах семейства Unix системный вызов `fork()` создаёт копию текущего процесса, копируя и его память. Чаще всего физического копирования памяти не происходит — вместо этого дочерний и родительский процессы пользуются одними и теми же областями памяти, а реальное копирование происходит при попытке изменить данные (copy-on-write).

Второй подход — блокировка страниц памяти, которые должны быть сохранены на диске. Если требуется изменить данные в заблокированной странице (показаны тёмными на рисунке ниже), то СУБД копирует эту страницу и использует для дальнейшей работы копию, а заблокированная страница помечается как ненужная и после записи на диск освобождается.



[1] Можно провести аналогию между сохранением снимка памяти и процедурой контрольной точки в дисковых СУБД.

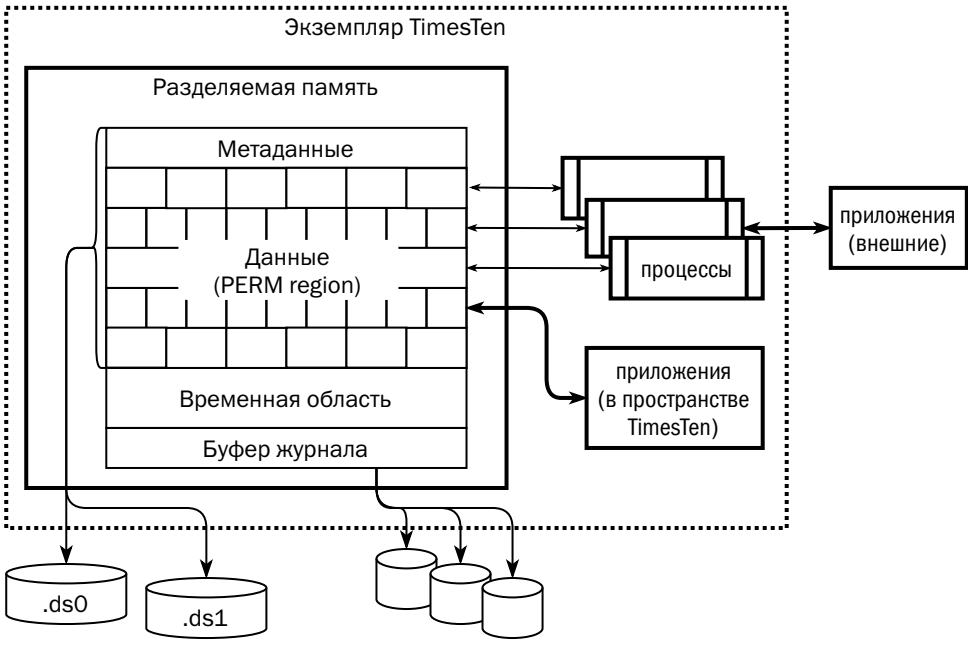
Практика показывает, что при правильной реализации второй подход эффективнее. Разница с первым подходом объясняется тем, что операционная система копирует целые страницы памяти, а СУБД — только собственные структуры, которые, как правило, имеют значительно меньший размер.

В отличие от файлов, отображённых в память, процесс сохранения может менять формат данных при записи на диск — например, сжимать данные.

Примеры СУБД в памяти

Oracle TimesTen — реляционная платформа, одна из старейших СУБД в памяти. Архитектурно она похожа на традиционные платформы, рассчитанные на работу с дисками:

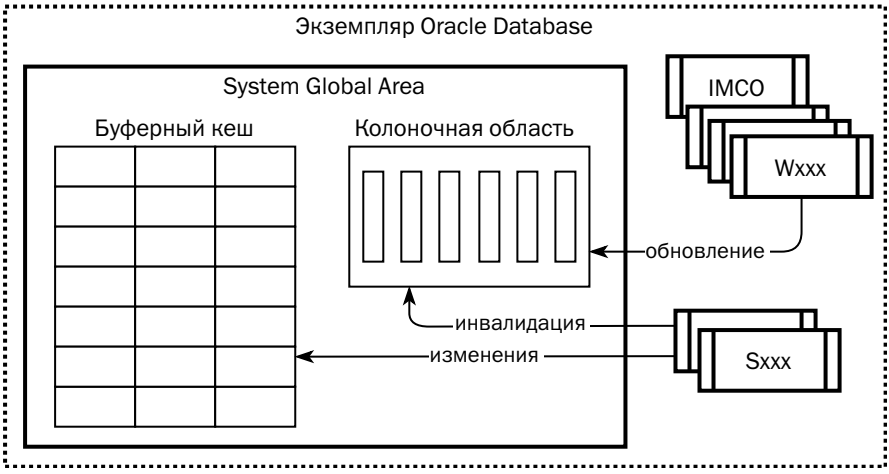
- Память, где хранятся данные (PERM region), разбита на страницы. Грязные (изменённые) страницы время от времени сохраняются в файлах снимков. Этих файлов всего два, и экземпляр использует их по очереди.
- Все данные, включая временную область и буферы журнала, располагаются в разделяемом сегменте памяти. Код сервера выполняется в нескольких процессах, которые конкурируют друг с другом за доступ к памяти.



Но есть и ряд отличий:

- страницы в памяти могут быть разного размера;
- изменения в индексах не записываются в журнал, поэтому если при восстановлении оказывается, что какой-то индекс должен быть изменён, этот индекс перестраивается целиком;
- клиентские приложения, требующие особенно быстрого отклика, могут запускаться на сервере БД и обращаться непосредственно к разделяемой памяти при помощи клиентской библиотеки TimesTen.

В отличие от TimesTen, **Oracle In-memory Option** — не самостоятельный продукт, а дополнение к флагманскому продукту — Oracle Database. К экземпляру Oracle добавляется отдельная область памяти, где данные хранятся в колоночном представлении:



Область колоночного хранения (columnar area) располагается в разделяемой памяти экземпляра (system global area — SGA) и имеет следующие отличия от буферного кеша:

- её размер фиксирован (в последних версиях администратор может её увеличивать без перезапуска экземпляра);
- данные хранятся не по строкам, как в буферах, а по колонкам;
- что именно в ней хранится, зависит не от того, к каким данным чаще всего обращаются, а от того, какие данные выберет администратор.

Набор данных колоночной области настраивается гибко: можно выбирать таблицы или их секции и колонки. После старта экземпляра фоновые процессы под управлением координатора (In-memory coordinator process, IMCO) начинают

наполнять колоночную область, считывая данные с диска. Впоследствии данные колоночной области используются для выполнения аналитических запросов. Заявление Oracle о том, что «In-memory Option ускоряет обработку транзакций в сотни раз», — некоторое лукавство. Исторически сложилось, что нагрузка на многие корпоративные приложения смешанная, то есть экземпляр не только выполняет короткие транзакции, но и строит отчёты, требующие чтения большого объёма данных. Для эффективного построения отчётов администраторы вынуждены создавать множество индексов, что существенно замедляет обработку пишущих транзакций. In-memory Option позволяет не только избавиться от «аналитических» индексов, но и эффективно выполнять аналитические запросы. Таким образом, многократное ускорение относится не к транзакционной, а к смешанной (и, разумеется, аналитической) нагрузке.

Для повышения эффективности обработки на колоночных данных строятся карты минимальных и максимальных значений (in-memory storage indexes). К данным применяется словарное и частотное сжатия.

Если транзакция изменяет какое-то значение, копия которого находится в колоночном хранилище, она помечает недействительной (инвалидирует) область, ограниченную с одной стороны изменённой колонкой, а с другой — набором строк. Аналитический запрос, которому нужно новое значение изменённых данных, может взять его из буферного кеша, пока фоновые процессы асинхронно не обновят колоночные данные.

СУБД **HANA (High performance ANalytical Appliance)** была выпущена компанией SAP в 2010 году как новая платформа для системы управления предприятием SAP ERP. Платформа базируется на ранее разработанных и купленных технологиях — TREX Search Engine (платформа поиска в колоночном хранилище), P*TIME (аналитическая БД в памяти) и СУБД MAX DB.

У HANA два механизма хранения данных — строчный и колоночный. Допускается их одновременное использование в одном экземпляре БД. В колоночном механизме применены все приёмы, описанные в разделе, посвящённом колоночному хранению, — буферная область для хранения изменений и все виды сжатия данных. Кроме того, колоночный механизм, в отличие от строчного, допускает шардирование базы данных на несколько узлов.

Если SAP R/3 выполняет всю обработку данных на серверах приложений, используя базу данных исключительно для простых операций чтения и записи, то SAP ERP на базе HANA, наоборот, активно использует хранимый код, выполняемый непосредственно на серверах БД.

Memcached был написан Брэдом Фитцпатриком (Brad Fitzpatrick) в 2003 году как кеш в памяти для проекта «Живой Журнал» (livejournal.com). Memcahced не является «базой данных» в привычном понимании: он представляет собой хранилище «ключ—значение», реализованное как хеш-таблица в памяти без возможности сохранения данных на диск. API Memcached состоит буквально из десятка команд доступа к данным и нескольких команд для мониторинга и управления экземпляром. Сегодня Memcached используется многими высоконагруженными сайтами, включая Twitter, Reddit и Facebook. API Memcached реализован в таких платформах, как Oracle Coherence, Hazelcast, MySQL, Couchbase, и других.

СУБД **Redis** (REmote DIctionary Server) впервые появилась в 2009 году. Сегодня это самая популярная СУБД в памяти, стабильно входящая в верхнюю десятку рейтинга сайта db-engines.com. Redis, как и Memcached, — хранилище «ключ—значение», однако возможности этой платформы несравненно шире:

- для восстановления после сбоев данные сохраняются на диск в виде журналов опережающей записи и периодических снимков, при создании которых порождаются новые процессы;
- в качестве значений поддерживаются сложные структуры данных — списки, ассоциативные массивы, множества, сортированные множества;
- есть возможность выполнять на сервере хранимые процедуры на языке Lua.

Redis не имеет собственной системы управления памятью, и его производительность зависит от того, с какой реализацией функции `malloc()` был скомпилирован исходный код.

Важная архитектурная особенность Redis заключается в том, что обработка данных выполняется в единственном потоке. Практика показывает, что при выполнении типовых задач, возлагаемых на СУБД в памяти, процессор не является узким местом — как правило, скорость обработки ограничивается производительностью памяти или дисков. Отказ от многопоточности позволяет существенно упростить код, а главное — отказаться от блокировок и, следовательно, ускорить обработку данных. Однопоточная модель доступа позволяет тем не менее запускать дополнительные потоки для вспомогательных операций — сохранения данных на диск, реорганизации памяти при удалении больших объектов и т. д.

Если производительности Redis не хватает для решения какой-либо задачи, следует разделить данные на шарды, запустив несколько экземпляров СУБД, пусть даже на одном сервере. При этом транзакционная целостность гарантируется только в том случае, когда все ключи, участвующие в транзакции, попадают в один шард.

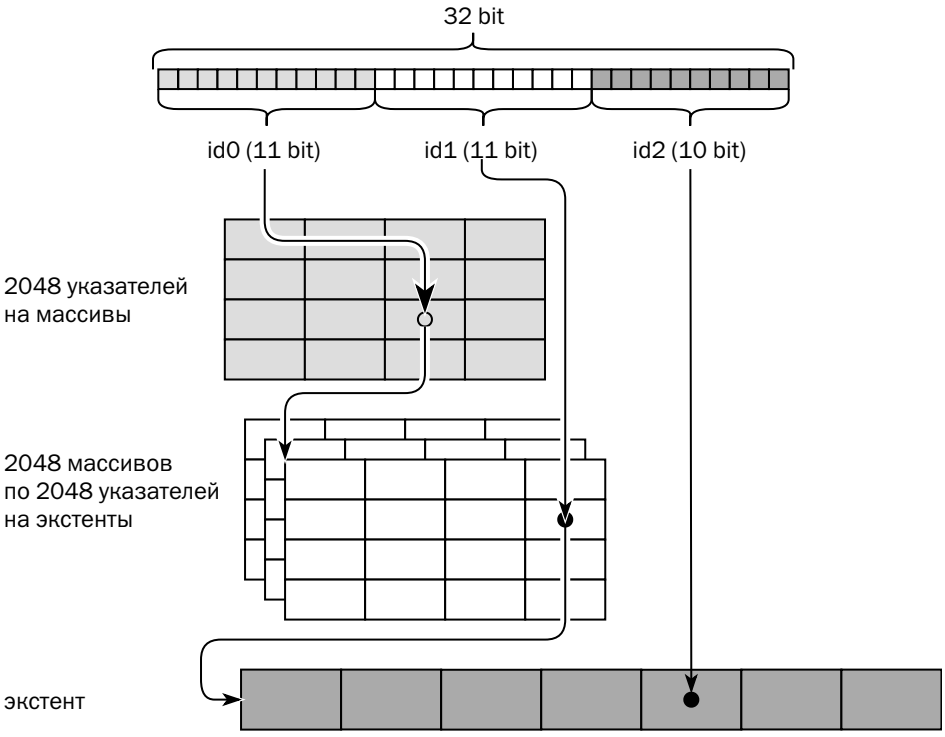
Платформа **Tarantool** (и её наследник, «форк» — Picodata) — практически ровесник Redis: её первый релиз состоялся в 2010 году. Архитектура Tarantool очень похожа на архитектуру Redis. Это тоже хранилище «ключ—значение». Сервер Tarantool однопоточный, для восстановления данных он также использует снимки и журналы опережающей записи. Как и в Redis, в Tarantool доступны хранимые процедуры (и триггеры!) на языке Lua.

Есть и ряд отличий:

- «значением» в Tarantool является кортеж (tuple), то есть набор атрибутов. Каждый атрибут, в свою очередь, может быть массивом (array), ассоциативным массивом (map) или иметь скалярный тип — целое число, число с плавающей точкой, строка, UUID;
- одинаковые кортежи хранятся в «пространствах» (space), что позволяет воспринимать данные Tarantool как реляционные и использовать для доступа к ним язык SQL (пространство — таблица, кортеж — строка). В платформе Picodata SQL уже является основным интерфейсом;
- Tarantool позволяет создавать вторичные индексы, причём поддерживается несколько типов индексов — B-дерево, хеш, R-дерево и битовая карта;
- JIT-компилятор Lua допускает загрузку дополнительных модулей, что позволяет использовать Tarantool одновременно в качестве хранилища данных и сервера приложений;
- при сохранении снимка данных делается снимок памяти средствами самой СУБД вместо копирования адресного пространства процесса при помощи вызова `fork()`;
- Tarantool допускает подключение нескольких движков хранения данных; к настоящему времени поставляются движки memtx для хранения данных в памяти и vinyl на основе LSM-деревьев для хранения данных на диске;
- помимо изоляции транзакций за счёт исполнения команд подряд в одном потоке появился менеджер транзакций, использующий версионирование данных.

В СУБД Tarantool реализована собственная подсистема управления оперативной памятью, обеспечивающая меньшие накладные расходы и большую производительность, чем использование стандартной библиотеки языка C. Важнейшим компонентом этой подсистемы является библиотека виртуализации памяти MATRAS (Memory Address TRANSlation)[1]:

[1] Буква N потерялась, и получилось русское слово «матрас».



Каждый указатель представляет собой 32-битное число. Чтобы получить реальный адрес в памяти, указатель проходит трёхступенчатую трансляцию:

- первые 11 бит используются как индекс в массиве из 2048 указателей на массивы;
- вторые 11 бит используются как индекс в массиве указателей на экстенты, адрес которого получен на первом этапе трансляции;
- последние 10 бит используются как индекс страницы в экстенте, адрес которого получен на втором этапе.

Страницы в разных экстентах могут быть разного размера, что позволяет выделять под каждый хранимый объект ровно столько памяти, сколько он занимает. За счёт того, что массивы, по которым осуществляется трансляция, целиком помещаются в кеш процессора, производительность «матраса» составляет десятки миллионов операций в секунду.

Использование системы трансляции позволяет:

- экономить память на указателях, используя 32-битные адреса вместо 64-битных;

- делать мгновенные снимки памяти прозрачно для высокоуровневых структур, использующих «матрас».

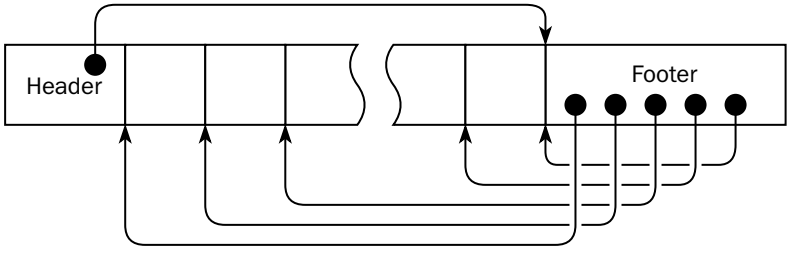
Транзакция, желающая сделать снимок памяти, просто создаёт ещё один заголовок «матраса» и включает его в список заголовков. В свою очередь если транзакция, меняющая данные, видит, что основной «матрас» в списке не один, она копирует объекты, которые собирается менять, включая массивы указателей. За счёт этого в разных транзакциях одни и те же идентификаторы разрешаются в разные адреса, где располагаются разные версии объектов.

Компания **GridGain**, выпускающая одноимённую платформу, была основана в 2007 году. С самого начала помимо коммерческой версии была доступна бесплатная версия. В 2014 году исходный код ядра системы был открыт, и проект был передан фонду Apache Software Foundation под названием Ignite.

Платформа Apache Ignite/GridGain относится к классу in-memory data grid и содержит три компонента:

- кластер хранения данных (data grid) — надёжно хранит данные в памяти серверов, входящих в кластер;
- вычислительный кластер (compute grid) — выполняет вычисления на узлах кластера. Платформа доставляет код на все узлы, где хранятся требующиеся для вычисления данные, вместо того чтобы перемещать данные к коду;
- исполнительный кластер (service grid) — управляет запуском сервисов, код которых загружен на узлы кластера; позволяет поддерживать постоянное количество экземпляров сервиса или автоматически масштабировать сервис, запуская и останавливая экземпляры.

GridGain — первая платформа, где была применена технология хранения данных вне кучи (off-heap memory). GridGain является платформой класса «ключ—значение», и значением является объект (экземпляр класса) Java. При передаче данных по сети, при записи на диск или в память вне кучи объекты в Java сериализуются. В GridGain реализован собственный сериализатор.



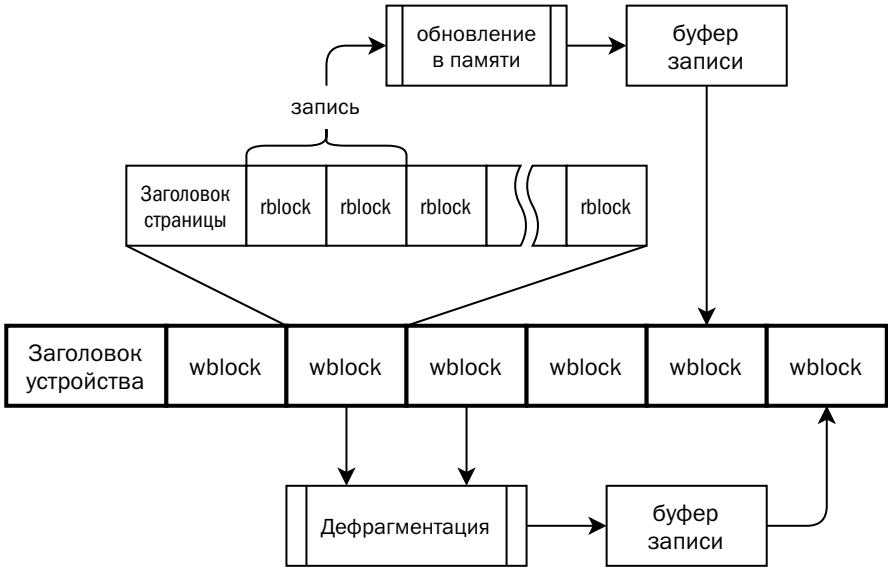
Заголовок сериализованного объекта содержит идентификатор класса и смещение хвоста объекта относительно заголовка, а хвост, в свою очередь, содержит ссылки на поля объекта. Если объект содержит ссылки на другие объекты, то объекты по ссылке сериализуются вместе с родительским объектом. Так, например, если записать в кеш дерево, то всё оно будет сериализовано целиком за один раз. Для обработки данных GridGain вынужден десериализовать объект, а если объект изменяется, то и сериализовать обратно. Поэтому для производительности важен размер класса: если класс большой, то возрастают накладные расходы на сериализацию и десериализацию, а если маленький — расходы на хранение, поскольку размер заголовка составляет 24 байта. Важно также помнить, что сериализованный объект содержит только данные объекта, но не определение класса, поэтому приложение должно загрузить определение класса самостоятельно.

Помимо сериализованных объектов Java GridGain позволяет хранить произвольные наборы байтов, содержимое которых клиент интерпретирует самостоятельно.

Память GridGain разбита на страницы, что позволяет делать инкрементальные снимки памяти, записывая на диск только изменённые страницы, а при необходимости вообще вытеснять редко используемые страницы на диск. Страницы размечены либо как неупорядоченные таблицы, либо как B-деревья (для индексов, используемых встроенным движком SQL). Архитектура системы хранения GridGain практически не отличается от архитектуры системы хранения традиционных дисковых баз данных с изменяемыми структурами.

Платформа **Aerospike** ведёт свою историю с 2010 года. Первые версии хранили данные исключительно в памяти, последние версии с гибридной архитектурой памяти рассчитаны на использование современных устройств хранения — флеш-памяти и энергонезависимой памяти (non-volatile memory, например Intel Optane). Эти устройства обеспечивают быстрый произвольный доступ к данным любой длины, но запись на них производится блоками фиксированного размера. Aerospike учитывает эти особенности и работает с устройствами напрямую, в обход файловой системы.

Архитектура системы хранения Aerospike похожа на движок Bitcask в Riak KV. Устройство делится на страницы (wblocks) одинакового размера, а каждая страница, в свою очередь, делится на блоки (rblocks) по 128 байт. Данные внутри страницы не перезаписываются: если значение меняется, то процесс записи сохраняет новую версию данных в буфере, занимая целое количество блоков, а когда объём данных в буфере достигает размеров страницы, данные сохраняются



в постоянное хранилище в новую страницу. Фоновый процесс дефрагментации собирает записи из блоков, где почти не осталось актуальных записей, и записывает их в новые блоки, помечая старые как свободные. За счёт того, что блоки перезаписываются по кругу, флеш-память изнашивается равномерно[1]. В памяти хранятся индексы, кеш и актуальная карта свободных дисковых блоков. Начиная с релиза 4.5 Aerospike умеет хранить индекс в энергонезависимой памяти, что существенно снижает стоимость хранения без потери производительности.

Мы видим, что с классификацией СУБД по среде хранения происходит то же самое, что с классификацией по модели данных: границы между классами систем постепенно размываются. Базы данных в памяти используют диск как для обеспечения надёжности, так и для хранения данных. Пожалуй, наиболее точное определение СУБД в памяти в противовес дисковой СУБД сегодня звучит так: **платформа, рассчитанная на эффективное использование большого объёма памяти**. «Большой объём» означает иное соотношение между объёмами, стоимостью и производительностью памяти и дисков, чем во времена разработки дисковых СУБД. Это соотношение диктует новые компромиссные решения с иными узкими местами.

[1] Контроллер SSD может изменять физический адрес блока с целью равномерного износа флеш-памяти; тем не менее перезапись данных по кругу подаётся компанией Aerospike как преимущество.

Литература

- Manos Athanassoulis, Michael S. Kester, Lukas M. Maas, Radu Stoica, Stratos Idreos, Anastasia Ailamaki, Mark Callaghan. Designing Access Methods: The RUM Conjecture.

  stratos.seas.harvard.edu/files/stratos/files/rum.pdf
- Physical location of a row in SQL Server.

  www.codeproject.com/Articles/159785/Physical-location-of-a-row-in-SQL-Server
- What is the Equivalent of ROWID in PostgreSQL?

  www.enterprisedb.com/postgres-tutorials/what-equivalent-rowid-postgresql
- RID and RID_BIT scalar functions.

  www.ibm.com/docs/en/db2/11.5?topic=functions-rid-rid-bit
- InnoDB: Tablespace Space Management.

  mysqlserverteam.com/innodb-tablespace-space-management/
- Руководство по архитектуре страниц и экстентов.

  docs.microsoft.com/ru-ru/sql/relational-databases/pages-and-extents-architecture-guide?view=sql-server-ver15
- Создание кластеризованных индексов.

  docs.microsoft.com/ru-ru/sql/relational-databases/indexes/create-clustered-indexes?view=sql-server-ver15
- Db2: CREATE TABLE statement.

  www.ibm.com/docs/en/db2/11.5?topic=statements-create-table
- MySQL: CREATE TABLE Statement.

  dev.mysql.com/doc/refman/8.0/en/create-table.html
- Clustered and Secondary Indexes.

  dev.mysql.com/doc/refman/8.0/en/innodb-index-types.html
- How Compression Works for InnoDB Tables.

  dev.mysql.com/doc/refman/8.0/en/innodb-compression-internals.html
- How PCTFREE and PCTUSED Work Together.

  docs.oracle.com/en/database/oracle/oracle-database/21/sqlrf/physical_attributes_clause.html#GUID-A15063A9-3237-43D3-B0AE-D01F6E80B393
- Реализация сжатия страниц.

  docs.microsoft.com/ru-ru/sql/relational-databases/data-compression/page-compression-implementation?view=sql-server-ver15
- What does the ‘B’ stand for in B-Tree?

  www.quora.com/What-does-the-B-stand-for-in-B-Tree

- Julian Dyke. Bitmap Index Internals.
  www.juliandyke.com/Presentations/BitmapIndexInternals.ppt
- Егор Рогов. PostgreSQL 15 изнутри. М.: ДМК Пресс, 2023.
  postgrespro.ru/education/books/internals
- Ben Stopford. Log Structured Merge Trees.
  www.benstopford.com/2015/02/14/log-structured-merge-trees/
- Константин Осипов. Хранение данных на виниле.
  habr.com/ru/company/mailru/blog/358210/
- LevelDB Files.
  github.com/google/leveldb/blob/master/doc/impl.md
- Bigtable: A Distributed Storage System for Structured Data.
  storage.googleapis.com/pub-tools-public-publication-data/pdf/68a74a85e1662fe02ff3967497f31fda7f32225c.pdf
- Memtable.
  github.com/facebook/rocksdb/wiki/MemTable
- Leveled Compaction.
  github.com/facebook/rocksdb/wiki/Leveled-Compaction
- Oracle Berkeley DB Programmer's Reference Guide.
  docs.oracle.com/cd/E17076_05/html/programmer_reference/BDB_Prog_Reference.pdf
- Justin Sheehy, David Smith. Bitcask: A Log-Structured Hash Table for Fast Key/Value Data.
  riak.com/assets/bitcask-intro.pdf
- Mark Callaghan. Index+log, v2.
  smalldatum.blogspot.com/2019/05/indexlog-v2.html
- Vincent Kulandaisamy. Db2 LUW with BLU Acceleration: Internals Concepts & Best Practices.
  wdug.com/resources/Documents/LUW Presentations/March 2019/Db2 BLU Internals Concepts and Best Practices updated for V11 1.pdf
- Keri Romanufa. Db2 Architecture. Overview and BLU.
  www.tridex.org/wp-content/uploads/Db2-Arch-Overview-and-BLU.pdf
- Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, Samuel Madden. The Design and Implementation of Modern Column-Oriented Database Systems.
  stratos.seas.harvard.edu/files/stratos/files/columnstoresfntdbs.pdf
- Индексы Columnstore. Обзор.
  docs.microsoft.com/ru-ru/sql/relational-databases/indexes/columnstore-indexes-overview?view=sql-server-ver15

- Alexandra Wang, Soumyadeep Chakraborty. ZedStore.

  www.pgcon.org/events/pgcon_2020/sessions/session/44/slides/13/Zedstore-PGCon2020-Virtual.pdf

- SAP IQ Zone Maps.

  help.sap.com/viewer/a8982cc084f21015a7b4b7fcdeb0953d/16.1.4.2/en-US/6604c2567d66453da391dee00dcf5d5c.html

- Vijayshankar Raman, Gopi Attaluri, Ronald Barber et al. Db2 with BLU Acceleration: So Much More than Just a Column Store.

  vldb.org/pvldb/vol6/p1080-barber.pdf

- Sándor Héman, Marcin Zukowski, et. al. Positional update handlinig in column stores.

  pdfslide.net/documents/positional-update-handling-in-column-stores.html

- Teradata Database Hardware and Software Architecture.

  docs.teradata.com/r/6Om32fxkCGM2_NXNgCnwWg/86ooNLqv9AoEZj2~W03Q4g

- Juan Pablo Carzolio. A Guide to Consistent Hashing.

  www.toptal.com/big-data/consistent-hashing

- Jason Resch. New Hashing Algorithms for Data Storage.

  www.snia.org/sites/default/files/SDC15_presentations/dist_sys/Jason_Resch_New_Consistent_Hashings_Rev.pdf

- Gartner DBMS Market Share Ranking 2011–2018.

  www.chinainternetwatch.com/29486/top-15-dbms-2018/

- Karthik Ranganathan. Four Data Sharding Strategies We Analyzed in Building a Distributed SQL Database.

  www.yugabyte.com/blog/four-data-sharding-strategies-we-analyzed-in-building-a-distributed-sql-database/

- Chris Jenkins. Oracle TimesTen In-Memory Database Architecture, Performance Tips, Use Cases.

  indico.cern.ch/event/757894/attachments/1720580/2777515/7c_TimesTen_ChrisJenkins.pdf

- Oracle In-Memory Column Store Architecture.

  docs.oracle.com/en/database/oracle/oracle-database/21/inmem/in-memory-column-store-architecture.html

- Discussion on different memory allocators tcmalloc and jemalloc used in Redis.

  developpaper.com/discussion-on-different-memory-allocators-tcmalloc-and-jemalloc-used-in-redis/

- Константин Осипов. Что особенного в СУБД для данных в оперативной памяти.

  habr.com/ru/company/oleg-bunin/blog/310560/
- Ivan Pavlukhin. Ignite Binary object format.

  cwiki.apache.org/confluence/display/IGNITE/Binary+object+format
- Dmitry Pavlov. Ignite Durable Memory — under the hood.

  cwiki.apache.org/confluence/display/IGNITE/Ignite+Durable+Memory+-+under+the+hood
- Aerospike Hybrid Memory Architecture.

  aerospike.com/products/features/hybrid-memory-architecture/
- Accelerated Real-Time Computing at Petabyte Scale.

  www.intel.ru/content/dam/www/public/us/en/documents/solution-briefs/aerospike-enterprise-database-solution-brief.pdf
- Ankur Goyal. MemSQL DB Class.

  www.slideshare.net/MemSQL/memsql-db-class-ankur-goyal

Глава 4
Обработка данных

> Не умеющему соображать информация к размышлению ни к чему.
>
> *В. Л. Гавеля, «Отзвуки житейской мудрости»*

Хранение данных — важная функция систем управления базами данных, но данные сами по себе никому не интересны. Нужна информация, которую скрывают в себе данные. В этом разделе рассмотрим процесс преобразования данных с целью извлечения информации и поговорим, как оптимизировать этот процесс, чтобы приложение могло работать наиболее эффективно.

4.1. Поиск данных

Поиск данных в структурах на основе LSM-дерева

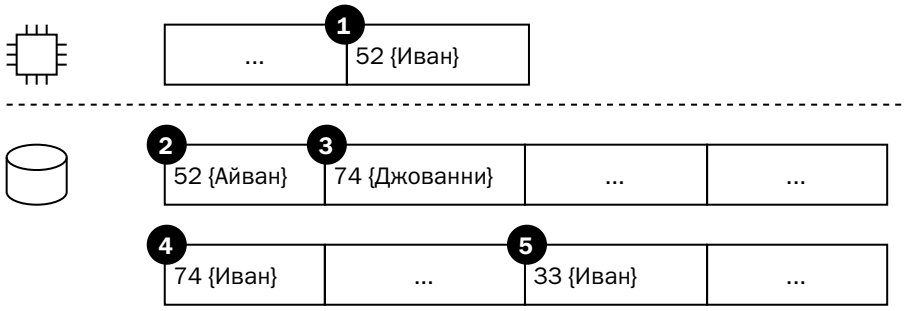
Рассмотрим процесс поиска объекта по ключу в хранилище на основе LSM-дерева:

1) поиск ключа и соответствующего ему объекта в буфере в оперативной памяти;
2) поиск ключа и соответствующего ему объекта в кеше в оперативной памяти, куда попадают наиболее часто запрашиваемые объекты;
3) поиск ключа и соответствующего ему объекта в файлах:
    - поиск файлов, в которых может находиться искомый ключ: просматриваются фильтры Блума для всех файлов первого уровня и для тех файлов следующих уровней, диапазоны ключей которых содержат искомый ключ; по возможности СУБД кеширует фильтры Блума в оперативной памяти;
    - поиск ключа в файлах, где он может находиться: в каждом из найденных на предыдущем шаге файлов просматривается индекс — либо в кеше индексов в оперативной памяти, либо на диске;
    - извлечение объекта из файла по найденному ключу.

Количество операций ввода-вывода при таком поиске слабо поддаётся оценке, поскольку оно зависит не только от самой структуры дерева, но и от профиля данных. Запись в LSM-дерево реализована значительно эффективнее, чем чтение.

При поиске значения по неключевым атрибутам СУБД вынуждена будет прочесть все файлы, каждый раз проверяя, не попадался ли найденный ключ ранее.

Предположим, что требуется найти все записи, в которых встречается имя Иван:



Найдя в буфере элемент с номером 1, движок должен зафиксировать, что ключ найден, и уже не обращать внимания на элемент с номером 2, находящийся в файле первого уровня, т. к. этот элемент представляет собой недействительную версию объекта с ключом 52. Прочитав из того же файла элемент с номером 3, движок должен зафиксировать и его ключ, несмотря на то что значение атрибута не совпадает с искомым. Если этого не сделать, то движок может выдать в ответ на запрос элемент с номером 4, который к моменту выполнения запроса уже не актуален. И наконец, движок вынужден прочитать все файлы до конца, в противном случае элемент 5 не будет найден.

Зная о такой неэффективности, некоторые платформы позволяют искать данные только по ключевым атрибутам или по тем атрибутам, по которым построен вторичный индекс.

В отличие от LSM-дерева, такие структуры, как неупорядоченная таблица или B-дерево, допускают гораздо больше вариантов доступа. О них и будем говорить дальше.

Поиск в неупорядоченной таблице

Далее в этом разделе исходные английские термины приводятся только там, где все производители промышленных платформ понимают их более или менее одинаково. В остальных случаях, чтобы не превращать раздел в сложную матрицу,

предоставим читателю самостоятельно выстроить соответствие между универсальными русскими терминами и терминами любимой платформы.

Итак, первый метод доступа — **чтение единственной строки по физическому идентификатору**. Напомню, что физический идентификатор состоит из адреса страницы, в которой находится строка, и номера строки внутри страницы. Как правило, чтению строки предшествует поиск ключа в индексе, откуда и берётся физический идентификатор строки в таблице[1].

При чтении строки выполняются следующие действия:

- поиск страницы, на которую указывает идентификатор, в буферном кеше;
- логическое чтение страницы, если она найдена в кеше;
- физическое чтение страницы, если она не найдена в кеше.

При **логическом чтении (logical read)** на страницу накладывается кратковременная блокировка[2], чтобы конкурирующая транзакция не изменила её во время чтения, затем содержимое страницы изменяется или, если необходимо, данные копируются из страницы в локальную память процесса, и блокировка снимается.

При **физическом чтении (physical read)** читающий процесс находит в кеше свободный буфер или буфер, который можно освободить, считывает содержимое страницы с диска в кеш, выполняет необходимые действия над прочитанными данными и оставляет страницу в кеше, где она становится доступна другим процессам. Страница может быть считана и из кеша операционной системы, но с точки зрения СУБД никакой разницы между чтением с диска и чтением из кеша ОС нет. Разумеется, общая производительность системы зависит от размера кеша ОС, но рекомендации производителей баз данных по его настройке существенно отличаются.

Второй метод доступа к данным в неупорядоченной таблице — **полный просмотр**. Сервер читает содержимое таблицы страницу за страницей, в каждой прочитанной странице перебирает все строки и, если строка удовлетворяет условию поиска, возвращает её вызывающей стороне. Очевидно, при полном просмотре выполняется чтение (логическое или физическое) всех страниц таблицы.

Если при полном просмотре работать со страницами так же, как при чтении единственной строки, то к концу операции таблица займёт значительную часть

---

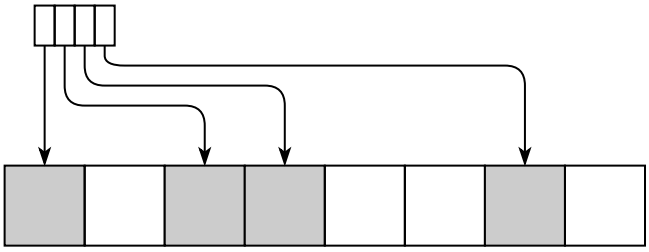
[1] Если в PostgreSQL в запросе к таблице явно указать идентификатор строки (на самом деле — версии строки), то в плане выполнения запроса эта операция (tuple id scan) будет показана. А если поиску предшествует извлечение идентификатора из индекса, то нет.

[2] В англоязычной документации для кратковременной блокировки используются термины, отличные от «lock». В Oracle и Db2 это «latch», а в PostgreSQL и Microsoft SQL Server — «spin-lock».

кеша, вытеснив оттуда другие объекты, возможно, более нужные. В большинстве случаев такой эффект нежелателен, и некоторые платформы предлагают механизмы борьбы с ним:

- PostgreSQL при полном сканировании большой таблицы («большой» считается таблица, занимающая больше четверти буферного кеша) выделяет в буферном кеше специальную область для загрузки страниц; прочитанные страницы занимают буферы только в выделенной области;
- Oracle при создании таблицы позволяет указать одно из ключевых слов, `CACHE` или `NOCACHE`; если таблица создана с атрибутом `NOCACHE`, то при полном чтении буферы с её страницами вместо обычного помещения в конец списка будут помещаться в начало списка давно не использовавшихся буферов (LRU-list, least recently used), откуда страницы вытесняются в первую очередь;
- при многопоточном чтении в процессе выполнения аналитических запросов (direct path read) Oracle вообще не помещает прочитанные страницы в буферный кеш, если только такое кеширование не включено специально[1].

Третий метод доступа к неупорядоченной таблице — **частичный просмотр**. Читающий процесс строит в памяти список страниц, в которых могут храниться интересующие его строки, а если места в памяти достаточно, то и список идентификаторов конкретных строк:



Как и при чтении единственной строки, чтению предшествует поиск строк в одном или нескольких индексах. Преимущество частичного просмотра перед чтением индекса с последующей выборкой строк по одной в том, что каждая нужная страница основной таблицы читается один раз независимо от того, сколько искомых записей в ней расположено. Недостаток — расход памяти на хранение списка страниц и пауза перед выдачей первой строки, во время которой строится список.

[1] См. Automatic Big Table Caching.

На примере такой простой структуры, как неупорядоченная таблица, можно увидеть, почему лишён смысла вопрос «какой метод лучше всех». Разумеется, единственная запись быстрее всего будет извлечена по физическому идентификатору, а все записи из таблицы быстрее извлечь полным просмотром — даже если в запросе, использующем поиск по физическому идентификатору, идентификаторы будут перечислены явно, а не будут предварительно извлекаться из индекса. Где-то между этими крайними случаями есть точка, в которой оба метода одинаково эффективны (или одинаково неэффективны), но конкретное положение этой точки зависит от многих факторов — насколько плотно заполнены страницы, насколько быстро работает кеш, сколько в нём свободного места и т. д. Частичный просмотр в большинстве случаев выглядит разумным компромиссом, но границы этого «большинства случаев» также зависят от конкретной базы данных. О том, какой же метод оптимален, поговорим позднее в этом разделе, а о том, как выбрать оптимальный метод в работающей базе, — в разделе 4.3 «Оптимизация запросов».

Индексирование

Индекс — вспомогательная структура, которая создаётся для ускорения поиска данных.

Упоминание об индексах встречается в разных частях книги, но системно рассказать о них сложно. Поэтому подраздел, посвящённый индексированию, выглядит как набор фактов, слабо связанных друг с другом, но важных для понимания дальнейшего материала.

Подавляющее большинство индексов построены на базе B-деревьев. Разумеется, это не единственная и не всегда лучшая структура для индекса, но в большинстве случаев именно она оказывается оптимальной.

Индекс представляет собой набор пар, состоящих из индексируемого значения и ссылки на строку (или объект), содержащую это значение. Если основные данные лежат в неупорядоченной таблице, то ссылка представляет собой физический идентификатор, непосредственно указывающий на место строки на диске. Если же основная структура для хранения данных — дерево (B-дерево или LSM-дерево), то ссылка представляет собой ключ, поиск по которому занимает некоторое время. Отсюда следует, что оптимальная структура для основного хранилища при необходимости искать данные в таблице по разным критериям и при наличии множества индексов — именно неупорядоченная таблица.

Важная характеристика для индексируемой колонки — **кардинальность**, то есть мера уникальности данных. Формально кардинальность можно определить как отношение количества уникальных значений к общему количеству записей: например, кардинальность колонки «пол» низкая (два значения на всю таблицу[1]), а кардинальность колонки «номер телефона» высокая — это практически (но не обязательно) уникальное значение. B-дерево хорошо работает на колонках с высокой кардинальностью. Для колонок с низкой кардинальностью лучше использовать другие структуры — битовые карты и инвертированные индексы.

Прежде чем строить индексы, следует изучить их реализацию в конкретной платформе. Так, например, в Apache Cassandra не рекомендуется строить индексы не только на колонках с низкой кардинальностью (например, типа `boolean`), но и на колонках со «слишком высокой» кардинальностью (например, индексирование песен по автору вместо индексирования по исполнителю).

В некоторых случаях (например, в слое детальных данных аналитического хранилища) поиск по индексу не нужен, однако индекс всё равно создаётся — для гарантии уникальности первичного ключа. Такие индексы можно создавать не по значению ключа, а по его инвертированному представлению — например, вместо 1234 хранить 4321. Индекс по инвертированному представлению равномерно распределяет нагрузку между узлами B-дерева: если учесть, что обычно значение первичного ключа монотонно увеличивается, в обычном индексе при вставке данных возникает высокая конкуренция за крайний правый листовой узел[2]. Oracle может автоматически переворачивать значение ключа, если при создании индекса указать ключевое слово `REVERSE`.

Индексы занимают дисковое пространство, замедляют вставку и удаление данных, а также обновление, если оно касается индексированных колонок, поэтому строить индексы «на всякий случай» не стоит. Более того, промышленные базы данных следует время от времени проверять на наличие неиспользуемых индексов. Все современные реляционные платформы предоставляют для этого встроенные механизмы.

Индекс, основанный на B-дереве, позволяет искать данные не только по полному значению ключа, но и по начальной части значения. Так, например, если

---

[1] В англоязычной литературе могут встречаться и другие значения.

[2] Прежде чем применять рекомендацию в конкретной платформе, рекомендуется изучить фирменную документацию и провести нагрузочный тест. Так, например, в PostgreSQL есть механизм «Fastpath For Index Insertion», особым образом обрабатывающий массированную вставку монотонно возрастающих значений.

на таблице городов построен индекс по названию, то можно быстро найти «Нижний…» и уже потом выбирать между Нижним Новгородом и Нижним Тагилом. Но при поиске строки, заканчивающейся на «…Новгород», индекс не поможет — придётся просмотреть таблицу полностью.

Индекс может быть составным, то есть включать в себя несколько колонок, и в этом случае можно также искать данные по значению первых колонок. Порядок перечисления колонок при построении индекса имеет значение.

Важно понимать, что если набор данных упорядочен по значению какой-то колонки, то из этого вовсе не следует, что он упорядочен по значению функции от этой колонки. Весьма распространённая ошибка — построить индекс по дате и искать данные за период по значению функции — например, `trunc()` и `to_char()` в Oracle или `date_part()` и `extract()` в PostgreSQL. Для эффективного поиска по индексу нужно или накладывать в запросе условие непосредственно на дату, или строить индекс по значению функции. Все современные реляционные платформы позволяют строить индексы по значению выражения, но конкретно в описанном случае лучше этого не делать. Индекс по дате гораздо более универсален, чем индекс по какой-то части даты, и позволяет ускорить гораздо большее количество разных запросов.

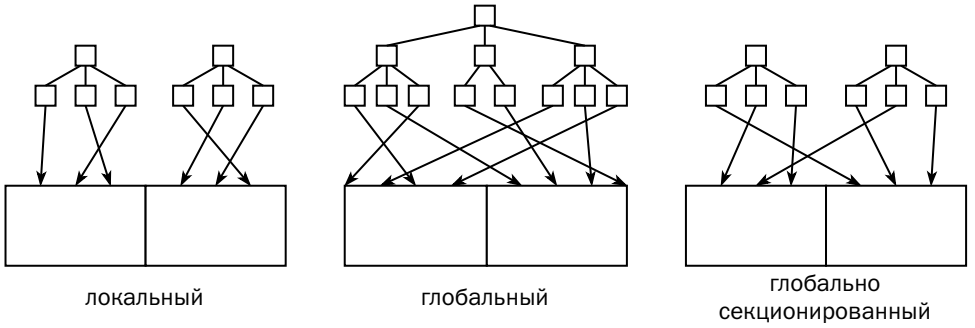
В некоторых случаях после поиска в индексе обращение к таблице не нужно, т. к. все необходимые колонки уже есть в самом индексе. В этом случае индекс называется покрывающим (covering) для запроса. Планируя создание индексов, иногда имеет смысл добавить после колонок, по которым требуется поиск, дополнительные колонки, которые чаще всего извлекаются запросами.

В Db2, PostgreSQL и Microsoft SQL Server термин «покрывающий индекс» обозначает индекс, к которому добавлены дополнительные колонки. В отличие от «обычного» индекса, значения дополнительных колонок покрывающего индекса не участвуют в сортировке элементов индекса. Преимущество, которое даёт этот механизм, — возможность расширять уникальные индексы.

При создании индекса программист может объявить его уникальным (unique). На физическом уровне уникальный индекс ничем не отличается от неуникального — любой индекс работает с уникальными ключами. В «обычном» (неуникальном) индексе к ключу добавлен идентификатор строки, соответствующей ключу, чтобы сделать его уникальным. В зависимости от реализации записи с одинаковыми значениями ключевых колонок в неуникальном индексе могут располагаться либо в порядке поступления, либо в порядке возрастания идентификаторов строк.

Возможна ситуация, когда некоторое подмножество данных в таблице представляет наибольший интерес. Например, неоплаченные счета или сетевые сеансы с IP-адресами, не относящимися к частной сети. PostgreSQL и Microsoft SQL Server позволяют построить выборочный индекс, в который попадут только строки, удовлетворяющие условию[1]. В PostgreSQL такие индексы называются частичными (partial), а в Microsoft SQL Server — отфильтрованными (filtered). Набор индексируемых колонок и набор колонок, по которым отбираются данные, не зависят друг от друга.

Если таблица секционирована, то индексы на такой таблице с точки зрения локализации данных могут быть локальными (local), глобальными (global) и глобально секционированными (globally partitioned).



Локальные индексы разбиты на секции, причём каждая секция такого индекса индексирует строки соответствующей секции таблицы. При добавлении секции в таблицу соответствующая секция добавляется и в локальный индекс, а при удалении секции из таблицы — удаляется и из индекса. Если локальный индекс объявлен как уникальный, то ключ секционирования должен входить в набор уникальных колонок, иначе СУБД не сможет гарантировать уникальность.

Если условия поиска не содержат ограничений на ключ секционирования, то эффективность локального индекса невелика, поскольку приходится просматривать все его секции. В этом случае, а также если требуется глобально уникальный идентификатор, не содержащий ключ секционирования (например, глобально

[1] В Oracle можно добиться похожего эффекта, построив индекс по выражению, которое принимает значение null, если строка не удовлетворяет условию поиска (значение null в Oracle не индексируется). Но и искать данные надо будет по значению этого же выражения, а не по значению колонки. Кроме того, в Oracle можно построить секционированный индекс, часть секций которого отсутствует.

уникальный идентификатор в таблице банковских транзакций, секционированной по дате совершения транзакции), на помощь приходит **глобальный индекс**, то есть монолитный индекс, который индексирует всю таблицу. Использование глобальных индексов считается «плохим тоном», и, например, сообщество PostgreSQL так и не пришло к решению о необходимости включения в свою платформу такой возможности. Дело в том, что при удалении секции из таблицы в индексе мгновенно появляется множество элементов, указывающих на несуществующие строки (orphaned entries). В ранних версиях Oracle это приводило к объявлению глобального индекса недействительным (инвалидации) и последующей его перестройке. В Db2 и современных версиях Oracle после удаления секции запускается асинхронный процесс очистки глобальных индексов. Microsoft SQL Server вообще не позволяет удалять секции из секционированной таблицы, если на ней определены глобальные индексы.

Глобально секционированный индекс тоже секционирован, но ключ его секционирования не совпадает с ключом секционирования таблицы (а в некоторых платформах индексируемая таблица может и не быть секционированной). Такая структура сочетает в себе недостатки локальных и глобальных индексов, но иногда используется для снижения конкуренции за блоки индекса при интенсивном изменении данных.

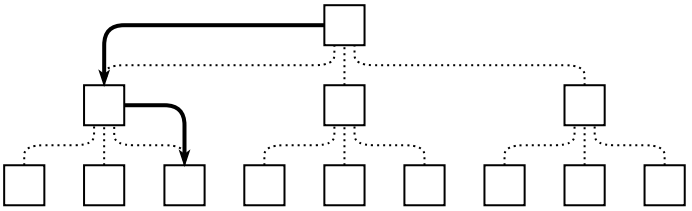
В таблице ниже указано, какие схемы локализации индексов поддерживают современные реляционные платформы.

	Oracle	PostgreSQL	MS SQL Server	Db2	MySQL
Локальные индексы	+	+	+	+	+
Глобальные индексы	+	–	+	+	–
Глобально секционированные индексы	+	–	+	–	–
Секционированные индексы на несекционированных таблицах	+	–	+	–	–

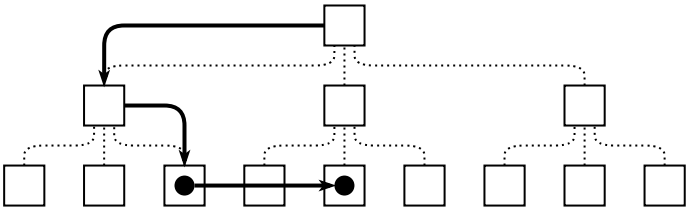
Поиск в B-дереве

B-дерево даёт больше возможностей для поиска, чем простая неупорядоченная таблица.

Самый простой метод доступа к B-дереву — **поиск уникального значения**. Этот метод используется, когда нужно извлечь из таблицы строку по первичному (или просто уникальному) ключу. Читающий процесс просматривает дерево, спускаясь от корня к листу, и возвращает единственное значение, или не возвращает ничего. За время спуска процесс просматривает H страниц, где H — глубина (высота) дерева, то есть количество уровней[1] (у дерева на рисунке ниже три уровня). Страница, соответствующая корню дерева, практически наверняка будет найдена в кеше, но по мере спуска по дереву вероятность обнаружить требуемую страницу в кеше уменьшается.



Следующий метод доступа — **поиск множества ключей**. Этот метод используется, когда индекс не является уникальным либо требуется поиск по части ключа или по диапазону ключей. Как и при уникальном сканировании, читающий процесс спускается по дереву до листа, в котором находится первое из искомых значений, а потом двигается по листам дерева вправо, пока значение ключа удовлетворяет условию поиска. Количество просмотренных страниц зависит от ширины запрашиваемого диапазона и от статистического распределения значений ключа.



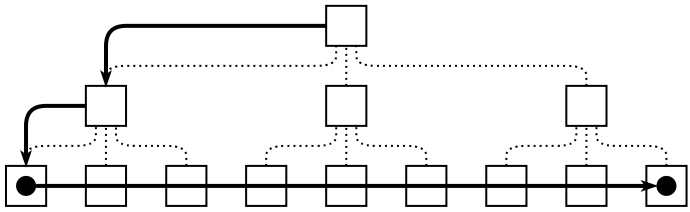
Важно, что при поиске по диапазону результат получается отсортированным по возрастанию значений ключа, и если результат запроса должен быть отсортирован, то за счёт использования индекса сортировки можно избежать. Чаще

[1] В некоторых источниках глубиной дерева называют количество спусков, которое всегда на единицу меньше, чем количество уровней.

всего запрос просматривает индекс слева направо, в порядке возрастания значений ключа, но в некоторых случаях (например, требуется обратная сортировка, или условие имеет вид `id < N`) запрос может спуститься к листу с наибольшим значением и просматривать листы справа налево — в порядке убывания значений.

Если сортировка результатов не требуется, то при поиске по диапазону процесс может накапливать в памяти найденные ключи и сортировать их в порядке возрастания физического адреса строки. Накопленный результат занимает место в памяти процесса, но зато позволяет впоследствии воспользоваться частичным просмотром таблицы, к которой относится индекс, вместо того чтобы многократно извлекать из неё по одной строке.

Достаточно экзотический метод доступа к B-дереву — **полный просмотр**. Читающий процесс спускается от корня к самому левому листу и просматривает все листы в порядке возрастания значений (или наоборот — спускается к самому правому листу и двигается влево, в порядке убывания). При этом методе будут просмотрены все листовые узлы, то есть практически всё дерево. Полный просмотр индекса оправдан в единственном случае — когда результат должен быть отсортирован, но по каким-то причинам СУБД[1] хочет избежать сортировки в памяти.

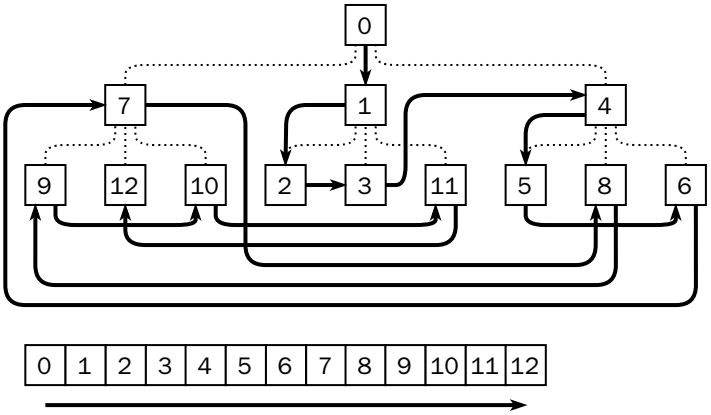


Гораздо чаще, чем полный просмотр, используется **быстрый просмотр** B-дерева. При быстром просмотре считываются все страницы дерева, но не в порядке возрастания или убывания значений ключа, а в том порядке, в котором эти страницы находятся на диске. Быстрый просмотр намного производительнее полного просмотра:

- требуется меньше операций ввода-вывода, т. к. одна операция может считывать несколько подряд идущих страниц;
- просмотр может выполняться параллельно несколькими потоками.

[1] Точнее, компонент под названием «оптимизатор», речь о котором — в разделе 4.3 «Оптимизация запросов».

Быстрый просмотр B-дерева очень похож на полный просмотр неупорядоченной таблицы. При полном просмотре свойство упорядоченности никак не используется: с точки зрения структуры B-дерева узлы обходятся в случайном порядке, и результат получается неупорядоченным.



Если поиск по диапазону невозможен (например, первое из индексированных полей не участвует в условии), то быстрый просмотр индекса может оказаться эффективнее полного просмотра таблицы в двух ситуациях:

- индекс является покрывающим для запроса (или индекс и есть таблица);
- требуется поиск по какому-нибудь редкому значению неключевого поля, входящего в индекс.

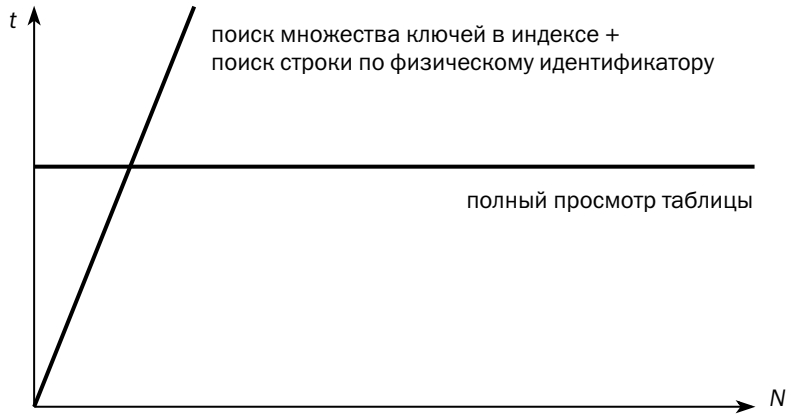
Пусть, например, в воинской части есть кадровая база, в которой построен индекс по фамилии и полу. Если запрос должен вернуть всех женщин, то лучшим способом выполнить такой поиск может оказаться частичный просмотр таблицы, список страниц для которого получен быстрым просмотром индекса.

В некоторых случаях при поиске могут использоваться несколько индексов одновременно. Читающий процесс просматривает индексы по очереди, а затем выбирает те идентификаторы строк (или страницы, если выборки получаются слишком объёмные), которые встречаются в обеих выборках или хотя бы в одной выборке — в зависимости от условий поиска. Такая операция получила название «соединение индексов» (index join), хотя она не имеет никакого отношения к соединению таблиц. Как обычно, некоторые производители используют свою терминологию — в MongoDB и Microsoft SQL Server эта операция называется index intersection, в MySQL — index merge, в PostgreSQL — bitmap and/or.

Самый быстрый способ поиска

Самый распространённый способ хранения данных в современных дисковых реляционных платформах, использующих изменяемые структуры, — неупорядоченная таблица и индексы на основе B-деревьев. Если колонка, по значению которой мы пытаемся отобрать данные, не проиндексирована, то единственный доступный метод поиска — полный просмотр таблицы. Но какой метод лучше, если колонка проиндексирована?

Рассмотрим типичный запрос, читающий данные из индексированной таблицы. Пусть, например, у мобильного оператора есть таблица с трафиком, и запрос должен отобрать звонки определённого клиента. На таблице определён индекс, первым (или единственным) ключом которого является идентификатор клиента. Посмотрим, как зависит время чтения данных (t) от количества выбираемых записей (N):



Время полного просмотра таблицы зависит от количества страниц, занимаемых таблицей, и практически не зависит от количества выбираемых строк.

Время, затраченное на выборку строк по индексу, растёт почти линейно с увеличением количества выбираемых строк. Угол наклона графика зависит от нескольких параметров:

- от качества индекса[1]: чем хуже индекс, тем круче кривая, т. к. больше страниц неупорядоченной таблицы придётся читать с диска;

[1] Качество индекса определяется метрикой под названием «фактор кластеризации», речь о которой — в разделе 4.3 «Оптимизация запросов».

- от стоимости операции чтения с диска: при полном просмотре сервер может считывать несколько страниц за одну операцию, а при поиске по идентификатору каждая операция считывает одну страницу.

Таким образом, если клиент — это частное лицо, то выборка по индексу однозначно будет быстрее, а если это транснациональная корпорация со всеми дочерними и зависимыми обществами, по какой-то прихоти объединёнными на одном лицевом счёте, то быстрее будет полный просмотр таблицы. Положение точки пересечения графиков, то есть объём выборки, при котором поиск по индексу становится менее выгодным, чем полный просмотр, зависит от конкретной базы данных, включая такие случайные характеристики, как порядок поступления строк в таблицу.

4.2. Операции реляционной алгебры

В отличие от хранилищ «ключ—значение», где основной операцией является поиск данных, реляционные платформы позволяют выполнять ряд дополнительных операций по обработке данных. К таким операциям относятся сортировка (`ORDER BY`), группировка (`GROUP BY`), соединение (`JOIN`) и операции над множествами — объединение (`UNION` и `UNION ALL`), пересечение (`INTERSECT`), вычитание (`MINUS`).

В современных платформах реализованы наиболее эффективные алгоритмы сортировки, и единственное, на что может повлиять оптимизатор, — объём сортируемых данных, который зависит от того, на каком этапе запроса выполняется сортировка.

Операции над множествами требуют обработки строк группами, в которых значения полей одинаковы. Для этого перед множественными операциями выполняется сортировка набора, но если глобальный порядок не нужен, то оптимизатор может заменить сортировку построением хеш-таблицы.

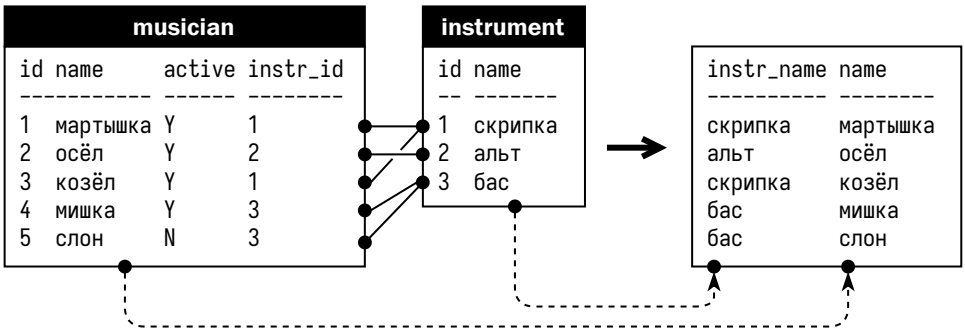
Наибольшее влияние на эффективность выполнения запросов оказывает эффективность соединений, которые мы и будем обсуждать в оставшейся части раздела.

Соединение — операция над двумя таблицами (отношениями), результатом которой является отношение, состоящее из подмножества колонок обеих соединяемых таблиц. Строка в результирующем отношении образуется из строк исходных таблиц, для которых выполняется условие соединения.

Пусть, например, нам требуется вывести имена участников квартета вместе с названиями инструментов, на которых они играют:

```
select
  m.name, i.name
from
  musician m join instrument i on m.instr_id=i.id
```

Условием соединения является совпадение значений внешнего ключа в таблице исполнителей (`instr_id`) и первичного ключа в таблице инструментов (`id`). Соединение, в котором условием является равенство, называется эквисоединением (equijoin) и используется чаще всего. Однако условие соединения может быть любым: частичное совпадение, попадание в диапазон, неравенство и др.



Соединение — важнейшая операция в реляционных платформах. Во многом именно благодаря операции соединения они получили столь широкое распространение и пока не собираются сдавать позиции.

С точки зрения реляционной алгебры соединение — это последовательное применение двух операций: декартова произведения (то есть составления всех возможных пар строк из двух таблиц, в примере выше их было бы $5 \times 3 = 15$) и выборки из него тех, которые удовлетворяют условию соединения. Однако на практике к такому алгоритму может привести только ошибочный запрос, а для выполнения соединений в реальных базах придуманы значительно более эффективные алгоритмы.

В отличие от методов доступа к таблицам и индексам, названия алгоритмов выполнения соединений одинаковы у всех производителей баз данных.

Соединение вложенными циклами (nested loops)

Одна из соединяемых таблиц (не забываем, что на месте таблицы может быть любое отношение — например, представление или результат предыдущей операции) выбирается ведущей (outer[1]), вторая — ведомой (inner). Алгоритм просматривает ведущую таблицу, выбирая из неё строку за строкой, и для каждой выбранной строки просматривает ведомую таблицу, выбирая строки, удовлетворяющие условию соединения. На псевдокоде алгоритм можно записать так:

```
for each i in outer_table
  for each j in inner_table
    if join_condition(i, j) output(i, j)
```

К достоинствам алгоритма можно отнести следующие особенности:

- алгоритм не требует никаких подготовительных действий, а сразу же приступает к работе, максимально быстро выдавая первые строки;
- алгоритм не требует дополнительной памяти;
- условие соединения может быть любым.

Главный недостаток соединения вложенными циклами состоит в многократном просмотре ведомой таблицы. Как следствие время соединения пропорционально произведению количества строк в ведущей и ведомой таблицах.

Вложенные циклы хорошо работают, если количество подходящих записей в ведущей таблице невелико, а просмотр ведомой таблицы выполняется быстро — например, ведомая таблица маленькая.

Особый случай, который чаще всего встречается в транзакционных системах, — когда на ведомой таблице определён индекс по колонке, по которой выполняется соединение. В этом случае полный просмотр, записанный на псевдокоде как `for each`, превращается в поиск по индексу, а условие соединения проверяется до извлечения данных из таблицы:

```
for each i in outer_table
  while ((rowid = search(i.column, inner_table.index)) != NULL) {
    j = read(inner_table, rowid)
    output(i, j)
  }
```

Основным методом выполнения соединений в транзакционных системах является именно алгоритм вложенных циклов с использованием индекса на ведомой таблице.

[1] Тот случай, когда русская терминология точнее выражает суть явления, чем английская.

Соединение слиянием (merge join)

При соединении слиянием обе соединяемые таблицы равноправны. На первом шаге выполняется сортировка данных таблиц, затем на первые строки отсортированных наборов устанавливаются указатели. Если значения ключей соединения в строках под указателями равны, то пара строк добавляется к результату соединения, иначе указатель той таблицы, в которой значение ключа соединения меньше, сдвигается на следующую строку.

Выражение «если значения ключей равны» вместо «проверяется условие соединения» не случайно: в большинстве современных платформ сортировка слиянием позволяет выполнять только эквисоединение. Алгоритм может работать и с условием вида `key1 < key2`, но слияние по условию «больше» или «меньше» реализовано только в Oracle.

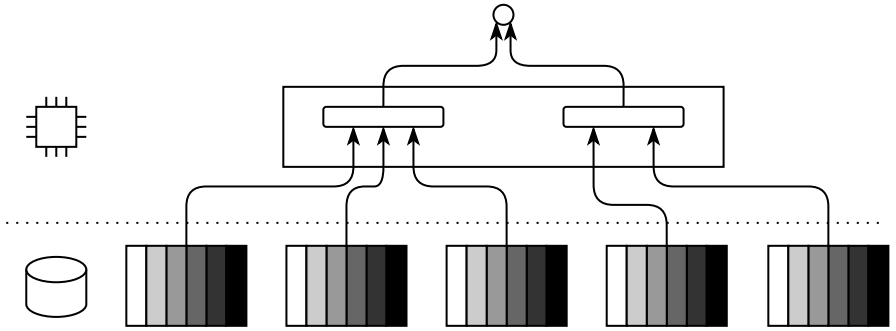
Псевдокод ниже исходит из предпосылки, что каждое значение в колонке, по которой выполняется соединение, встречается в обеих соединяемых таблицах по одному разу. Реализации соединения слиянием в современных платформах работают и в том случае, когда значение встречается в колонке несколько раз.

```
temporary t1[], t2[];  // временные структуры в памяти или на диске
t1 = sort(table1);
t2 = sort(table2);
i = shift(t1);         // получаем очередной элемент
j = shift(t2);
while (i && j) {
  if (key(i)==key(j)) output(i, j);
  if (key(i)<=key(j)) new_i = shift(t1); else new_i = i;
  if (key(i)>=key(j)) j = shift(t2);
  i = new_i;
}
```

Очевидное достоинство алгоритма — однократный просмотр обеих соединяемых таблиц независимо от того, сколько строк выбирается. Столь же очевидный недостаток — необходимость сортировки обеих таблиц.

Если области сортировки в памяти не хватает, сервер сортирует часть соединяемой таблицы, которая помещается в память, записывает отсортированную часть на диск и повторяет операцию до тех пор, пока не будет прочитана вся таблица. Затем сервер одновременно считывает строки из всех сохранённых наборов, выбирает строку с наименьшим значением ключа соединения, выдаёт её как очередную строку в глобально отсортированном потоке, а на её место считывает следующую строку из того же набора. Этот алгоритм называется сортировкой

слиянием (merge sort). Интересно, что последнюю стадию сортировки слиянием можно выполнять одновременно с соединением:



Обычно данные в исходных таблицах никак не отсортированы, поэтому соединение слиянием используется крайне редко. Главным образом этот алгоритм применяется в тех случаях, когда одно из соединяемых отношений уже отсортировано по ключу соединения — например, в результате выполнения предыдущих операций или чтения данных из таблицы полным просмотром покрывающего индекса.

Кстати, и результат соединения также получается отсортированным — это свойство может быть использовано при дальнейшей обработке результата.

Соединение хешированием (hash join)

При соединении хешированием, как и при соединении вложенными циклами, одна таблица является ведущей, другая — ведомой. Алгоритм читает ведущую таблицу и сохраняет её строки в хеш-таблице, где ключ хеширования совпадает с ключом соединения. После того как хеш-таблица построена, алгоритм читает строки ведомой таблицы и ищет соответствующие строки ведущей таблицы, используя хеш-сумму ключа соединения.

На псевдокоде соединение хешированием выглядело бы так:

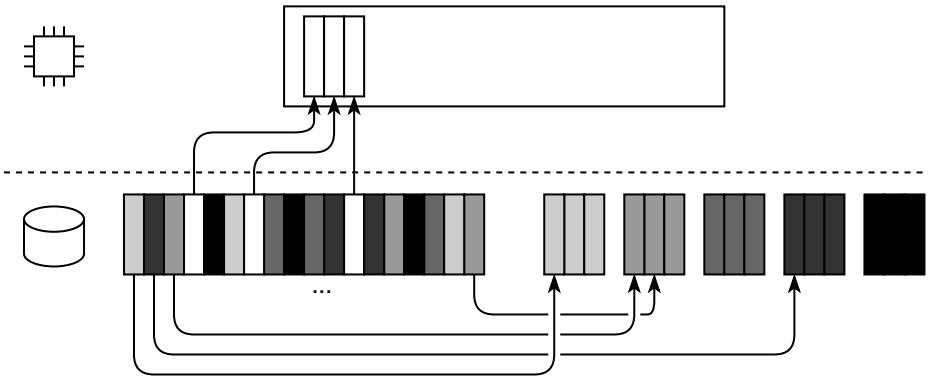
```
temporary h{};                    // хеш-таблица в памяти или на диске
for each i in outer_table
  h{hash(key(i))} = i;
for each j in inner_table
  if (exists(h{hash(key(j))}))
    output (h{key(j)}, j);
```

В примере выше предполагается, что хеш-функция порождает хеш-таблицу нужного размера и не имеет коллизий (или хеш-таблица разрешает эти коллизии прозрачно для пользователя). Промышленные реализации соединения хешированием удовлетворяют этим условиям — механизм разрешения коллизий встроен в используемые структуры.

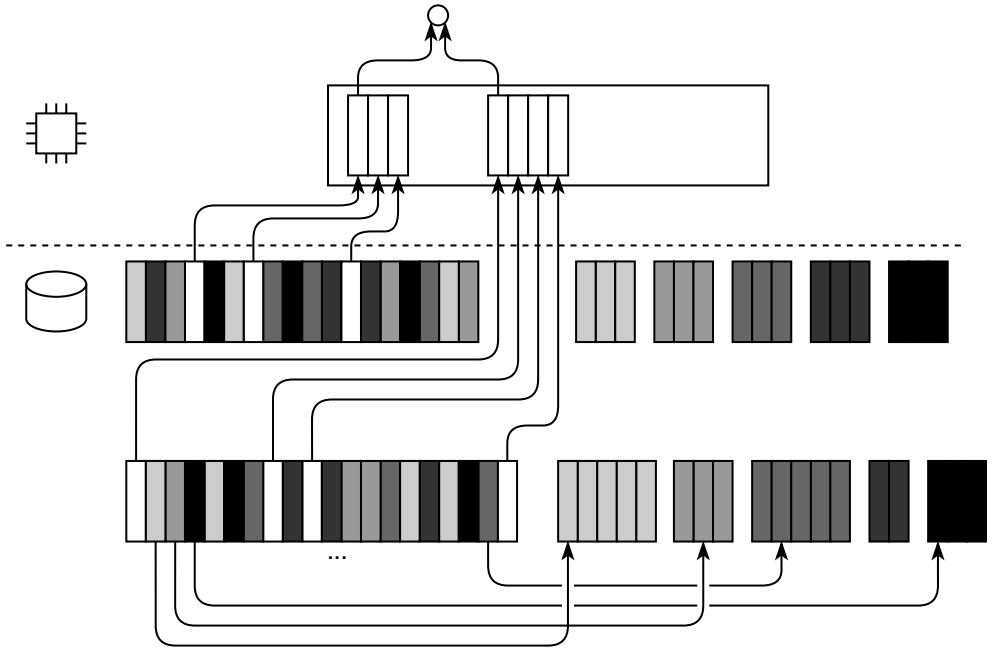
В отличие от предыдущих алгоритмов, хешированием можно выполнить только эквисоединение.

Как и в случае соединения слиянием, обе таблицы, ведущая и ведомая, просматриваются однократно. Так же, как и соединение слиянием, соединение хешированием требует дополнительной памяти, в данном случае — для хранения хеш-таблицы. Правда, объём этой памяти существенно меньше, чем объём области сортировки, т. к. в памяти хранится только ведущая таблица, а подготовка к соединению значительно быстрее, поскольку построение хеш-таблицы, в отличие от сортировки, не требует перемещения элементов. Соединение хешированием работает быстрее, если ведущей таблицей выбрана та, из которой извлекается меньше записей.

Если оперативной памяти для хеш-таблицы не хватает, то приходится использовать временное пространство на диске. При исчерпании области хеширования в памяти сервер разбивает хеш-таблицу на несколько корзин (buckets) и все корзины, кроме первой, записывает на диск:



При чтении ведомой таблицы для каждой прочитанной строки вычисляется хеш-сумма ключа, и если строка попадает в первую корзину, то выполняется соединение, а если не попадает, то данные ведомой таблицы также сохраняются во временное пространство на диске:



По окончании чтения ведомой таблицы и соединения строк, относящихся к первой корзине, первая корзина ведущей таблицы заменяется в памяти на вторую, которая соединяется со второй корзиной ведомой таблицы, и т. д.

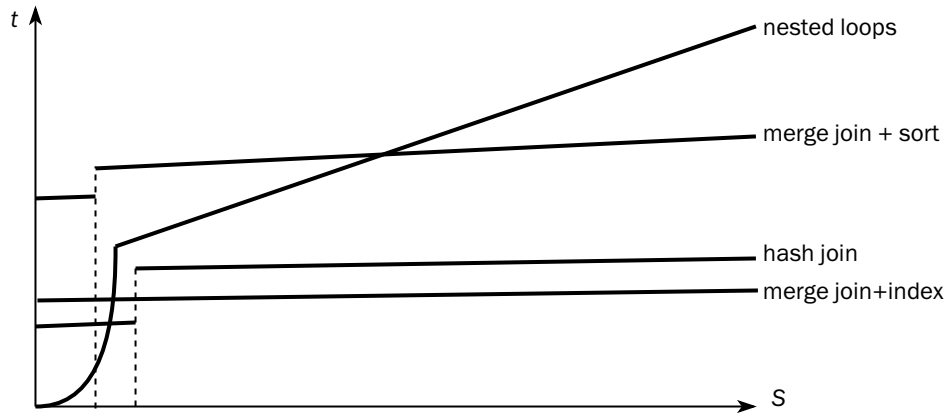
Важным преимуществом соединения хешированием является возможность эффективного выполнения в несколько параллельных потоков как построения хеш-таблицы для ведущей таблицы, так и поиска подходящих элементов в ведомой таблице.

В аналитических системах с большим объёмом оперативной памяти, достаточным для хранения хеш-таблицы по любой из хранимых таблиц, соединение хешированием является основным способом выполнения соединений.

Самый быстрый способ соединения

Как мы убедились выше, не существует «самого быстрого» способа выборки данных из таблицы. С алгоритмами соединения ситуация намного сложнее, но всё же какие-то зависимости проследить можно.

В отличие от графика для выборки, здесь мы оцениваем долю строк, участвующих в соединении, от общего количества строк в таблицах (S):



При соединении вложенными циклами время выполнения операции зависит от времени чтения ведущей таблицы и времени поиска в ведомой. Поначалу, пока доступ к обеим таблицам осуществляется по индексу, время выполнения растёт быстро — пропорционально произведению количества строк, выбранных из ведущей таблицы, на количество строк, выбранных из ведомой таблицы. В какой-то момент прочитать ведущую таблицу целиком становится быстрее, и рост замедляется, превращаясь в линейный, т. к. время чтения ведущей таблицы остаётся постоянным, и растёт только время поиска в ведомой таблице. На графике видно, что при малых объёмах выборки соединение вложенными циклами является самым быстрым, а при больших объёмах — самым медленным.

Соединение хешированием требует полного просмотра ведомой таблицы, поэтому даже если из ведущей таблицы выбрана единственная запись, время соединения будет относительно велико. При выполнении реальных запросов автору не приходилось наблюдать ситуаций, когда для соединения хешированием данные из ведущей таблицы выбирались бы по индексу, поэтому время соединения практически не зависит от количества соединяемых записей. Ступенька на графике соответствует моменту, когда хеш-таблица для ведущей таблицы перестаёт помещаться в память.

Соединение слиянием практически всегда проигрывает по скорости двум другим алгоритмам. Ступенька на графике также соответствует исчерпанию области сортировки в оперативной памяти. Память заканчивается на меньшем количестве строк, чем память для хеш-таблицы.

Соединение может работать быстрее, если оно выполняется несколькими параллельными процессами. Соединение вложенными циклами распараллеливанию не поддаётся, при соединении слиянием сортировка может быть выполнена

параллельно, а при соединении хешированием параллельно могут выполняться обе части алгоритма — и построение хеш-таблицы, и само соединение.

Как было сказано выше, соединения больших таблиц, из которых выбирается значительная часть данных, характерны для аналитических систем. Отсюда следуют и характерные настройки аналитических баз данных — высокий уровень параллелизма и большие области памяти для сортировки и хеширования, даже ценой уменьшения буферного кеша.

4.3. Оптимизация запросов

Запрос, отправленный к серверу базы данных (или, если база данных встраиваемая, в библиотеку работы с данными), до исполнения проходит несколько этапов:

- **Разбор (parse)**. В результате разбора текст запроса превращается в дерево операций, которые надо выполнить, и объектов, к которым надо обратиться. Разбор, в свою очередь, делится на три фазы:
  - лексический разбор — разбиение текста на лексемы: операторы, имена объектов и подобъектов и т. д.;
  - синтаксический разбор — проверка, что набор лексем соответствует грамматике языка, то есть составляет корректное предложение;
  - семантический разбор — проверка возможности выполнения запроса: все ли объекты, перечисленные в запросе, существуют, ко всем ли есть доступ, верны ли типы данных в выражениях и т. д.
- **Преобразование (rewrite)**. На этом этапе сервер избавляется от представлений (views), заменяя их подзапросами, образующими эти представления. Представления хранятся в разобранном виде, и при преобразовании лист дерева, соответствующий представлению, заменяется поддеревом, представляющим запрос из этого представления. Кроме того, на этапе преобразования сервер может, например, добавить условия, реализующие контроль доступа на уровне строк (RLS, row-level security).
- **Оптимизация (optimization)**. На этом этапе дерево логических операций превращается в эквивалентное дерево физических операций (например, логический узел «соединение» превращается в один из физических узлов «соединение вложенными циклами», «соединение хешированием» или «соединение слиянием»).

Языки, используемые для извлечения данных из базы (SQL, языки графовых баз, язык MongoDB, основанный на JSON, и другие), — не процедурные, а декларативные. Это значит, что запрос описывает, что он должен получить, но не описывает, как. Компонент БД, превращающий описание результата в алгоритм для получения этого результата, называется не компилятором (compiler), а **оптимизатором** (optimizer), поскольку одному и тому же запросу может соответствовать несколько алгоритмов (деревьев физических операций), и СУБД должна выбрать оптимальный. Сам алгоритм получения данных называется **планом выполнения запроса** (query plan).

План выполнения запроса

План представляет собой дерево, в листах которого находятся операции чтения данных, в узлах — действия над данными, полученными из дочерних узлов (сортировка, соединение и др.), а корнем является финальная операция получения результата запроса.

Рассмотрим запрос из раздела, посвящённого соединениям:

```
select
  m.name, i.name
from
  musician m join instrument i on m.instr_id=i.id
```

Этот запрос может выполняться по-разному. Вот, например, один из возможных планов его выполнения:

```
SELECT STATEMENT
  MERGE JOIN
    TABLE ACCESS BY INDEX ROWID (INSTRUMENT)
      INDEX FULL SCAN (PK_INSTRUMENT)
    SORT JOIN
      TABLE ACCESS FULL (MUSICIAN)
```

Этот и другие планы, приведённые в данном разделе для иллюстрации, получены в СУБД Oracle. Оптимизаторы других реляционных платформ могли бы построить точно такие же планы, однако названия узлов были бы другими. Точную терминологию конкретных платформ можно найти в фирменной документации.

В листах этого плана мы видим полный просмотр таблицы MUSICIAN и полный просмотр первичного индекса на таблице INSTRUMENT. Большинство родительских узлов имеют по одному потомку — операции в этих узлах обрабатывают данные, полученные от дочернего узла. Так, например, узел SORT JOIN сортирует данные из таблицы MUSICIAN, а узел TABLE ACCESS BY INDEX ROWID извлекает

из неупорядоченной таблицы INSTRUMENT строки по физическим идентификаторам, полученным из индекса. Узлы, выполняющие соединение, имеют по два потомка — так, например, в приведённом плане узел MERGE JOIN выполняет соединение слиянием двух предварительно отсортированных наборов данных. В данном случае порядок дочерних узлов не важен, но для других методов соединения, хеширования и вложенных циклов, ведущий набор данных соответствует первому (верхнему) потомку, а ведомый — второму.

Каждый узел дерева содержит какую-либо операцию над данными:

- поиск данных — просмотр таблицы или индекса;
- фильтр — пропуск на следующий уровень строк, удовлетворяющих определённым условиям;
- сортировка — для вывода данных в порядке, определённом фразой `ORDER BY`, или для последующих операций: соединения слиянием, группировки, множественных операций;
- хеширование — для последующей группировки, если после группировки не требуется сортировка;
- соединение;
- множественные операции — объединение, пересечение, разность множеств; эти узлы имеют по два или более дочерних узла;
- управление процессом обработки запроса — итерация по секциям, координация параллельных процессов и т. д.

Полный список операций и их названий можно найти в документации к конкретным платформам.

План запросов, выполняемых сервером в текущий момент, можно посмотреть в специальных представлениях, отражающих текущее состояние сервера. Однако разработчику, безусловно, хотелось бы увидеть план выполнения до того, как запрос будет запущен на реальных данных. Для разбора запроса и генерации плана служит команда `EXPLAIN`. Возможности команды, её синтаксис и способ работы с возвращаемыми результатами существенно отличаются от платформы к платформе.

План, приведённый выше, не единственный, который может получиться при оптимизации обсуждаемого запроса. Вот ещё один возможный план его выполнения:

```
SELECT STATEMENT
  HASH JOIN
    TABLE ACCESS FULL (INSTRUMENT)
    TABLE ACCESS FULL (MUSICIAN)
```

А вот ещё один план для того же запроса:

```
SELECT STATEMENT
  NESTED LOOPS
    NESTED LOOPS
      TABLE ACCESS FULL (MUSICIAN)
      INDEX UNIQUE SCAN (PK_INSTRUMENT)
    TABLE ACCESS BY INDEX ROWID (INSTRUMENT)
```

Если речь идёт о чтении из одной таблицы (или коллекции), то планы могут отличаться способом поиска — используются ли индексы, и если да, то какие именно. При выборке по нескольким условиям, соединённым предикатом «или», оптимизатор может разбить запрос на несколько запросов, каждый из которых использует свой индекс, или выполнить один запрос, просматривающий таблицу целиком.

Для запроса, в котором участвует несколько таблиц, возможности оптимизатора по выбору плана гораздо шире. Помимо выбора методов поиска данных оптимизатор может выбирать:

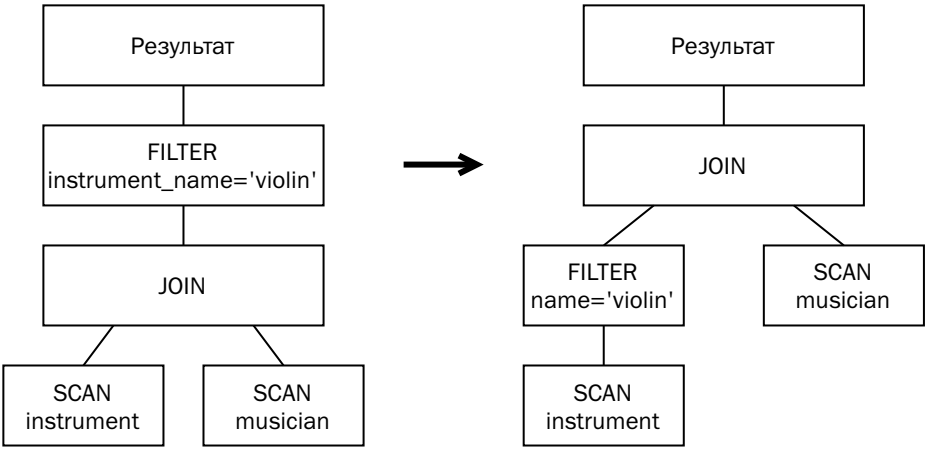
- методы соединения;
- порядок соединения таблиц;
- использование параллельных процессов и алгоритм деления данных между процессами;
- момент применения фильтров — чем раньше будут применены фильтры, тем меньше записей будет передано на верхние уровни дерева.

Количество возможных планов выполнения достаточно сложного запроса может исчисляться сотнями тысяч и даже миллионами, и оптимизатор должен выбрать из них лучший.

Два основных приёма, используемых оптимизаторами, — это применение правил (rule-based optimization) и динамическое программирование (dynamic programming).

Динамическое программирование — метод решения задачи путём её разбиения на связанные подзадачи. Этот метод применяется оптимизаторами для поиска оптимального порядка соединения таблиц.

Оптимизация на основе правил заключается в том, что оптимизатор строит произвольный план запроса, а затем применяет к нему различные преобразующие правила. Большинство правил работают с логическими операциями, то есть их применение не зависит от того, какие именно алгоритмы будут использованы. Примером такого правила может служить «проталкивание предиката» (predicate push-down):



Оптимизатор может опираться только на правила и алгоритмы либо же учитывать информацию о конкретных данных, хранящихся в базе. Первый подход называется эвристической оптимизацией (heuristic optimization), а второй — стоимостной оптимизацией (cost-based optimization).

Эвристическая оптимизация

Эвристическая оптимизация использует набор эмпирических фактов (эвристик) вида «операция A, как правило, выполняется быстрее, чем операция B». Так, например, у эвристического оптимизатора[1] Oracle есть список из 15 проранжированных методов доступа к данным. Оптимизатор выбирает план так, чтобы его выполнение содержало методы доступа с наименьшими рангами.

- RBO Path 1: Single Row by Rowid;
- RBO Path 4: Single Row by Unique or Primary Key;
- RBO Path 8: Composite Index;
- RBO Path 9: Single-Column Indexes;
- RBO Path 10: Bounded Range Search on Indexed Columns;
- RBO Path 11: Unbounded Range Search on Indexed Columns;
- RBO Path 12: Sort Merge Join;
- RBO Path 13: MAX or MIN of Indexed Column;
- RBO Path 14: ORDER BY on Indexed Column;
- RBO Path 15: Full Table Scan.

[1] Oracle использует термин «rule-based optimizer», «оптимизатор, основанный на правилах».

Выше приведён список методов с сокращениями. Соединения хешированием в этом списке нет, поскольку реализация этого алгоритма появилась в Oracle 7 одновременно со стоимостным оптимизатором. Из двух оставшихся планов, приведённых выше, план, который использует соединение слиянием, хуже с точки зрения эвристического оптимизатора: оба плана читают таблицу MUSICIAN целиком, но таблицу INSTRUMENT для соединения слиянием приходится сортировать по индексу (ранг 14), а для вложенных циклов — выбирать записи при помощи индекса по одной (ранг 4).

Эвристический оптимизатор прост, но зачастую неэффективен: чем шире использование баз данных, тем чаще встречаются такие наборы данных, на которых эвристические правила оказываются неверными.

Эвристический оптимизатор использовался в ранних релизах Interbase. СУБД Oracle использовала эвристический оптимизатор в релизах со второго по шестой (1979–1988) и в целях совместимости поддерживала в релизах с седьмого по десятый (1992–2005). Сегодня чисто эвристический оптимизатор для совместимости поддерживается только в СУБД Db2 при выборе классов оптимизации 0 или 1.

Все современные реляционные платформы используют стоимостные оптимизаторы, хотя в них могут использоваться некоторые эвристики для сужения пространства перебора.

Оптимизация, основанная на стоимости

Задачей стоимостного оптимизатора является поиск плана с наименьшей (сюрприз!) стоимостью. **Стоимость** — это количественное выражение оценки затрат сервера на выполнение запроса в некоторых «условных единицах». Не всегда, но в большинстве случаев справедливо утверждение, что чем меньше стоимость плана, тем быстрее выполнится запрос, для которого построен план.

Сильно упрощая, можно считать, что стоимость узла равна произведению количества выполненных операций на стоимость одной операции, а стоимость выполнения запроса равна сумме стоимостей всех узлов.

Рассмотрим простейший запрос — поиск данных в таблице по значению колонки. На языке SQL этот запрос записывается так:

```
select *
from musician
where instr_id=1
```

Если колонка проиндексирована, то оптимизатор может выбрать один из двух планов. Первый план предполагает полное чтение таблицы:

```
SELECT STATEMENT (ROWS=2, COST=3)
  TABLE ACCESS FULL (MUSICIAN) (ROWS=2, COST=3)
```

Второй план предусматривает поиск значения по индексу:

```
SELECT STATEMENT (ROWS=2, COST=2)
  TABLE ACCESS BY INDEX ROWID BATCHED (MUSICIAN) (ROWS=2, COST=2)
    INDEX RANGE SCAN (X_MUSICIAN_INSTRUMENT) (ROWS=2, COST=1)
```

На каждом шаге оптимизатор вычисляет стоимость выполнения шага и количество строк, которые вернёт этот шаг. В данном запросе мы видим сканирование индекса по диапазону значений и частичный просмотр таблицы. Откуда же оптимизатор может знать, сколько строк будет выбрано на каждом шаге?

Для вычисления количества строк оптимизатор использует **статистику** (statistics) — интегральную информацию о данных в таблицах и индексах. Для сбора этой информации СУБД периодически запускает специальные запросы, выборочно читающие данные и вычисляющие статистические показатели. В ранних системах, использовавших стоимостной оптимизатор, обязанность сбора статистики лежала на администраторе базы данных, который мог запускать сбор статистики вручную или по расписанию. Однако статистика настолько важна для работы современных платформ, что сервер в некоторых случаях может самостоятельно принимать решение о сборе статистики — например, после операций, меняющих существенное количество строк в таблице, или даже непосредственно в момент разбора запроса, если статистики нет или она устарела.

Статистическая информация сохраняется в системных таблицах и при помощи специальных процедур может быть перенесена между базами. Так, например, статистика, собранная на промышленной БД, может быть скопирована в тестовую базу, чтобы увидеть, как запросы будут планироваться при реальной эксплуатации системы. Профиль данных (количество записей и распределение значений колонок) в промышленной и тестовой базах могут существенно отличаться.

Список собираемых метрик может отличаться для разных платформ, но основные метрики для таблицы перечислены ниже:

- **Количество строк**. Позволяет оценить, сколько строк будет прочитано из таблицы при полном просмотре. Если запрос выбирает данные по какому-либо условию, метрики столбца позволяют оценить долю выбранных строк от общего количества.
- **Количество занятых страниц**. Системные таблицы, описывающие распределение дискового пространства, содержат информацию о том, сколько страниц выделено таблице, но не все выделенные страницы могут быть заняты данными.

- **Средний размер строки**. При построении структур в памяти (для хеширования или сортировки) позволяет оценить необходимый объём буфера. Данные некоторых типов — строки (за исключением строк постоянной длины), двоичные данные, в некоторых платформах и числа — имеют переменную длину, и оценка, использующая только описание колонок в словаре, будет завышенной.

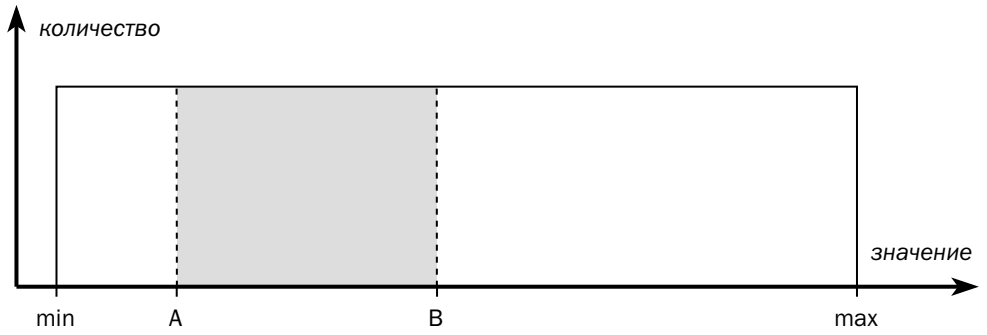
Кроме общей статистики по таблице оптимизатор использует статистику по каждой колонке:

- **Количество различных значений**. Помогает оценить количество строк, которое будет выбрано по условию, наложенному на колонку. Например, если в таблице миллион строк, а в колонке сто тысяч различных значений, то фильтр по единственному значению колонки оставит примерно десять строк.
- **Средний размер данных в колонке**. Требуется для более точной оценки размеров буферов в памяти, когда выбирается не вся строка целиком, а только отдельные колонки.
- **Количество неопределённых значений (null)**. Используется для корректировки оценок количества выбираемых строк и размера буферов.
- **Минимальное и максимальное значения**. Также используются для корректировки оценки количества строк. Если хотя бы одна из границ, указанных в условии запроса, выходит за диапазон значений в таблице, количество выбранных строк будет меньше.

Долгое использование стоимостных оптимизаторов показало, что частота разных значений в данных отнюдь не одинакова, и из-за этого оценки, базирующиеся только на интегральных показателях, становятся неточными и не позволяют выбрать оптимальный план.

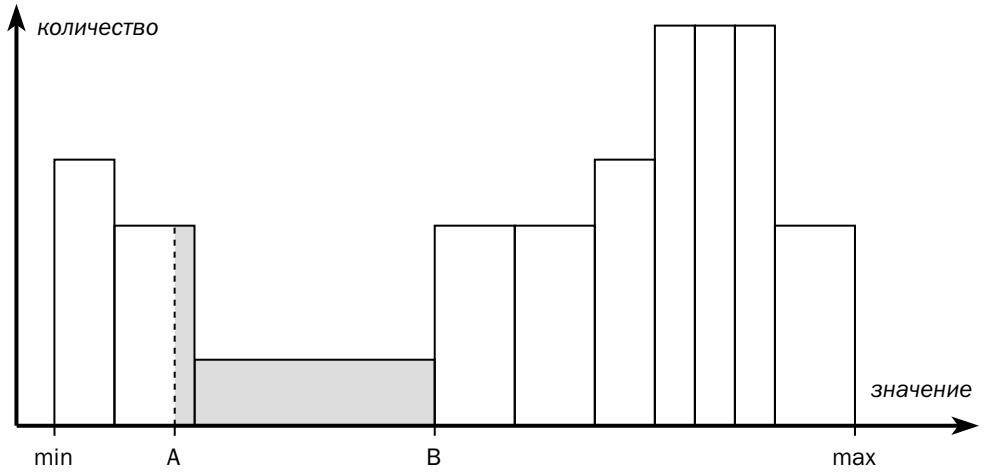
Предположим, что значения в колонке распределены равномерно. Построим прямоугольник, горизонталь которого соответствует диапазону значений (от min до max), а площадь равна общему количеству строк в таблице, то есть высота прямоугольника соответствует количеству строк, в которых встречается любое значение. На иллюстрации видно, что если выбрать строки, в которых значение колонки находится в диапазоне от A до B, то мы ожидаем получить примерно треть всех строк (32,5 %).

В действительности данные распределены так, что выборка по условию «значение колонки лежит в диапазоне между A и B» должна вернуть примерно десятую часть таблицы, то есть втрое меньше, чем предполагает оптимизатор.



Чтобы помочь оптимизатору учитывать профиль данных, современные платформы собирают расширенную статистику:

- **Часто встречающиеся значения (frequent-value)**. Выбирается несколько десятков или даже сотен наиболее часто встречающихся значений, и сохраняются как сами значения, так и количество соответствующих им строк. Такая метрика позволяет намного точнее оценить количество записей в выборке с фильтром по колонке.
- **Гистограмма (histogram), или квантили (quantiles)** — статистика частот значений, разбитая по диапазонам. Этот способ — лучший по соотношению объёма занимаемого пространства к точности оценки количества записей в выборке.



При построении гистограмм помимо минимального и максимального значений система выбирает ещё несколько точек так, чтобы количество строк, значения колонок в которых лежат в диапазоне между соседними точками, было примерно одинаковым. В терминах графиков это означает, что площади всех прямоугольников (столбцов гистограммы) равны, а их суммарная площадь равна площади единственного прямоугольника на верхнем графике с двумя точками min и max.

Имея гистограмму, оптимизатор может точнее оценить количество выбираемых строк. На рисунке выше площадь заштрихованного прямоугольника составляет 12,5 % от общей площади гистограммы, что намного точнее описывает текущий профиль данных в таблице, чем только минимальное и максимальное значения.

Современные стоимостные оптимизаторы могут использовать одновременно и гистограммы, и списки часто встречающихся значений.

Нередко случается, что значения в нескольких колонках коррелируют. Пусть, например, торговые точки отбираются по городу и области. Отбор по области даёт нам 10 % строк, а отбор по городу — 2 %, из чего оптимизатор может сделать вывод, что выборка по двум критериям даст всего 0,2 %. Человеку очевидно, что на самом деле ограничение по области ничего не добавляет к ограничению по городу, и запрос выберет 2 % записей. Чтобы оптимизатор тоже об этом узнал, требуется собрать **статистику по набору колонок**.

Платформа сама по себе не рассчитывает корреляцию данных в колонках. Понять, какие колонки связаны между собой, и собрать статистику по связанным наборам — задача администратора.

Вообще, администратор приложения играет важную роль при сборе необходимой статистики для обеспечения корректной работы оптимизатора и, соответственно, повышения производительности базы данных. Помимо выявления связанных наборов данных, администратор приложения должен принимать решения и по другим вопросам:

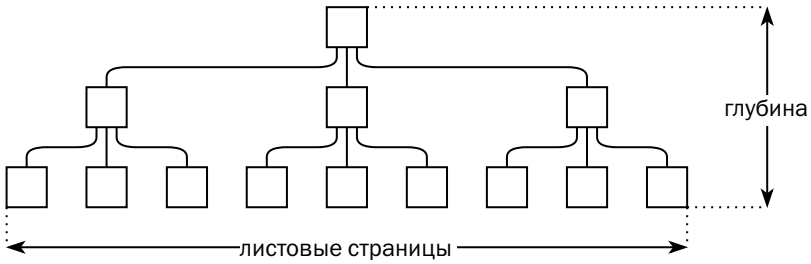
- По каким колонкам собирать частые значения и гистограммы. Сбор такой статистики по колонкам, не участвующим в условиях отбора, отнимает ресурсы системы, но не повышает скорость выполнения запросов. Строить гистограммы рекомендуется как минимум для всех индексированных колонок (за исключением первичного ключа).
- Какую часть данных нужно оценить при сборе статистики. Современные платформы имеют возможность считывать не все данные, а какое-то

случайное подмножество, и важно найти компромисс между скоростью сбора статистики и её точностью.

- Сколько требуется наиболее частых значений и сколько столбцов[1] в гистограмме по каждой колонке. Большое количество столбцов замедляет работу оптимизатора, но с какого-то момента уже не повышает точность оценок.

Помимо статистики по таблицам, базы данных собирают и статистику по индексам, чтобы оптимизатор мог оценить эффективность использования этих индексов:

- **Глубина**, то есть количество уровней в B-дереве. Позволяет оптимизатору оценить количество страниц, которые потребуется просмотреть при поиске значения.
- **Количество листовых страниц**, то есть количество страниц на последнем уровне. Позволяет оценить количество прочитанных страниц при полном просмотре и (в сочетании с общим количеством страниц, занятых индексом) количество напрасно прочитанных страниц при быстром просмотре.

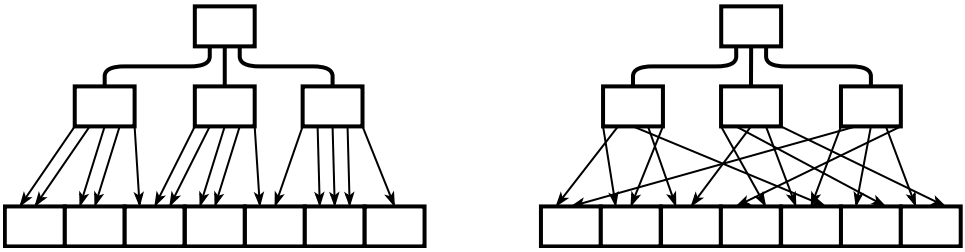


- **Количество ключей** и **среднее количество ключей на странице** позволяют оценить количество просмотренных страниц при поиске по ключу или диапазону ключей.
- **Фактор кластеризации**[2] показывает, насколько близко друг к другу находятся в таблице строки с одинаковым значением индексированного поля, и позволяет оценить количество страниц таблицы, просматриваемых при поиске по этому полю.

---

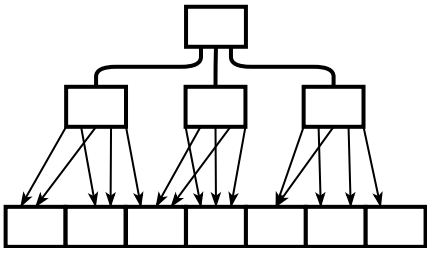
[1] В базе данных гистограмма представлена набором точек, обозначающих границы столбцов. Площадь каждого столбца равна частному от деления общего количества записей на количество столбцов.

[2] Clustering factor — термин Oracle. Другие платформы могут иметь аналогичные по смыслу статистики, но с иными названиями и иными алгоритмами расчёта.



На рисунке выше показаны индексы с низким фактором кластеризации (идеальный фактор кластеризации равен количеству страниц в таблице) и с высоким фактором кластеризации (наихудший фактор кластеризации равен количеству строк в таблице). Единственный способ изменить фактор кластеризации индекса — переупорядочивание данных в таблице, то есть в системах, обрабатывающих транзакции, влиять на этот параметр практически невозможно. Если на таблице определено несколько индексов, то, перестраивая таблицу с целью уменьшения фактора кластеризации одного индекса, мы увеличим фактор кластеризации другого.

Простейший способ расчёта фактора кластеризации состоит в том, чтобы двигаться по листовым страницам слева направо и добавлять к фактору кластеризации единицу каждый раз, когда следующий элемент индекса указывает на иную страницу, чем текущий. Однако при использовании такого алгоритма возможны ошибки:



Индекс на рисунке выше хорошо кластеризован, но его фактор кластеризации будет высоким, то есть плохим. Для корректного расчёта в алгоритм вводится параметр, характеризующий количество кешированных страниц индекса. При расчёте фактора кластеризации единица добавляется только в том случае, если блока, на который указывает очередной элемент, нет в кеше.

Учёт фактора кластеризации позволяет точнее оценить стоимость сканирования индекса по диапазону значений. На расчёт стоимости извлечения уникального значения фактор кластеризации не влияет.

Но вернёмся к исходному запросу. Для того чтобы сравнить эффективность разных планов выполнения этого запроса, оптимизатору требуется вычислить их стоимость.

Рассмотрим план, выполняющий полный просмотр таблицы. Зная количество страниц, которое занимает таблица, оптимизатор может рассчитать, сколько раз нужно обратиться к диску, чтобы считать все требуемые страницы. Зная, что сервер будет просматривать все страницы подряд, оптимизатор будет оценивать многостраничные чтения, то есть операции, считывающие за раз несколько страниц данных.

Стоимость плана, использующего индекс, вычисляется сложнее. Основываясь на гистограммах для поля `INSTR_ID`, оптимизатор оценит количество извлекаемых записей. Затем, взяв глубину индекса и составив пропорцию из количества выбранных записей, общего количества записей и количества листовых блоков, рассчитает количество прочитанных индексных страниц. И наконец, приняв во внимание фактор кластеризации индекса, приблизительно вычислит количество страниц, которые надо будет просмотреть для извлечения искомых строк. Поскольку просматриваемые страницы разбросаны хаотично, каждая операция будет считывать одну страницу, а не несколько, как при полном просмотре.

Осталось добавить, что стоимость остальных узлов плана (соединений, сортировок, множественных операций) оценивается так же, как и стоимость поиска: сначала оценивается количество и размер строк, поступающих из дочерних узлов, а затем исходя из этих параметров вычисляется стоимость необходимых операций. Стоимость может расти нелинейно: так, например, стоимость сортировки зависит от количества записей в лучшем случае как $n \times \log n$, а если область сортировки или хеш-таблица не помещается в соответствующую область памяти, то алгоритм начинает использовать временное пространство на диске, выполняя другие операции с совсем другой стоимостью.

В результате сравнения планов мы получили набор операций, которые будут выполнены при реализации плана, и оценку количества этих операций. Для получения стоимости запроса осталось умножить количество операций на «цену» операции. Но где взять эту цену?

На этот вопрос разные платформы отвечают по-разному:

- У Db2 и Microsoft SQL Server есть некие внутренние константы, характеризующие стоимость каждой операции. Возможно, константы могут меняться в зависимости от конфигурации сервера, но администратор не может управлять этими изменениями.

- У PostgreSQL константы стоимости для планировщика (planner cost constants) вынесены в параметры экземпляра — при необходимости их можно изменить.
- В MySQL можно создать специальные таблицы (optimizer cost model), куда записывается стоимость операций. Если таких таблиц нет, или в таблицах нет нужного значения, оптимизатор использует внутренние константы.
- В Oracle за относительную скорость (цену) выполнения тех или иных операций отвечает системная статистика (system statistics), которая может быть собрана администратором в ручном или автоматическом режиме.

Важно понимать, что абсолютная цена операций не важна — важно соотношение цен разных операций. Так, если в PostgreSQL увеличить значение всех констант вдвое, это никак не повлияет на работу оптимизатора. Константы рассчитаны на некоторую «эталонную» конфигурацию сервера, и править их имеет смысл, если конфигурация значительно отличается от эталонной — например, в сервере установлены медленные дешёвые диски и большое количество процессорных ядер, или наоборот — NVMe-flash в качестве дискового накопителя, но медленный процессор и мало памяти.

После того как план построен, начинается выполнение запроса. Стоимость плана даёт примерное представление о времени выполнения, но на время влияет и текущее состояние базы данных и экземпляра. Пусть, например, в плане выполняется поиск по индексу, для чего экземпляр обращается к страницам индекса и таблицы. Сколько раз при этом придётся читать данные с диска? Ответ на этот вопрос даст статистика использования кеша (cache hit ratio), показывающая, насколько часто искомая страница оказывается в кеше. Очевидно, что эта статистика зависит от объёма кеша, от характера нагрузки и от того, о какой именно странице идёт речь — так, например, корневая страница индекса будет в кеше практически со стопроцентной вероятностью.

Итак, есть четыре параметра, влияющих на скорость выполнения запроса:

1) **набор выполняемых операций**, который определяется выбранными методами поиска или соединения;
2) **количество обработанных записей**, которое зависит от профиля данных и на этапе оптимизации может быть оценено при помощи статистики;
3) **цена каждой операции**, которая определяется способом, специфическим для платформы;
4) **настройки экземпляра и профиль нагрузки**, уникальные для каждой базы данных.

Если первые три параметра можно оценить на этапе оптимизации, то влияние четвёртого проявляется только непосредственно в момент выполнения. Оптимизатор делает некоторые предположения — насколько часто запрос сможет избежать физического чтения, обнаружив требуемую страницу в кеше, или насколько вероятно, что область сортировки поместится в оперативную память. Однако фактическое использование памяти зависит от активности конкурентных сеансов и от того, к каким данным они обращаются.

Важно понимать, что планы запросов, оптимальные на тестовых базах, могут не быть оптимальными на базе под реальной нагрузкой. Чтобы оптимизатор тестовой базы принимал такие же решения, как и оптимизатор промышленной базы, можно скопировать с промышленной базы статистику по данным и по стоимости выполнения операций, но скопировать нагрузку не получится. Некоторые возможности эмуляции промышленной среды даёт Db2, позволяя указать при помощи параметров характеристики «настоящей» базы — мощность и количество процессоров, объём буферов, количество активных сеансов и т. п.

Кеширование запросов и планов

Разбор и оптимизация запроса — операция относительно дорогая. В аналитических системах, когда запрос обрабатывает десятки миллионов строк и выполняется много секунд или даже минут, стоимостью разбора можно пренебречь. Однако в случае коротких запросов в системах обработки транзакций стоимость разбора многократно превосходит стоимость исполнения запроса. Чтобы уменьшить накладные расходы на разбор запросов, современные платформы кешируют как текст выполняемых запросов, так и планы выполнения.

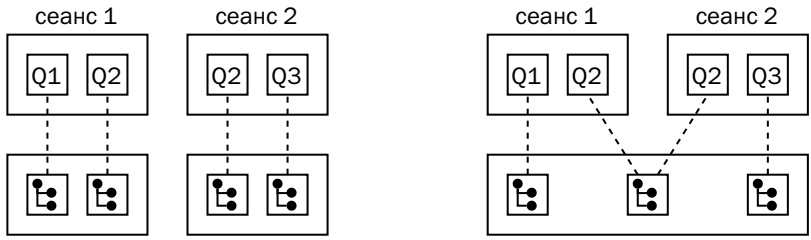
В PostgreSQL или MySQL каждый сеанс поддерживает свой собственный кеш запросов и планов[1]. Но в Oracle, Db2[2], Microsoft SQL Server кеш запросов и планов находится в общей области памяти, доступной всем сеансам.

Как водится, каждый производитель придумал для обозначения этой области свой термин: library cache (Oracle), plan cache (Microsoft SQL Server), package cache (Db2).

Получив запрос, сервер ищет такой же запрос в кеше, и если находит, то использует готовое дерево разбора и, если возможно, готовый план. Критерий

[1] Коммерческая СУБД Pangolin, разрабатываемая компанией «Сбербанк-Технологии» на основе PostgreSQL, может поддерживать общий кеш запросов.

[2] В зависимости от настроек Db2 может использовать и локальный кеш для каждого сеанса.



PostgreSQL, MySQL

Oracle, DB2, MS SQL

сходства запросов — посимвольное совпадение, включая пробелы, регистр букв и комментарии. Некоторые платформы (Db2, Microsoft SQL Server, ранние релизы Oracle) позволяют смягчить эти критерии (например, игнорировать комментарии) при помощи настроек сервера.

Найти совпадающий запрос в кеше — быстрее, чем полностью разобрать запрос заново, но медленнее, чем обратиться к подготовленному запросу, зная его дескриптор. В современных приложениях, ориентированных на обработку транзакций, динамическое создание SQL-запросов считается дурным тоном — вместо этого для достижения высокой производительности рекомендуется использовать статические подготовленные запросы. Подробнее о подготовке запросов поговорим в следующем подразделе.

Очевидно, что сервер стремится сохранять запросы и их планы в кеше как можно дольше, в идеале — всё время работы. Однако если место в кеше заканчивается, сервер вынужден вытеснять какие-то запросы и планы из кеша. Есть также ряд событий, из-за которых план запроса становится недействительным даже в том случае, если он хранится в кеше и активно используется. К таким событиям относятся:

- изменение структуры объектов, участвующих в запросе, — например, добавление колонки к таблице;
- удаление объектов, используемых в плане, — таблиц или индексов; при удалении индекса план будет перестроен заново, а удаление таблицы делает недействительным не только план, но и дерево разбора, и очередное выполнение запроса приведёт к ошибке разбора;
- появление новых индексов, которые могут быть использованы при выполнении запроса;
- изменение параметров хранения объектов, участвующих в запросе, — например, перенос таблицы в другое табличное пространство;

- появление обновлённой статистики по объектам, участвующим в запросе;
- ручной сброс кеша администратором базы данных.

Оптимизация параметризованных запросов

Предположим, что мы пишем приложение, позволяющее пользователю выбрать музыкантов, играющих на произвольном инструменте. Есть два способа взаимодействия приложения с базой данных.

Первый заключается в том, что приложение каждый раз динамически конструирует новый запрос. Запрос для поиска скрипачей будет выглядеть так:

```
select * from musician where instr_id=1
```

Для поиска альтистов запрос изменится:

```
select * from musician where instr_id=2
```

Преимущество такого подхода в том, что оптимизатор в момент разбора запроса знает всё о данных, которые этот запрос собирается извлечь. Недостатки гораздо серьёзнее: во-первых, при каждом выполнении происходит полный разбор запроса, а во-вторых, достаточно скоро эти запросы начнут вытеснять из кеша все остальные запросы, заставляя сервер разбирать их также заново. В платформах с общим кешем запросов динамический SQL негативно влияет на производительность всех приложений, работающих с текущим экземпляром. Если же таким образом формируются запросы для выборки по первичному ключу, то сформированные запросы вообще лишены преимуществ, поскольку идентификатор уникален, что известно даже без статистики.

В подавляющем большинстве случаев выгоднее один раз разобрать запрос, а потом много раз его выполнить. Текст запроса при этом будет выглядеть так:

```
select * from musician where instr_id=:id
```

Здесь литерал, начинающийся с двоеточия (`:id`), обозначает формальный параметр. Приложение, использующее параметризованный запрос, должно выполнить четыре шага:

1. **Подготовка (prepare)**. Сервер разбирает запрос, сохраняет дерево разбора и иногда — план выполнения. В результате выполнения этого шага клиент получает дескриптор (handle) подготовленного запроса.
2. **Привязка локальных переменных к формальным параметрам (bind)**. На этом шаге приложение указывает драйверу БД, откуда брать значения параметров.

3. **Присваивание** привязанным переменным фактических значений параметров для конкретного вызова.
4. **Исполнение (execute)** — непосредственно выполнение запроса с фактическими параметрами.

Первые два шага выполняются один раз, а третий и четвёртый — многократно.

Поскольку стадии разбора (лексического, синтаксического, семантического) и переписывания работают совершенно одинаково вне зависимости от значения параметров, сервер всегда может сохранить дерево разбора — структурированное представление запроса, содержащее логические операции. Кроме того, сервер может сделать некоторые предположения относительно значения параметров, основываясь на статистике по колонкам, с которыми сравниваются параметры, и построить универсальный план выполнения запроса.

Дальнейшая работа с универсальным планом зависит от платформы.

Так, например, Microsoft SQL Server для подготовленного запроса всегда использует универсальный план[1].

Db2 позволяет для каждого запроса выбрать режим исполнения — либо будет использоваться универсальный план, либо при каждом вызове запрос будет оптимизирован заново с учётом значений параметров.

PostgreSQL помечает, что общий план (generic plan) ещё ни разу не использовался и первые несколько раз оптимизирует запрос заново, строя частные планы (custom plans) и оценивая стоимость их исполнения. Если частные планы оказываются в среднем дешевле, чем общий, то сервер продолжает и дальше для каждого набора параметров оптимизировать запрос заново. Если же общий план достаточно хорош, то сервер перестаёт строить частные планы и переключается на использование общего. При необходимости программист может запретить использование общего плана и заставить оптимизатор каждый раз строить частный план (или наоборот, заставить всегда использовать общий план), но, в отличие от Db2, такое указание распространяется на все запросы, выполняемые в сеансе или в функции, а не на конкретный запрос.

В отличие от всех рассмотренных выше подходов, в Oracle адаптивные планы (adaptive query plans) используют не только статистику, сохранённую в базе, но и реальную статистику, собранную во время выполнения запроса.

---

[1] Если Microsoft SQL Server работает в режиме хранилища данных, выполняющего уникальные запросы (optimize for ad hoc workloads), то в кеш запросов не сохраняется ничего — только «заглушка», т. к. вероятность повторного выполнения запроса считается крайне низкой. Кроме того, есть ещё ряд условий, при которых кешируется дерево запроса, но не кешируется план.

Рассмотрим, например, план, полученный для одного из запросов выше:

```
SELECT STATEMENT
  NESTED LOOPS
    NESTED LOOPS
      TABLE ACCESS FULL (MUSICIAN)
      INDEX UNIQUE SCAN (PK_INSTRUMENT)
    TABLE ACCESS BY INDEX ROWID (INSTRUMENT)
```

Oracle начинает с использования общего плана, в котором для каждого узла хранится ожидаемое количество обработанных строк. Если во время выполнения реально полученное количество строк сильно отличается от ожидаемого (например, превосходит его), оптимизатор Oracle перестраивает план запроса, оставляя как старую, так и новую ветви:

```
SELECT STATEMENT
- NESTED LOOPS
-   NESTED LOOPS
-     TABLE ACCESS FULL (MUSICIAN)
-     INDEX UNIQUE SCAN (PK_INSTRUMENT)
-   TABLE ACCESS BY INDEX ROWID (INSTRUMENT)
  HASH JOIN
    TABLE ACCESS FULL (INSTRUMENT)
    TABLE ACCESS FULL (MUSICIAN)
```

Затем сервер продолжает выполнять запрос, используя активную ветвь плана, не отмеченную на иллюстрации минусами. При следующем запуске запроса сервер начинает исполнение со старой ветви и при превышении количества обработанных строк может вновь переключиться на новую. Если в большинстве случаев новая ветвь оказывается эффективнее, то в дальнейшем сервер может выбрать её как основную.

Для повышения производительности старых (legacy) приложений, динамически генерирующих запросы, в платформах с общим кешем планов предусмотрен режим, когда сервер преобразует все литералы, найденные в запросе, в параметры[1].

Предположим, нам требуется извлечь из базы данных список виолончелистов, но не всех, а только тех, которые продолжают выступать в Квартете:

```
select
  m.name
from
  musician m join instrument i on m.instr_id=i.id
where
  m.active='Y' and i.name='альт'
```

[1] `CURSOR_SHARING = FORCE` (Oracle), `PARAMETERIZATION = FORCED` (Microsoft SQL Server), `stmt_conc = LITERALS` (Db2).

В режиме принудительной параметризации при разборе запроса оба литерала будут заменены на параметры, и запрос примет следующий вид:

```
select
  m.name
from
  musician m join instrument i on m.instr_id=i.id
where
  m.active=:1 and i.name=:2
```

Такая параметризация может служить временным решением, но при разработке новых приложений рассчитывать на неё не стоит. В рассматриваемом запросе значение первого параметра сильно влияет на план, поскольку рано или поздно количество «бывших» музыкантов будет существенно больше, чем количество реальных участников Квартета. Для фильтров по колонкам, значения которых распределены столь неравномерно, эффективнее было бы использовать литерал. Названия же инструментов распределены по строкам более или менее равномерно, и второй параметр правильнее сделать именно параметром. При принудительной параметризации запросов оптимизатор не делает различий и заменяет параметрами все литералы.

В заключение стоит отметить, что параметризация запросов делает приложения не только более производительными, но и более безопасными. Если запрос параметризован, то можно быть уверенным в том, что структура запроса не изменится при подстановке в качестве параметра любого значения. Если же вместо привязки параметров динамически генерировать новые запросы, подставляя в текст литералы, то в лучшем случае можно получить ошибку, а в худшем — выполнение кода, записанного в подставленном значении. Классический пример такой ошибки — имена, содержащие апостроф. Если, например, вместо обычной Мартышки попытаться найти её французскую коллегу d'Ouistiti, то строка закончится закрывающим апострофом после буквы d, а оставшиеся символы приведут к ошибке разбора:

```
select
  m.name, i.name
from
  musician m join instrument i on m.instr_id=i.id
where
  m.name='d'Ouistiti'
```

Безусловно, существуют ситуации, где литералы полезны, но в этих ситуациях следует очень внимательно относиться к экранированию спецсимволов.

Ручное управление планами

Несмотря на то что современные оптимизаторы прошли огромный путь и научились учитывать почти все факторы, влияющие на производительность базы данных, периодически даже они могут ошибаться. Для исправления ошибок оптимизаторов у разработчика прикладного кода или у администратора БД должны быть инструменты, позволяющие вручную управлять планами запросов.

Степень допустимости ручного влияния на оптимизатор отличается от платформы к платформе, но в основе всех подобных инструментов лежит одна из трёх идей:

- ограничение набора допустимых алгоритмов;
- подсказки оптимизатору, предписывающие использование определённого алгоритма;
- явная запись нужного плана в кеш.

Помимо общих настроек оптимизатора — таких, например, как максимальное количество рассматриваемых планов, — некоторые платформы позволяют явно включить или выключить использование определённых алгоритмов. Например, если в PostgreSQL требуется при исполнении нашего запроса исключить соединение хешированием, можно выполнить следующий код:

```
set ENABLE_HASHJOIN=FALSE;
select
  m.name, i.name
from
  musician m join instrument i on m.instr_id=i.id;
```

У механизма параметров крайне низкая избирательность: параметр действует не только на все соединения, выполняемые в запросе, но и на все запросы, выполняемые в сеансе или в транзакции. Поэтому подобные параметры могут быть использованы только как кратковременная мера по исправлению проблем конкретного сеанса.

Гораздо более избирательный механизм — подсказки (hints). Подсказка — часть запроса, предписывающая оптимизатору использовать (или наоборот — не использовать) конкретный алгоритм при выполнении данного запроса. Впервые подсказки появились в Oracle 7 как временное решение при переходе на тогда ещё несовершенный стоимостной оптимизатор.

Рассматриваемый запрос с подсказкой в Oracle выглядит так:

```
select --+ ordered use_nl(i)
  m.name, i.name
from
  musician m join instrument i on m.instr_id=i.id
```

Синтаксически подсказка в Oracle представляет собой комментарий специального вида, начинающийся со знака «+». Подсказка может управлять порядком соединения, явно указывать алгоритмы соединения, предписывать полный просмотр таблицы или доступ с использованием индекса, управлять параллельным исполнением запроса... В частности, в приведённом примере подсказка ORDERED сообщает оптимизатору, что таблицы надо соединять именно в том порядке, в каком они перечислены в запросе, а USE_NL(I) предписывает при соединении таблицы instrument, переименованной в запросе в i, использовать вложенные циклы.

Подсказка в MySQL — тоже комментарий, начинающийся с плюса, но ключевые слова в нём иные, чем в Oracle. Если запрос корректен с точки зрения обеих платформ, то он будет выполнен и там, и там, но каждый оптимизатор увидит лишь свои подсказки.

В Microsoft SQL Server подсказки встраиваются в SQL-оператор, поэтому запрос с подсказкой другой платформой понят не будет. В рассматриваемом запросе к подсказкам относятся ключевое слово LOOP и фраза OPTION (FORCE ORDER):

```
select
  m.name, i.name
from
  musician m inner LOOP join instrument i on m.instr_id=i.id
OPTION (FORCE ORDER)
```

Наконец, некоторые платформы позволяют создать план запроса и принудительно закрепить его в кеше запросов, чтобы оптимизатор использовал готовый план, а не строил новый.

Надо ли говорить, что интерфейсы ручного управления запросами и даже термины (за исключением hints), применяемые для описанных выше механизмов, отличаются от платформы к платформе. Ниже приведена сводная таблица используемых терминов.

	Параметры	Подсказки	Хранимые планы
Oracle	—	Hints (comments)	SQL Plan Baseline
Microsoft SQL Server	—	Hints (query hints, join hints, table hints)	Forcing a particular plan (sp_query_store_force_plan)
PostgreSQL	Planner Method Configuration	Hints[1]	Outline[2]; Specific plan[3]

[1] Реализованы в расширении pg_hint_plan.

[2] Реализованы в расширении pg_outline.

[3] Реализованы в расширении sr_plan.

	Параметры	Подсказки	Хранимые планы
IBM Db2	—	—	Optimization profiles and guidelines
MySQL	Switchable optimizations	Hints	—

Отметим, что все без исключения руководства рекомендуют ручное управление планами исключительно как последнее средство для быстрого исправления ошибок. Системно улучшить производительность базы данных позволит только корректный, полный и своевременный сбор статистики[1].

4.4. Реализация бизнес-логики

Средства работы с базами данных, о которых мы говорили (прежде всего язык SQL), являются декларативными. Но всё же для описания бизнес-процессов необходимы процедурные языки. Традиционные базы данных работают в архитектуре клиент-сервер, то есть логика исполняется приложением-клиентом, а обработка данных — сервером. Однако с точки зрения программиста работа со встраиваемыми базами данных (SQLite, Berkeley DB и др.) практически не отличается от работы с клиент-серверными платформами.

Клиентские модули

Для общения между клиентским приложением и сервером БД используется протокол, специфичный для конкретной платформы. Чаще всего описание протокола есть в открытом доступе: для платформ с открытым исходным кодом это стандартная часть документации, но даже такие компании, как Microsoft и SAP, публикуют описание своих протоколов. Нет документации лишь по протоколам, используемым в платформах Oracle[2](Oracle RDBMS, TimesTen, Essbase, Oracle NoSQL) и IBM (Db2, Netezza).

Реализация клиент-серверного протокола каждым приложением бессмысленна, поэтому с каждой платформой поставляется библиотека доступа к данным. Традиционно библиотека представляет собой объектные файлы или динамические библиотеки с кодом и заголовочные файлы для языка C.

---

1 И разумеется, неуклонное повышение квалификации разработчиков приложений.

2 Платформа MySQL также принадлежит Oracle, однако клиент-серверный протокол MySQL опубликован.

В конце 1980-х компания Microsoft совместно с Sybase и Tandem Computers начала работу по стандартизации библиотек доступа к реляционным базам данных; позже к группе разработчиков присоединились эксперты из Digital Equipment Corporation. Стандарт был опубликован в декабре 1991 года под названием SQLC (SQL Connectivity), а первую реализацию Microsoft и Simba Technologies выпустили в декабре 1992 года уже под новым названием — ODBC (Open Database Connectivity).

Поначалу драйвер ODBC включал в себя транслятор языка SQL, разработанный компанией Simba Technologies. Драйвер преобразовывал SQL-запросы на диалекте Simba во внутреннее представление, а затем из этого представления конструировал запрос на языке платформы, с которой работало приложение. Однако такой подход показал свою неэффективность, и сегодня SQL-запросы, передаваемые приложением в ODBC-драйвер, пересылаются серверу без изменений. Таким образом, миграция приложения с одной СУБД на другую требует переписывания SQL-запросов, даже если приложение использует ODBC.

Функции ODBC делятся на несколько групп.

- Соединение с базой данных: установка и разрыв соединения, получение различной информации о соединении — параметров драйвера, списка поддерживаемых функций и типов данных и т. д.
- Управление параметрами соединения: определение уровня изоляции транзакций, настройка буферов памяти и тайм-аутов, включение асинхронного обмена и т. д.
- Работа с каталогом, то есть извлечение списка таблиц и хранимых процедур, а также детальной информации о них: для таблиц — списка полей, первичных и внешних ключей, привилегий; для процедур — списка параметров.
- Выполнение SQL-запросов.
- Работа с подготовленными запросами: разбор SQL-запроса и получение его дескриптора, извлечение информации о параметрах запроса и привязка к этим параметрам переменных, исполнение подготовленного запроса с использованием дескриптора.
- Завершение команд: подтверждение или откат транзакции, а также досрочное прерывание исполняемого запроса.
- Работа с курсорами: получение описания курсора — количества и типа полей; привязка переменных к курсору; извлечение данных курсора; перемещение по курсору.

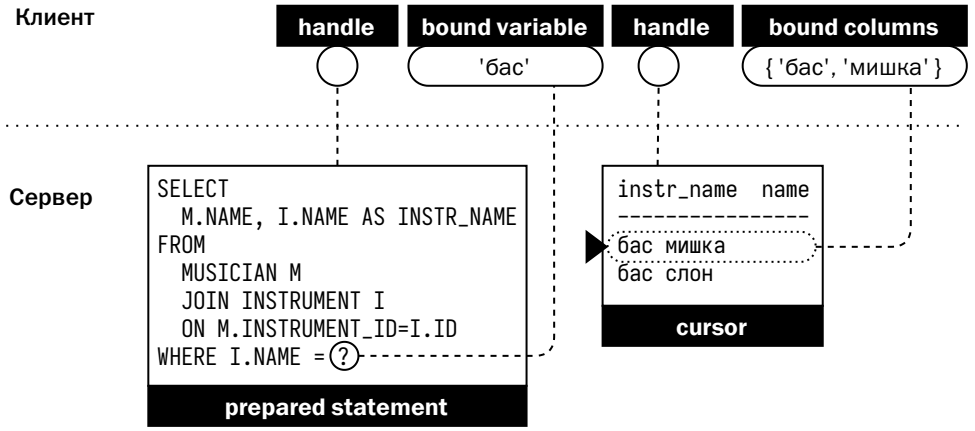
Курсором (cursor) называется временный объект, представляющий набор записей, полученных в результате исполнения запроса — команды `SELECT`, а также

`UPDATE`, `INSERT` или `DELETE` с фразой `RETURNING`. Курсор состоит из области памяти для хранения данных набора и указателя на текущую запись. В зависимости от режима, в котором открыт курсор, указатель может двигаться только вперёд вместе с извлечением текущей записи или перемещаться по набору в произвольном направлении (scroll). Курсоры с возможностью перемещения создают бóльшую нагрузку на сервер, чем курсоры, позволяющие только последовательно извлекать данные.

Типичный сценарий работы приложения состоит из следующих действий:

1. Подготовка запроса и получение его дескриптора.
2. Привязка переменных к формальным параметрам подготовленного запроса.
3. Запись фактических параметров в привязанные переменные.
4. Исполнение запроса; если запрос возвращает данные, то при выполнении запроса открывается курсор.
5. Итерация по данным курсора.
6. Закрытие курсора.

На рисунке ниже показаны структуры клиентского приложения и сервера на шаге 5.



Извлекать данные можно как по одной записи, так и по несколько. Извлечение нескольких записей требует большего объёма памяти в клиентском приложении, но существенно ускоряет работу с базой данных за счёт уменьшения количества сетевых обращений к серверу (roundtrips).

Помимо извлечения данных курсор иногда позволяет изменять извлечённые записи и сохранять изменения в исходных таблицах.

Как уже отмечалось выше, использование драйвера ODBC не гарантирует переносимость программ, поскольку тексты SQL-запросов специфичны для каждой платформы. Из-за этого ODBC-драйверы сегодня используются редко.

Интерпретируемые языки используют библиотеки для доступа к базам данных, которые обращаются к «родным» API каждой СУБД. Набор функций, предоставляемых этими библиотеками, примерно соответствует набору функций спецификации ODBC.

Стандарт некоторых языков (Perl, Python, Ruby) включает в себя API для работы с базами данных, и клиентский код для работы с разными платформами выглядит одинаково с точностью до диалекта языка SQL. Другие языки (PHP, JavaScript с фреймворком Node.js) не регламентируют интерфейс БД, и библиотеки для работы с разными платформами отличаются даже именами функций.

Компилируемые языки (C/C++, Go, Rust) чаще работают с «родными» API платформ и с построенными поверх них классами-обёртками.

Помимо функций, определяемых стандартом ODBC, «родные» API могут содержать функции, специфичные для конкретной СУБД:

- управление экземпляром БД — запуск и остановка экземпляра, создание и изменение базы данных, изменение параметров экземпляра…;
- мониторинг — получение данных мониторинга в формате, специфичном для конкретной СУБД;
- получение расширенной диагностической информации об экземпляре БД, клиентском программном обеспечении, пользовательском сеансе и т. д.;
- быстрая загрузка данных в обход транзакционного механизма БД;
- управление восстановлением при сбоях СУБД: уведомление приложения о сбоях, автоматическое подключение к резервной БД, повторное воспроизведение сбойных транзакций и т. п.;
- доступ к журналам БД для репликации;
- работа со специфическими типами данных — JSON, XML и др., а также с большими объектами (BLOB — Binary Large OBject);
- работа с национальными языками и форматами сообщений.

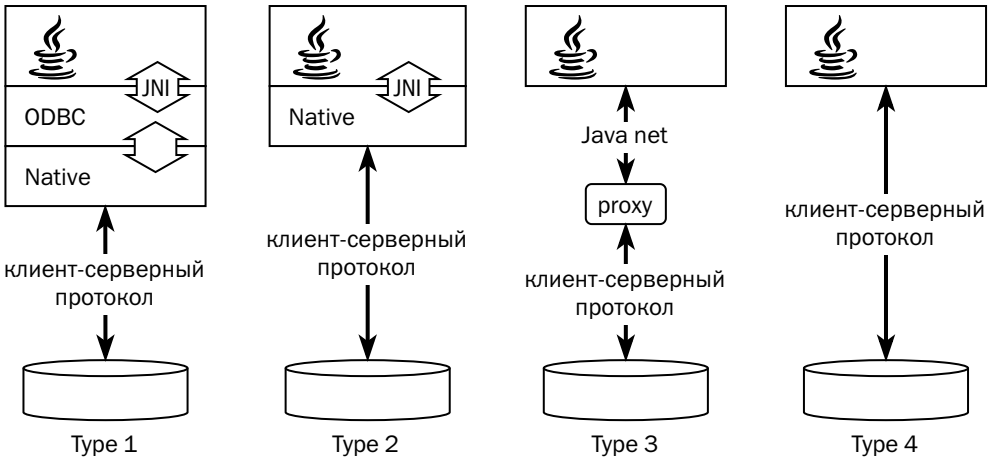
Полный API таких платформ, как Oracle и Db2, насчитывает более 500 функций.

В экосистеме Java для работы с базами данных используется интерфейс JDBC (Java Database Connectivity), предложенный компанией Sun Microsystems в 1997 году и являющийся частью стандарта Java SE.

JDBC содержит класс, позволяющий получить информацию о наборе доступных баз данных, а также несколько классов, описывающих типы хранимых в БД

данных, события (events), ошибки (errors) и исключения (exceptions). Кроме того, JDBC предоставляет около 40 интерфейсов (Java interface), которые должны быть реализованы поставщиком платформы в драйвере (JDBC driver) — библиотеке классов, обеспечивающих доступ приложений, выполняемых в виртуальной машине Java, к конкретной СУБД. Как и при использовании ODBC, код приложения, использующего JDBC, не зависит от типа базы данных, за исключением текста формируемых запросов.

Первые драйверы JDBC представляли собой классы, вызывающие из Java функции ODBC при помощи механизма Java Native Interface, JNI (JDBC-драйверы первого типа). Позже от использования ODBC производители перешли к использованию «родных» интерфейсов, также с вызовом «родного» кода операционной системы из виртуальной машины Java (драйверы второго типа), а затем — к использованию промежуточного ПО на стороне сервера (драйверы третьего типа). JDBC-драйверы четвертого типа реализуют клиент-серверный протокол на Java непосредственно в коде драйвера:



Подавляющее большинство современных JDBC-драйверов составляют драйверы типа 4, поскольку они не зависят от ОС и аппаратной платформы, как и весь код, выполняемый в виртуальной машине Java, а также обеспечивают максимальную производительность. Однако ряд драйверов (например, SQLite) относятся к типу 2, поскольку клиент-серверный протокол у встраиваемых СУБД отсутствует, и разработчики JDBC-драйвера вынуждены вызывать двоичный код «родного» драйвера.

Интерфейсы, входящие в стандарт JDBC, определяют те же группы функций, что и ODBC. В дополнение к этому JDBC содержит следующую функциональность:

- работа с данными в формате XML и с большими объектами: BLOB (Binary Large OBject) и CLOB (Character Large OBject);
- использование набора (пула) соединений с БД для выполнения потока транзакций;
- участие в распределённых транзакциях (XA).

Единственным стандартом для работы с нереляционными платформами является JSR 107, предоставляющий крайне простой и ограниченный интерфейс. Разработчикам, желающим полноценно использовать возможности баз данных NoSQL, следует изучать API конкретной платформы.

Для некоторых платформ (например, Cassandra или Tarantool) существует по нескольку клиентских модулей для популярных языков (Java, Go), поэтому перед началом разработки следует выбрать и клиентскую библиотеку.

ORM

Большая часть современного программного обеспечения пишется на объектно-ориентированных языках — Java (а также других языках, работающих с виртуальной машиной Java, — Kotlin, Scala, Groovy и др.), языках семейства .NET (C#, F#), Ruby, Python и других[1]. Для хранения данных, как правило, используются реляционные базы данных. Однако объектная модель данных, принятая в этих языках, отличается от реляционной. Список отличий довольно внушительный:

- Программисты на языках высокого уровня привыкли оперировать коллекциями, все элементы которых находятся в памяти. Традиционный подход в клиентском приложении, работающем с базой данных, — данные на диске и курсор, содержащий небольшую выборку данных, необходимых для выполнения текущей операции.
- Элемент коллекции — объект, элемент таблицы — строка, и преобразование объекта в строку реляционной таблицы и обратно может оказаться нетривиальным.
- Единственная связь, возможная между таблицами реляционной БД, — внешний ключ. Классы могут содержать ссылки на объекты других классов,

[1] Многие из перечисленных языков относят к мультипарадигменным, однако в контексте обсуждаемой темы важно, что они поддерживают объектную парадигму.

полностью включать в себя экземпляры других классов и даже коллекции экземпляров, а также могут наследовать друг другу.

- Объектные переменные считаются равными, если они ссылаются на один и тот же объект (==) либо значения всех полей объектов, на которые они ссылаются, совпадают (equals). В реляционных БД равенство строк — это равенство первичных ключей, что не соответствует ни одному из рассмотренных отношений между объектами.
- Традиционный для объектной модели доступ к объектам — переход от одного объекта к другому и извлечение объектов по мере необходимости. При работе с реляционными БД принято извлекать все связанные строки одним запросом, минимизируя количество обращений к серверу.

Большую популярность получили библиотеки класса ORM (object-relational mapping), избавляющие разработчика от множества рутинных операций, связанных с переходом от объектной модели к реляционной. В таблице ниже приведены примеры таких библиотек для разных языков.

Язык	Библиотеки
Java	Hibernate, EclipseLink, TopLink
.NET	ADO.NET Entity Framework, Castle ActiveRecord, NHibernate
Python	Django, SQLAlchemy
Ruby	ActiveRecord
JavaScript	Prisma, TypeORM
PHP	Doctrine

Объектный интерфейс для Java зафиксирован стандартом JPA (Jakarta Persistence API), описанным в JSR 220 (версия 1.0), JSR 317 (версия 2.0) и JSR 338 (версия 2.1).

Библиотека ORM позволяет создать модель данных в виде кода на языке, на котором будет написано клиентское ПО, и связать эту модель с реляционной моделью базы данных.

Способ объяснить компилятору, какие именно классы представляют собой сущности, хранимые в базе данных, зависит от языка и от библиотеки:

- аннотации классов (Hibernate);
- перечисление классов-сущностей в отдельном конфигурационном файле (Hibernate);

- наследование от класса типа «хранимая сущность» (Django);
- объявление классов-множеств, содержащих хранимые сущности (.NET Framework).

Существует три подхода к созданию модели данных и соответствующей ей структуре классов:

1. **«Сначала модель»** (model first). Разработка начинается с создания модели БД в каком-либо инструменте, внешнем и по отношению к языку, на котором разрабатывается клиентское приложение, и по отношению к базе данных. Впоследствии из этого описания генерируется и структура классов, и структура базы данных. Этот подход постепенно теряет популярность, поскольку добавляет к исходному тексту приложения лишний артефакт. В частности, Microsoft перестаёт поддерживать подход «сначала модель» при переходе от .NET Framework к более современному .NET Framework Core.
2. **«Сначала код»** (code first). Разработка начинается с проектирования структуры классов, а затем утилита, входящая в набор вспомогательных инструментов для библиотеки, генерирует скрипт для создания схемы БД (DDL).
3. **«Сначала данные»** (database first). Разработка начинается с создания модели базы данных, а затем специальная утилита генерирует из этой структуры классы для клиентского приложения.

Подход «сначала код» обычно используется на первых этапах разработки, но с переходом к промышленной эксплуатации приложения разработчики чаще отдают предпочтение подходу «сначала данные», то есть меняют структуру базы данных и переносят эти изменения в объектную модель. Подход «сначала данные», в отличие от остальных, позволяет в полной мере учитывать и использовать особенности каждой конкретной платформы — специфические индексы на основе битовых карт или хеш-таблиц, управление физическим размещением данных в файлах и табличных пространствах, локализацию данных (секционирование) и т. д.

ORM предоставляет функции для поиска объектов, навигации по объектам, изменения и удаления существующих объектов и создания новых, управления транзакциями. При этом библиотека формирует запросы к базе данных, берёт на себя извлечение данных (в том числе «ленивое», когда данные считываются в память клиентского приложения только в момент первого обращения), собирает объекты из прочитанных данных и раскладывает изменённые поля объектов по таблицам. В библиотеке обязательно присутствуют отладочные компоненты, позволяющие просматривать выполняемые запросы.

ORM может также управлять кешированием данных в клиентском приложении. Например, платформа Terracotta BigMemory выросла из проекта Ehcache, «простого кеша для Hibernate». Сегодня в качестве кеша для Hibernate, в т. ч. распределённого, можно использовать практически любую платформу класса in-memory data grid — Hazelcast, Apache Ignite, Oracle Coherence, Infinispan и другие.

Некоторые ORM позволяют писать запросы к базе данных на собственном языке — например, HQL в Hibernate и LINQ в ADO.NET Entity Framework. Эти запросы транслируются библиотекой в SQL-запросы к базе данных с учётом её особенностей, а также могут пользоваться результатами кеширования.

Использование готовых SQL-запросов в приложениях, написанных с применением ORM (например, nativeQuery в Hibernate), не рекомендуется. Такие запросы могут стать некорректными не только при миграции приложения на другую платформу, но и при изменении модели данных с помощью ORM.

Некоторые библиотеки поддерживают не только реляционные платформы, но и платформы с другими моделями данных. Так, например, Hibernate OGM (object-grid mapping) может работать с Infinispan, MongoDB и Neo4j, а также с любой платформой, для которой сторонние разработчики напишут адаптер[1]. Проект Django-nonrel, не входящий в официальный репозиторий Django, позволяет в качестве хранилища использовать такие платформы, как MongoDB, Elasticsearch, Cassandra, Redis. Разумеется, даже при использовании такого высокого уровня абстракции, как OGM, разработчик должен понимать и учитывать все ограничения, накладываемые распределёнными системами.

Хранимый код

Хранимый код (stored programs) — это программное обеспечение, которое исполняется непосредственно сервером базы данных. Большинство серверов баз данных хранят не только скомпилированный код, но и исходный текст этих программ, который самостоятельно компилируют во внутреннее представление.

Существует три типа объектов, содержащих хранимый код:

1. **Функция** (UDF, user-defined function) — программа, используемая в основном так же, как и встроенные функции, — в выражениях, содержащихся в тексте запросов. Функция не имеет побочных эффектов[2], то есть не

[1] На момент написания поддерживаются Apache Cassandra, CouchDB, Ehcache, Apache Ignite, Redis.

[2] Большинство платформ позволяют создавать функции с побочными эффектами, но по рассматриваемой классификации такая программа будет процедурой, несмотря на то что синтаксически оформлена как функция.

использует для вычисления результата никаких данных, кроме значений параметров, и не изменяет состояние базы данных.

2. **Процедура** (stored procedure)[1] — произвольный код, исполняемый на стороне сервера, когда этого явно требует вызывающая сторона (клиент или другая хранимая процедура). Хранимая процедура может читать и изменять произвольные данные. В зависимости от возможностей конкретного языка процедура может также возвращать одно или несколько скалярных значений или курсор.
3. **Триггер** (trigger) — код, исполняемый сервером автоматически по какому-либо условию — при обновлении данных или при наступлении системного события (открытия базы данных, открытия соединения с БД и др.).

Хранимый код в реляционных СУБД было принято писать на языке SQL, однако производители добавляли в SQL процедурные расширения — переменные, сложные типы данных (структуры, массивы, коллекции), команды управления потоком исполнения (ветвления, циклы) и другие, что в результате сделало эти расширения самостоятельными языками. Наиболее популярным можно считать основанный на Ada язык PL/SQL, развиваемый корпорацией Oracle с шестого релиза СУБД, выпущенного в 1998 году. Сегодня процедурные расширения регулируются стандартом SQL/PSM (SQL/Persistent Stored Modules), первая версия которого была принята в 1996 году и который в значительной мере повторяет PL/SQL.

Преимущество PL/SQL и подобных расширений перед традиционными языками программирования в том, что объекты доступа к данным являются не библиотечными функциями, а элементами языка. Рассмотрим в качестве примера цикл на языке PL/SQL:

```
for c in (
  select
    d.id, dd.id as ddid, amount,
    sum(amount) over(partition by d.id) as amount_t
  from document d join document_detail dd on d.id=dd.doc_id
  where d.date_doc>=p_date and d.date_doc<p_date+1
) loop
  insert into transaction
    values (v_trx_id+v_trx,'c','41.1',c.amount);
  v_trx := v_trx+1;
```

[1] Язык PL/SQL позволяет объединять процедуры и функции в пакеты. Код, использующий пакет, зависит только от его спецификации (заголовка), поэтому изменение тела пакета не требует перекомпиляции зависимого кода. Кроме того, пакет может иметь внутреннее состояние, хранящееся в памяти сеанса.

```
    if c.id<>v_cur_id then
      insert into transaction
        values (v_trx_id+v_trx,'d','50.1',c.amount_t);
      v_trx := v_trx+1;
    end if;
  end loop;
```

Приведённый код прозрачно для разработчика делает множество шагов:

- подготавливает запрос с параметрами;
- подставляет в подготовленный запрос фактические значения параметров;
- создаёт тип-структуру и переменную этого типа, а также привязывает поля структуры к колонкам запроса;
- открывает курсор;
- выбирает строки из курсора, помещая выбранные значения в поля созданной структуры, пока данные не закончатся.

В результате код для работы с данными, написанный на процедурных расширениях SQL, получается очень компактным. Однако многие СУБД позволяют писать хранимый код и на «обычных» процедурных языках:

Платформа	Расширение SQL	Другие поддерживаемые языки
Oracle	PL/SQL	Java
Oracle TimesTen	PL/SQL	
PostgreSQL	PL/pgSQL (близок к PL/SQL)	PL/Tcl, PL/Perl, PL/Python
Microsoft SQL Server	Transact-SQL	Языки платформы .NET
Db2	PL/SQL, SQL/PL (близок к SQL/PSM)	Java
MySQL	язык, близкий к SQL/PSM	
Informix	SPL	Java
Interbase/Firebird	PSQL	
SAP HANA	SQLScript	R

Многие нереляционные БД также позволяют создавать и выполнять хранимый код. Общего подхода к хранимому коду в нереляционных СУБД не существует; за подробностями следует обращаться к документации конкретной платформы.

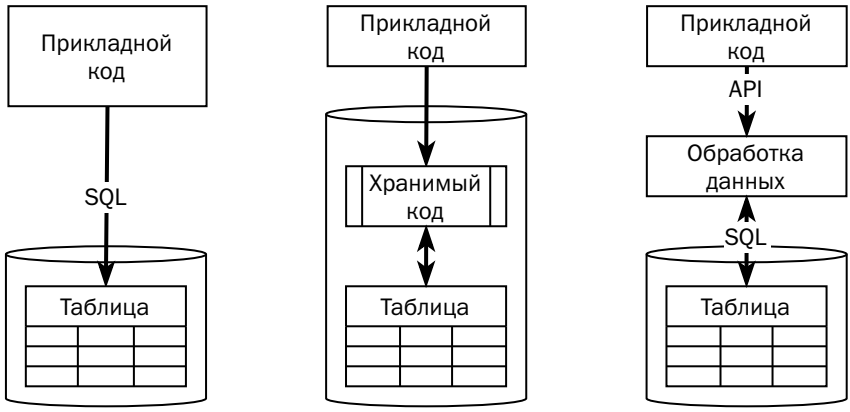
Язык	Платформы
Java	GridGain/Apache Ignite, Hazelcast, Infinispan, Pivotal GemFire/Apache Geode, Cassandra (только UDF и триггеры)

Язык	Платформы
Lua	Redis, Tarantool, Aerospike (только UDF)
JavaScript	MongoDB (только UDF)

Использование хранимого кода в базах данных даёт разработчику ряд преимуществ перед размещением кода бизнес-логики на выделенных серверах:

- **Высокая производительность**. Между базой данных и клиентским приложением передаются только аргументы вызова и результат, а не полный набор обрабатываемых данных. Такой подход радикально сокращает сетевой трафик и уменьшает время получения результата. Особенно это ускорение актуально для БД в памяти, где время передачи данных по сети может на порядки превосходить время доступа к данным.
- **Согласованность данных**. Логика приложения может требовать более сложных ограничений целостности, чем те, которые можно обеспечить при помощи ограничений на таблицы. Например, «проводка состоит из двух или более частей, среди которых есть как дебетовые, так и кредитовые, причём суммы по дебету и по кредиту равны». Использование хранимого кода позволяет унифицировать сложные проверки.
- **Гранулярное разграничение доступа**. Пользователю можно предоставить права на выполнение процедуры, но не давать доступа к данным в таблице. Пользователь получит только те данные, к которым у него должен быть доступ, причём соответствующим образом отфильтрованные и обработанные.

Последние два преимущества (соблюдение согласованности данных и разграничение доступа) могут быть получены и другим способом — выделением кода, работающего с данными, в специальные модули:



Есть у хранимого кода и недостатки:

- **Сложность масштабирования**. Мощность подсистем хранения (базы данных) и обработки (серверов бизнес-логики) данных может меняться по отдельности — в зависимости от того, какую нагрузку создаёт приложение. При использовании хранимого кода обе подсистемы жёстко связаны. Если платформа монолитная, как большинство реляционных платформ, то мощность обеих подсистем ограничена единственным сервером[1].
- **Снижение надёжности**. Несмотря на все ограничения, накладываемые механизмами исполнения хранимого кода, ошибки в пользовательских процедурах всё равно могут привести к нарушениям работы узлов вплоть до отказа. Обычно отказ узлов бизнес-логики не влияет на общую работоспособность приложения, отказ же узла хранения, который совмещён с узлом бизнес-логики, приводит к прерыванию сервиса.
- **Снижение удобства разработки**. Как правило, инструменты отладки, контроля версий и «бесшовного» обновления, интегрированные с СУБД, имеют ограниченную функциональность. А «канареечные» релизы, когда новая функциональность работает на части серверов приложений, в случае использования хранимого кода невозможны.
- **Привязка к платформе (vendor lock)**. Синтаксис и набор библиотек языка, на котором пишется хранимый код, уникальны для каждой платформы, поэтому использование хранимого кода затрудняет (и без того крайне трудную) миграцию или параллельную поддержку нескольких платформ.

Тенденция использования хранимого кода носит циклический характер: в 90-х годах XX века СУБД Oracle завоевала огромную популярность не в последнюю очередь благодаря возможности создания хранимых процедур, а сегодня большинство разработчиков стараются не использовать хранимый код. Самая популярная система управления предприятиями SAP R/3 не содержала ни строчки хранимого кода и поддерживала любые реляционные платформы — Oracle, Db2, Microsoft SQL Server. Современные версии SAP работают только с собственной платформой SAP HANA, и значительная часть функциональности реализована именно в виде хранимых процедур.

Исходя из опыта проектирования промышленных систем, автор рекомендует использовать хранимый код в распределённых СУБД в памяти и по возможности

[1] Разумеется, возможно шардирование средствами приложения или вынос читающей нагрузки на реплики.

не использовать в других платформах. Разумеется, ваше законное право — не прислушиваться к этой рекомендации.

В заключение — несколько слов о триггерах.

Главное отличие триггера от обычной хранимой процедуры в том, что программист не может явно вызвать триггер — платформа выполняет его автоматически. Наличие триггера создаёт иллюзию, что его код будет обязательно выполнен при наступлении соответствующего события. Однако триггер может быть отключен и включен заново, и в общем случае нет способа выяснить, какие изменения были внесены в данные с отключенными триггерами.

Есть несколько типичных задач, решаемых с помощью триггеров, однако для большинства из них есть и другие средства:

Задача	Современный подход
поддержка сложных ограничений целостности	реализация бизнес-логики в слое API
аудит изменения данных	подсистема аудита СУБД
генерация идентификаторов строк	создание колонки типа `IDENTITY` или генерация идентификатора в клиентском приложении — например, алгоритмами семейства «снежинка» (snowflake)
заполнение вычисляемых колонок	вычисляемые колонки в определении таблиц[1]
репликация данных	логическая репликация на основе журналов БД
ограничение доступа к данным	ограничение средствами приложения

Типовая задача, которая решается только триггерами, — создание изменяемых представлений, содержащих соединения таблиц. Выполняя команды DML над такими представлениями, СУБД не может определить, в какую из таблиц следует вносить изменения, и поэтому команда завершится с ошибкой, если только на представлении не определён триггер типа «instead of».

Автор рекомендует никогда не использовать триггеры. Но разумеется, если платформа предоставляет такую возможность, то разработчик волен ей воспользоваться.

[1] В разных платформах используются разные термины: virtual columns (Oracle), computed columns (Microsoft SQL Server), generated columns (Db2, PostgreSQL, MySQL).

Литература

- Apache Cassandra: How is data read?
  docs.datastax.com/en/cassandra-oss/3.x/cassandra/dml/dmlAboutReads.html
- PostgreSQL Buffer Ring Replacement Strategy.
  github.com/postgres/postgres/blob/master/src/backend/storage/buffer/README#L208
- Jonathan Lewis. Clustering_Factor.
  jonathanlewis.wordpress.com/2019/10/23/clustering_factor-7/
- Kendra Little. Truncate Table with Partitions Fails if You Have Non-Aligned Indexes.
  www.littlekendra.com/2016/03/22/
  truncate-table-with-partitions-fails-if-you-have-non-aligned-indexes/
- DataStax Enterprise Architecture Guide: How indexes are stored and updated?
  docs.datastax.com/en/dse/5.1/dse-arch/datastax_enterprise/dbInternals/dbIntIndexInternals.html
- Егор Рогов. PostgreSQL 15 изнутри. М.: ДМК Пресс, 2023.
  postgrespro.ru/education/books/internals
- Oracle: using rule-based optimizer.
  docs.oracle.com/cd/B10500_01/server.920/a96533/rbo.htm
- Владимир Озеров. Как мы делали SQL в Hazelcast.
  assets.ctfassets.net/oxjq45e8ilak/55zNvFHRSQCAE3LTar6axR/
  2843a7226d92b1de867293aeaf9d9815/
  Vladimir__Ozerov_Kak_my_delali_SQL_v_Hazelcast_2020_11_25_17_41_31__1_.pdf
- Db2 optimization classes.
  www.ibm.com/docs/en/db2/11.5?topic=plans-optimization-classes
- Mahmmoud Adel. All About Statistics In Oracle.
  dba-tips.blogspot.com/2012/11/all-about-statistics-in-oracle.html
- Jonathan Lewis. Clustering factor.
  jonathanlewis.wordpress.com/2018/07/02/clustering_factor-5/
- Execution Plan Cost Model.
  www.qdpma.com/CBO/ExecutionPlanCostModel.html
- PostgreSQL: Константы стоимости для планировщика.
  postgrespro.ru/docs/postgresql/13/runtime-config-query#
  RUNTIME-CONFIG-QUERY-CONSTANTS
- Oracle system statistics.
  docs.oracle.com/en/database/oracle/oracle-database/21/tgsql/optimizer-statistics-concepts.html#GUID-FE339C05-E7A4-4953-89C4-D5C69F0C08BA

- Db2 Configuration parameters that affect query optimization.

  www.ibm.com/docs/en/db2/11.5?topic=
  parameters-configuration-that-affect-query-optimization
- Bill Arledge. Demystifying the Db2 Dynamic Statement Cache.

  slideplayer.com/slide/14079485/
- Kalen Delaney. SQL Server Internals: Plan Caching Pt. I — Reusing Plans.

  www.spotlightcloud.io/blog/sql-server-internals-plan-caching-pt-i-reusing-plans
- Db2 Access Plan reuse.

  www.ibm.com/docs/en/db2/11.5?topic=plans-access-plan-reuse
- Oracle Adaptive Query Plans.

  docs.oracle.com/en/database/oracle/oracle-database/21/tgsql/query-optimizer-concepts.html#
  GUID-5A1EB094-1A9E-4B69-9BE5-39BDA2B3253C
- Microsoft SQL Server: Руководство по архитектуре обработки запросов.

  docs.microsoft.com/ru-ru/sql/relational-databases/query-processing-architecture-guide?view=
  sql-server-ver15
- ODBC Function Summary.

  docs.microsoft.com/en-us/sql/odbc/reference/syntax/odbc-function-summary?view=
  sql-server-ver15
- Javin Paul. Difference between Type 1, 2, 3 and 4 JDBC Driver in Java?

  www.java67.com/2015/07/difference-between-type-1-2-3-and-4-jdbc-drivers-java.html
- Введение в ORM (Object Relational Mapping).

  internetka.in.ua/orm-intro/
- Всё о триггерах в Oracle.

  habr.com/ru/post/256655/

Часть III

Архитектура СУБД

Глава 5
Гарантии корректности данных

Целое больше суммы своих частей.

Аристотель, «Метафизика»

Работая с базой данных, пользователь ожидает, что данные сохраняются корректно. В этой главе мы попытаемся выяснить, что же именно понимается под «корректностью», и рассмотрим механизмы, которые её обеспечивают.

5.1. Транзакции

Работа с базой данных состоит из **транзакций (transactions)** — наборов операций над данными. Список свойств, которые приложение ожидает от транзакции, был сформулирован в конце 1970-х годов Джимом Греем (James Nicholas "Jim" Gray). Эти свойства описываются аббревиатурой ACID:

- **Атомарность (Atomicity)**: либо все операции, входящие в группу (транзакцию) выполняются, либо не выполняется ни одна из них.
- **Согласованность (Consistency)**: транзакция переводит базу данных из одного согласованного состояния в другое[1].
- **Изоляция (Isolation)**: транзакции выполняются так, чтобы не мешать друг другу, то есть одновременно выполняемые транзакции не должны влиять на результат друг друга.
- **Долговечность[2] (Durability)**: если клиент получил от базы данных ответ, что транзакция успешно завершена, то результат этой транзакции не исчезнет, что бы ни случилось.

Описания свойств транзакции («согласованное состояние», «не мешать», «что бы ни случилось») весьма расплывчаты, и далее мы постараемся их уточнить.

---

[1] Определение согласованности получилось рекурсивным, но позже мы его раскроем.

[2] Правильнее, конечно, сказать «надёжность», но термин «долговечность» сохраняет аббревиатуру «АСИД».

Заметим также, что далеко не все платформы обеспечивают эти свойства в полном объёме. Так, например, в описании ранних распределённых платформ упоминается альтернативный подход — BASE (Basically Available, Soft state, Eventually consistent), который весьма вольно трактует согласованность и изоляцию. Современные же платформы, включая распределённые, стремятся предоставлять разработчикам полноценные ACID-транзакции.

В текущей главе мы поговорим о механизмах, используемых монолитными системами, а в разделе, посвящённом распределённым системам, обсудим алгоритмы, позволяющие согласованно изменять данные на нескольких узлах, обеспечивая в распределённых системах те же гарантии, что и в монолитных.

Атомарность

Гарантия атомарности заключается в том, что транзакция либо будет выполнена целиком, либо не будет выполнена вообще. Тонкость в том, что именно считать транзакцией. Как правило, платформы выбирают одно из трёх определений:

1. Транзакцией считается произвольное количество операций, выполненных между явным началом и окончанием транзакции. Реляционные СУБД обычно автоматически открывают транзакцию при выполнении любой команды, изменяющей данные[1] (`INSERT`, `UPDATE`, `DELETE`, `MERGE`), а команда завершения транзакции называется `COMMIT`. Интерфейсы нереляционных платформ содержат другие методы управления транзакциями — например, пару функций `begin_transaction()` и `commit_transaction()`. Как правило, гарантии первого типа предоставляют монолитные платформы (Oracle, PostgreSQL, Microsoft SQL Server и др.).
2. Каждая операция обновления данных представляет собой транзакцию. Как правило, такие гарантии предоставляют распределённые системы[2] (Cassandra, MongoDB и др.).

---

[1] Здесь мы говорим только об операциях изменения данных (DML, data manipulation language), но не об операциях изменения структуры данных (DDL, data definition language). Такие платформы, как Oracle и MySQL, для каждой операции определения данных автоматически открывают и закрывают транзакции. PostgreSQL полностью поддерживает транзакции для DDL, а Microsoft SQL Server и Db2 — частично.

[2] Клиентские приложения и драйверы монолитных систем могут иметь режим автоматического подтверждения транзакций, autocommit, при включении которого команда `COMMIT` выполняется автоматически после каждой команды изменения данных. Команда языка SQL может изменить произвольное количество данных.

3. Транзакцией считается любое количество операций, выполненных в одном пакете. Разница с первым типом заключается в том, что пользователь не может получить управление в середине транзакции: весь пакет операций должен быть отправлен одномоментно. Такие гарантии предоставляют СУБД, использующие сериализацию транзакций в потоке, — например, Redis[1] и VoltDB.

Некоторые современные распределённые платформы (например, Apache Ignite, MongoDB и др.) предлагают два вида гарантий — атомарные операции (atomic, тип 2 по вышеприведённой классификации) и полноценные транзакции (transaction, тип 1). Другие распределённые платформы (так называемые «NewSQL», например YugabyteDB, CockroachDB и др.) изначально рассчитаны на то, что пользователь будет работать с ними так же, как с монолитными системами, и потому обеспечивают полноценные транзакции (тип 1).

Отметим, что атомарность — первое, но не единственное требование к транзакции, и из выполнения какого-то одного требования не следует выполнение остальных. Так, например, при определённых настройках в распределённых системах (например, `CONSISTENCY ONE` в Apache Cassandra или `writeConcern:{w:"1"}` в MongoDB) атомарное изменение может потеряться, нарушив требование надёжности, или дойти до разных читателей с разными задержками, нарушив требование изоляции.

Согласованность

Этот термин разные авторы понимают по-разному, поэтому каждый раз необходимо уточнять, о какой именно «согласованности» идёт речь. Когда говорят об ACID применительно к СУБД, то для каждой платформы или класса платформ этот термин может означать что-то своё. Типичные гарантии выглядят так:

- Платформы класса «ключ—значение» гарантируют уникальность ключа. Границы этой уникальности (вся база, коллекция и т. д.) зависят от возможностей конкретной платформы. Кроме того, гарантируется, что в базе хранится значение, записанное одной из операций, то есть не может быть значения, часть которого записана одной операцией, а часть — другой.

[1] Пакетная загрузка (bulk load) в Redis работает с единственным шардом и позволяет только записывать данные, но не позволяет читать и в рамках транзакции принимать решения на основании прочитанного.

- Платформы, имеющие представление о структуре хранимых данных (документоориентированные, семейства колонок, графовые, реляционные), гарантируют, что каждый элемент (колонка в реляционной БД или хранимое значение целиком в документоориентированной) корректен с точки зрения формата и типа данных.
- Реляционные платформы дополнительно гарантируют выполнение декларативных ограничений целостности (integrity constraints) — допустимых диапазонов значений колонки, уникальных и внешних ключей.
- Если платформа поддерживает вспомогательные структуры данных (например, вторичные индексы), то она гарантирует, что эти структуры изменяются синхронно с остальными. Другими словами, приложение должно получить одинаковые результаты запроса независимо от того, читает оно данные непосредственно из таблиц (коллекций) или из каких-то вспомогательных структур[1].

Важно, однако, понимать, что требования к согласованности со стороны приложения могут быть значительно сильнее, чем то, что способна обеспечить база данных. Классический пример такого требования — нулевой баланс операций, то есть требование, чтобы при перемещении денег на счёт зачислялось ровно столько, сколько снято с другого счёта. За его выполнением может следить приложение, но не база данных. Если платформа предоставляет гарантии транзакционной целостности первого или третьего типа, приложение может выполнять операции списания и зачисления в одной транзакции, в противном случае приложение должно обеспечить выполнение своих требований при помощи иных механизмов.

Изоляция

Изоляция транзакций, как сказано выше, позволяет транзакциям выполняться, не мешая друг другу.

Идеальное с точки зрения изоляции поведение базы данных — последовательное выполнение, когда исход транзакции зависит только от состояния данных перед её началом и от действий внутри самой транзакции. На практике обычно несколько транзакций выполняются параллельно, и СУБД должна создать у каждой транзакции иллюзию, что она — единственная в данный момент времени.

[1] В современных СУБД приложение чаще всего не может непосредственно влиять на то, какие именно структуры используются при чтении данных. Эквивалентность содержимого разных структур используется самими платформами при поиске оптимального способа извлечения данных.

Глубина иллюзии называется **уровнем изоляции (isolation level)**. Наивысший уровень изоляции — сериализация (serializable), при которой эффект параллельного выполнения транзакций в точности соответствует эффекту их последовательного выполнения. Обеспечение сериализации весьма затратно для СУБД, и поэтому в большинстве платформ существуют более слабые уровни изоляции, когда одновременное выполнение транзакций может приводить к нежелательным эффектам, называемым **аномалиями (phenomena)**.

Именно через возможные аномалии определяются уровни изоляции, поэтому перечислим здесь некоторые известные аномалии и посмотрим, как они возникают.

Грязная запись (dirty write): две транзакции конкурентно изменяют одну и ту же строку[1]. Ни одна из современных платформ не допускает такой аномалии, поскольку возможность одновременно изменять одну и ту же строку равносильна отсутствию изоляции.

Первая транзакция	Вторая транзакция
	Создаёт или изменяет строку
Перезаписывает строку, изменённую параллельной транзакцией	
	Подтверждает изменения (commit)

Грязное чтение (dirty read): транзакция читает данные, которых никогда не было в БД.

Первая транзакция	Вторая транзакция
	Создаёт или изменяет строку
Читает строку, изменённую, но не подтверждённую параллельной транзакцией	
	Откатывает изменения (rollback)

[1] Аномалии традиционно формулируются в терминах реляционных БД, поэтому здесь и далее используется термин «строка» (row). От замены «строки» на «объект» или «запись» суть аномалий не изменяется.

Потерянные изменения (lost updates): транзакция перезаписывает данные, которые обрабатывает другая транзакция.

Первая транзакция	Вторая транзакция
	Читает строку или набор строк
Изменяет строки, прочитанные параллельной транзакцией, и подтверждает изменения	
	Изменяет прочитанные строки и подтверждает изменения, уничтожая данные, записанные первой транзакцией

Неповторяющееся чтение (non-repeatable read): транзакция несколько раз считывает одни и те же строки и получает разные результаты.

Первая транзакция	Вторая транзакция
Читает строку	
	Изменяет строку, прочитанную первой транзакцией, и подтверждает изменения
Повторно читает ту же строку, получая другой результат	

Несогласованное чтение (read skew): транзакция получает некорректную информацию, читая несколько строк из одной таблицы или стро́ки из нескольких таблиц. Часть прочитанных данных соответствует результату выполнения одной последовательности транзакций, а другая часть — результату другой последовательности.

Первая транзакция	Вторая транзакция
Читает строку	
	Изменяет какую-то строку, отличную от той, которую прочитала первая транзакция, и подтверждает изменения
Читает строку, изменённую второй транзакцией	

Несогласованная запись (write skew): транзакция делает расчёты на основании информации, которую изменяет другая транзакция, и сохраняет результат. Эта аномалия похожа на потерянные изменения с той разницей, что в конкурирующих транзакциях изменяется не одна и та же строка, а разные. Из-за комбинации изменений, внесённых транзакциями, целостность набора нарушена.

Первая транзакция	Вторая транзакция
Читает набор строк, проверяет, что вносимые изменения не нарушат целостности набора	
	Читает набор строк, проверяет, что вносимые изменения не нарушат целостности набора. Вносит изменения и подтверждает их
Вносит изменения в базу и подтверждает их	

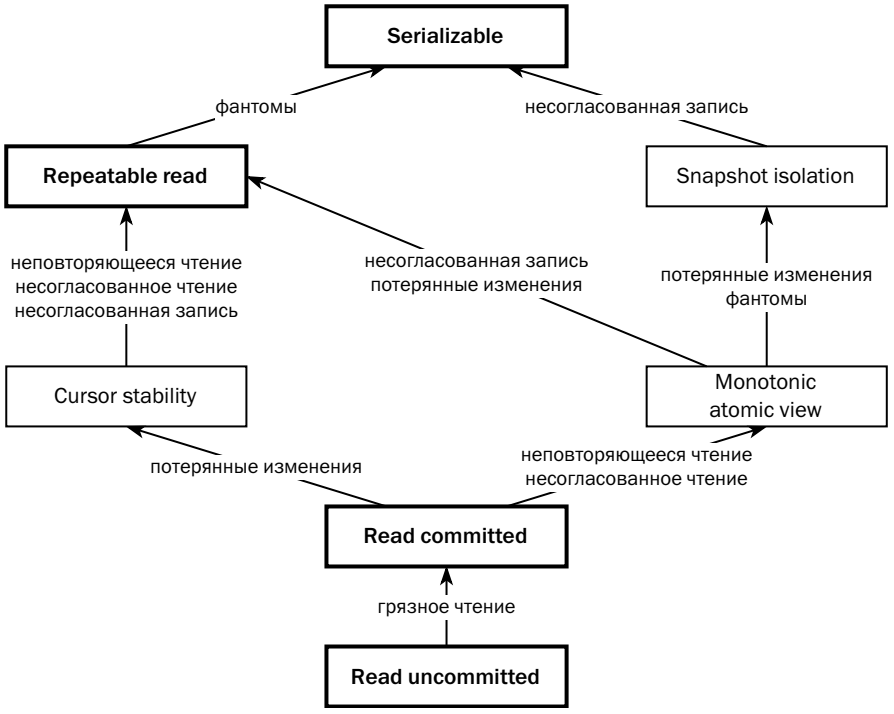
Фантомное чтение (phantom read): транзакция несколько раз выполняет один и тот же запрос, получая в результате разные наборы строк.

Первая транзакция	Вторая транзакция
Запрашивает набор строк, удовлетворяющих условию	
	Создаёт новую строку, удовлетворяющую условию, и подтверждает изменения
Повторно запрашивает тот же набор строк, получая дополнительные строки	

Поскольку никто не может дать гарантий, что список аномалий полон[1], нет и гарантии, что возможны только те уровни изоляции, которые перечислены в таблице ниже. Стандарт ANSI SQL ограничивается рассмотрением только аномалий чтения и определяет всего четыре уровня — read uncommitted, read committed, repeatable read и serializable. Но на практике это не так важно, поскольку у каждой платформы — свой набор уровней изоляции и своя классификация.

[1] Например, в списке не упомянута аномалия читающей транзакции (read-only transaction anomaly).

Уровень изоляции	Грязное чтение	Потерянные изменения	Неповторяющееся чтение	Несогласованное чтение	Несогласованная запись	Фантомы
Read uncommitted	+	+	+	+	+	+
Read committed	–	+	+	+	+	+
Cursor stability[1]	–	–	+[2]	+	+	+
Monotonic atomic view	–	+	–	–	+	+
Repeatable read	–	–	–	–	–	+
Snapshot isolation	–	–	–	–	+	–
Serializable	–	–	–	–	–	–



[1] Вводится понятие «курсора» — одного из наборов записей, с которыми работает транзакция; неизменность данных гарантируется, пока курсор не закрыт. Завершение транзакции закрывает курсоры автоматически.

[2] Аномалия возможна, если для повторного считывания открывается новый курсор.

Уровни изоляции образуют иерархию. На диаграмме стрелка указывает на более сильный уровень изоляции, а надпись на стрелке означает аномалию, от которой мы избавляемся по сравнению с более слабым уровнем. Жирным контуром выделены уровни, определённые в стандарте ANSI SQL.

При добавлении новых аномалий и новых уровней граф на рисунке выше может расти вбок, но не вверх: при сериализации транзакций не может возникать никаких аномалий, включая неизвестные.

Стандарт предписывает системам управления базами данных поддерживать все уровни изоляции, однако на практике каждая платформа реализует те уровни, которые кажутся важными её создателям или которые позволяет реализовать архитектура платформы. Список избранных платформ и названий поддерживаемых ими уровней изоляции приведён в таблице ниже[1].

	Read uncommitted	Read committed	Cursor stability	Repeatable read	Snapshot isolation	Serializable
Oracle		+				+
Db2	+		+	+		+
Microsoft SQL Server	+	+		+	+	+
PostgreSQL	+	+		+		+
MySQL	+	+		+		+
MongoDB[2]	+	+			+	
SQLite					+	
CockroachDB						+
YugabyteDB		+			+	+

Таблица носит лишь справочный характер; при разработке приложений, полагающихся на гарантии базы данных относительно изоляции транзакций, следует внимательно прочесть документацию и даже провести серию тестов, показывающих, что поведение платформы соответствует (или не соответствует) ожидаемому.

[1] Уровень Monotonic atomic view в реальных платформах не встречается.

[2] Начиная с версии 4.2, где доступны многодокументные транзакции.

Внимательный читатель может покритиковать таблицу, заметив, как минимум, что:

- в Oracle уровень `serializable` обеспечивает лишь snapshot isolation;
- в Db2 уровень `repeatable read` обеспечивает serializable, а «истинный» repeatable read называется `read stability`;
- в PostgreSQL грязное чтение невозможно, и уровень `read uncommitted` на самом деле эквивалентен уровню `read committed`;
- в MongoDB используется собственная терминология для обозначения уровней изоляции; так, `local` соответствует read uncommitted, а `majority` — read committed.

Возможны и другие нюансы реализации изоляции в различных платформах.

Долговечность (надёжность)

Под надёжностью традиционно понимают устойчивость к сбоям. Если клиент получил от базы данных подтверждение, что его транзакция успешно завершена, то что бы ни случилось с оборудованием, платформа гарантирует, что когда база данных снова будет доступна, результаты этой транзакции будут видны.

Обычно, говоря о «надёжности» применительно к базе данных, имеют в виду устойчивость к сбоям сервера — например, отключению питания. При этом подразумевается, что диск надёжен и всегда доступен, и после восстановления или замены сервера к серверу можно будет подключить тот же диск. Кроме того, не накладывается никаких ограничений на время восстановления доступности данных — важно, чтобы доступ в принципе можно было восстановить. И наконец, считается, что все действия пользователя корректны, то есть если данные уничтожены вследствие ошибки пользователя или администратора, то контракт платформы не считается нарушенным.

Далее в этом разделе мы будем говорить о надёжности именно с такими допущениями, а надёжности в более широком смысле посвящена часть V «Восстановление при сбоях».

Напомню, что надёжность следует воспринимать только в связке с согласованностью. Если транзакция не завершена, то база данных не просто не обязана сохранить результаты этой транзакции, но более того — обязана сделать так, чтобы после восстановления никаких следов этой транзакции не осталось.

Далее мы рассмотрим известные механизмы обеспечения целостности данных и убедимся, что разработчики стандарта ANSI SQL, скорее всего, думали в терминах блокировок, о которых поговорим в этом разделе чуть позже.

5.2. Журналирование

Важнейшим средством обеспечения перечисленных в предыдущем разделе свойств транзакций (за исключением изоляции) является журнал, куда последовательно и максимально компактно записываются все изменения, сделанные в базе данных.

Назначение журнала

В документации к разным платформам журнал называется журналом транзакций (transaction log), журналом завершённых транзакций (commit log), журналом повторного выполнения (redo log) или журналом опережающей записи (write-ahead log, WAL). Эти термины отражают разные аспекты работы с журналом.

Термин «журнал транзакций» говорит о том, что в журнал записывается информация об изменении данных и о завершённых транзакциях. Для платформ, использующих изменяемые страничные структуры хранения, это физические изменения, то есть последовательности байтов, которые нужно перенести в страницу. Для платформ, использующих неизменяемые структуры на основе сортированных таблиц (SSTables) и LSM-деревьев, журнал содержит логические изменения, то есть ключи изменённых записей и их значения (или только значения изменённых атрибутов).

Транзакциями содержимое журнала не исчерпывается. В журнал могут быть записаны системные события — например, изменение физической структуры базы данных (добавление или удаление файла). Детальный список событий, подлежащих записи в журнал БД, достаточно велик и специфичен для каждой платформы. Так, например, в документации к Db2 этот список содержит четыре десятка позиций.

«Повторное выполнение» означает, что при начале работы с базой данных экземпляр проверяет согласованность файлов данных. Если обнаруживается несогласованное состояние, то сервер повторно выполняет транзакции, сохранённые в журнале, тем самым восстанавливая изменения файлов данных, которые не дошли до диска, пропав вместе с состоянием оперативной памяти экземпляра.

«Опережающая запись» значит, что информация о транзакции записывается в файл журнала раньше, чем в файлы данных, и даже раньше, чем система ответит пользователю, что транзакция успешно завершена. Такой режим работы

гарантирует, что при сбое сервера информация о завершённой транзакции не будет потеряна, т. к. она уже надёжно сохранена в журнале.

Ещё одна функция, которую может выполнять журнал, — хранение информации для отката транзакций. При восстановлении базы данных после сбоя происходит повторное выполнение транзакций, которые не успели сохранить своё состояние на диск, а потом для транзакций, которые к моменту сбоя не успели завершиться, выполняются компенсирующие действия на основе информации отката. Механизм входит в семейство алгоритмов ARIES, описанное инженерами компании IBM в 1992 году. В большинстве современных платформ журнал не выполняет эту функцию, а откат при восстановлении происходит за счёт механизмов версионирования, о которых поговорим в этой главе позже.

Кроме восстановления данных при сбое, журнал можно использовать и для репликации[1], в том числе логической, поэтому в современных платформах со страничной организацией хранения в журнал можно добавить и логические изменения, но только в дополнение к физическим.

Работа с журналом

Любая операция модификации меняет данные в оперативной памяти — в кеше, если БД дисковая, или в основном хранилище, если БД в памяти. Информация об изменении записывается также в журнальный буфер (log buffer), работающий по принципу кольцевого буфера (ring buffer). Данные из журнального буфера постоянно записываются на диск в файлы журнала, поэтому его размер, как правило, ограничен несколькими мегабайтами, в отличие от буферного кеша и областей сортировки, размер которых может достигать сотен гигабайт и даже нескольких терабайт.

Когда объём информации в буфере превышает некоторый порог, фоновый процесс записывает содержимое буфера в файл на диске. При завершении транзакции сеанс записывает в журнальный буфер маркер окончания транзакции и даёт сигнал процессу записи журнала принудительно сбросить содержимое буфера на диск. Управление пользователю возвращается только после того, как содержимое буфера успешно записано до позиции маркера завершения.

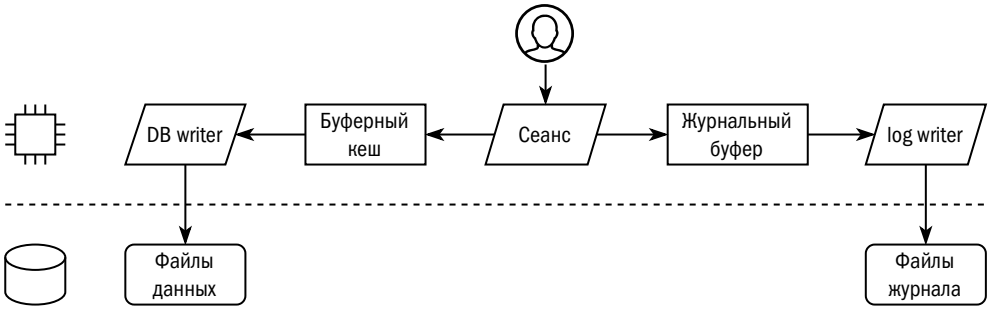
Принцип опережающей записи в журнал соблюдается не всегда — например, Apache Cassandra с настройкой по умолчанию `commitlog_sync = periodic` записывает содержимое буфера на диск только после заполнения или при истечении

[1] Подробнее о репликации — в главе 9 «Репликация».

периода времени. Более того, даже запись данных на диск при помощи функций операционной системы не гарантирует надёжного сохранения: у операционной системы есть собственный файловый кеш, и успешная запись означает всего лишь успешное помещение в кеш. Для обеспечения записи именно на диск используется системный вызов `fsync()` или специальный режим работы с файлами без кеширования (`O_DIRECT`), и администратор базы данных отвечает за корректную настройку синхронизации журнала.

На самом деле даже запись на диск не всегда гарантирует надёжность, поскольку у диска может быть собственный кеш, содержимое которого теряется при отключении питания. К счастью, корректная настройка дисковой подсистемы — зона ответственности системного администратора.

В системах, рассчитанных на интенсивную обработку транзакций, завершение транзакций происходит значительно чаще, чем процесс записи журнала способен сбрасывать на диск содержимое буфера, поэтому некоторые современные платформы (Oracle, PostgreSQL, Apache Cassandra) позволяют настроить пакетирование транзакций. Журнал записывается по таймеру примерно раз в 10–500 мс, а пользователь ждёт не только изменения данных, но и записи журнала. Такой подход увеличивает время каждой транзакции (latency), но существенно повышает общую производительность системы (throughput).



Возникает вопрос: почему бы не сохранять данные сразу в те структуры, из которых они потом будут прочитаны?

Для надёжного сохранения структуры на основе LSM-дерева без журнала пришлось бы после каждого изменения записывать на диск всю таблицу в памяти, что по объёму существенно больше изменённого значения.

При страничной организации дисковой памяти после любого изменения требуется записывать на диск целую страницу, размер которой больше, чем размер изменённых данных. Замеры, сделанные на промышленных БД, показывают, что

объём записи в файлы данных в 5–7 раз превосходит объём записи в журнал[1]. Кроме того, страницы с изменёнными данными хаотично разбросаны, а запись в журнал идёт последовательно, что позволяет писать на жёсткий диск очень быстро. В 80-х и 90-х годах XX века даже существовала рекомендация для достижения максимальной производительности базы данных использовать под журнал выделенный физический диск.

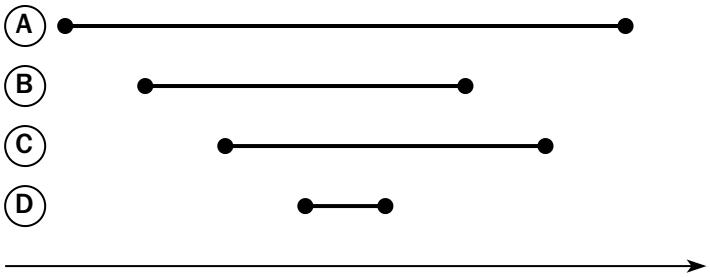
С повсеместным распространением RAID-массивов рекомендация потеряла актуальность, поскольку у дисковых массивов всегда есть собственный кеш с резервным питанием, и запись с сервера в массив выполняется быстро независимо от позиции, а реальная запись на диски происходит асинхронно.

Появление SSD и энергонезависимой памяти совершило ещё одну революцию на рынке систем хранения данных. Производительность записи многократно выросла, при этом SSD может параллельно выполнять множество операций, и скорость их выполнения не зависит от того, как по отношению друг к другу расположены записываемые блоки данных. Возможно, в ближайшем будущем мы увидим СУБД, обходящиеся без журнальных файлов.

Транзакция может быть сколь угодно длинной и затрагивать произвольное количество объектов, поэтому одной транзакции может соответствовать несколько журнальных записей.

Транзакции в журнале упорядочены, и их порядок соответствует порядку меток завершения. Если система многопоточная, как большинство современных СУБД, и может параллельно выполнять несколько транзакций, то журнальные записи одной транзакции не обязательно идут подряд. В этом случае порядок завершения транзакций может не совпадать с порядком их начала.

Пусть, например, в системе выполняется четыре транзакции:



[1] Сравнение объёма реплицируемых данных при использовании репликации средствами СХД и физической репликации БД при помощи передачи журнальной информации. Подробности — в главе 9 «Репликация».

Транзакции начинаются в порядке A-B-C-D, но порядком их выполнения будет считаться D-B-C-A. Содержимое журнала базы данных во время выполнения этих транзакций могло бы выглядеть, например, так:

Серым выделены записи, содержащие метки окончания транзакции. Количество записей для разных транзакций разное и зависит от их продолжительности — так, например, для транзакции A понадобилось четыре записи, а вся информация о транзакции D уместилась в одну.

Сохранение в файлы данных всех изменений, информация о которых находится до определённой позиции в журнале, называется контрольной точкой (checkpoint). После выполнения контрольной точки все файлы обновлены, и для восстановления данных при сбое журнальная информация до позиции контрольной точки не нужна.

Администратор базы данных может установить контрольную точку вручную, но в основном сервер обновляет файлы данных автоматически. К установке контрольной точки и началу сброса данных из кеша приводит множество событий, например переключение на следующий файл журнала.

При настройке производительности важно подобрать оптимальный размер журнальных файлов. С одной стороны, использование слишком маленьких файлов приводит к частым контрольным точкам и лишней нагрузке на подсистему ввода-вывода. Слишком большие файлы и редкие контрольные точки ведут к неудобствам при резервном копировании и к длительному восстановлению при сбоях. Эмпирическое правило гласит, что база данных при интенсивной нагрузке должна переходить к новому журналу каждые 5–10 минут. Естественно, в периоды спада нагрузки смена журнальных файлов будет происходить реже.

СУБД позволяют настроить либо временной интервал, либо объём накопленной журнальной информации, после которого начинается контрольная точка. Некоторые платформы позволяют настроить даже желаемое время восстановления при сбое, которое зависит не только от длины журнала, но и от текущей нагрузки на сервер. Разумеется, это время оценивается сервером приблизительно на основании некоторых эвристик. Желаемое время восстановления задаётся администратором базы данных через параметры системы (Oracle — `FAST_START_MTTR_TARGET`, Microsoft SQL Server — `TARGET_RECOVERY_TIME`, Db2 — `softmax` или `page_age_trgt_mcr` и `page_age_trgt_gcr` и т. д.).

Экземпляр базы данных постоянно пишет данные в журнальные файлы, но читает в исключительных случаях — при восстановлении после сбоя или при репликации с задержкой.

Рассмотрим процесс восстановления базы данных после сбоя с использованием журнала. При запуске система выполняет следующие шаги:

1. Читает журнал[1] в поисках информации о завершённых контрольных точках и корректной остановке экземпляра. Если экземпляр остановлен корректно, то файлы данных находятся в согласованном состоянии, и система может переходить к выполнению пользовательских запросов.
2. Если экземпляр остановлен аварийно, то система читает журнал начиная с последней завершённой контрольной точки и вносит изменения в данные, как если бы они были вызваны пользовательскими операциями.
3. В момент окончания повторного выполнения команд из журнала (используется также термин «накат», roll-forward) система может начинать исполнение пользовательских запросов.
4. Какие-то транзакции после восстановления оказываются незавершёнными. Система откатывает эти транзакции. Объём работ по откату транзакций зависит от особенностей конкретной платформы — так, например, Oracle записывает в журнал информацию об откате транзакций, а у PostgreSQL такой необходимости нет.

Устройство журнала

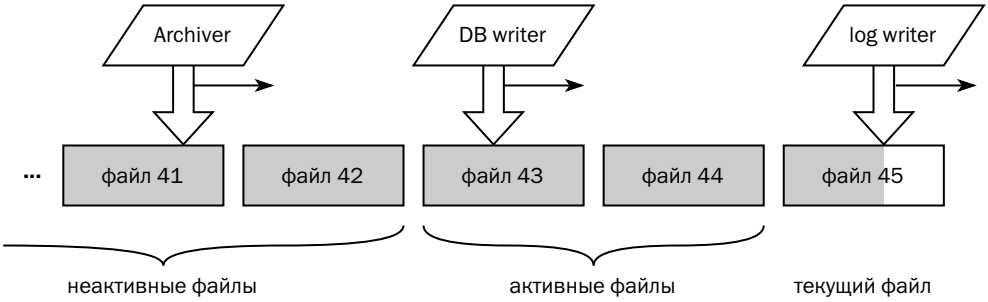
Журнал представляет собой последовательность файлов. После того как очередной файл заканчивается (достигает заданного максимального размера или принудительно закрывается), система создаёт следующий файл, помечает его как текущий и продолжает записывать в него журнальную информацию. Предыдущий файл помечается как активный, т. к. в случае сбоя сервера он понадобится для восстановления изменений.

После записи в основное хранилище фоновым процессом контрольной точки всех изменений, информация о которых помещена в очередной файл журнала, файл помечается как неактивный, и его удаление не повлияет на восстановление базы.

---

[1] В некоторых системах нет необходимости сканировать журнал целиком, т. к. система поддерживает файл с информацией о последней контрольной точке. Например, в PostgreSQL это файл pg_control.

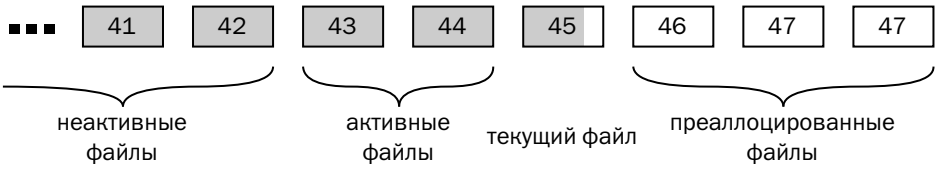
В подавляющем большинстве промышленных баз данных осуществляется резервное копирование журналов, и физически удалять журнал можно только после его резервного копирования. Специальный процесс архивирования копирует файлы журнала на внешний носитель.



На рисунке процесс-архиватор копирует только неактивные журнальные файлы, но в некоторых платформах он может копировать даже текущий файл, обгоняя процесс сохранения изменений в файлы данных и отставая от процесса записи журнала на считанные секунды.

Организация журнальных файлов отличается от платформы к платформе.

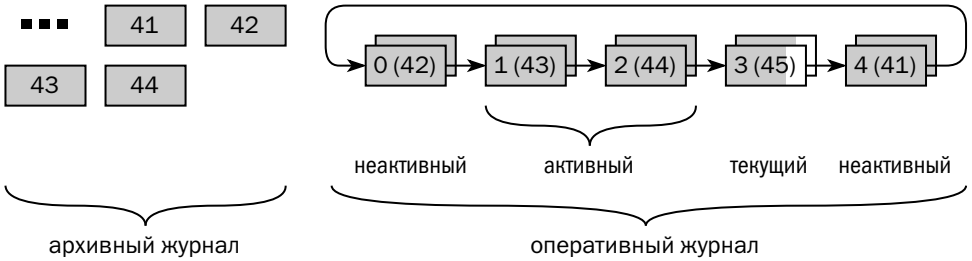
У таких платформ, как PostgreSQL, MongoDB и Cassandra, журнал представляет собой последовательность физических файлов. Система пишет данные журнала в первый файл, после его заполнения переходит ко второму и т. д. Поскольку создание файла — длительная процедура, система может создавать файлы заранее, а в некоторых случаях может переиспользовать существующие файлы.



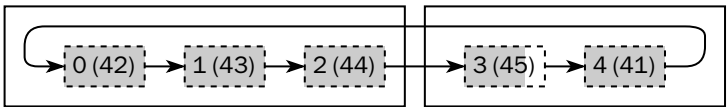
В Oracle есть фиксированный набор файлов, называемых файлами оперативного журнала (online redo log). Система использует эти файлы по принципу кольцевого буфера: когда закончится первый файл, переходит ко второму, после второго — к третьему, а когда закончится место в последнем файле, снова пишет в первый. За то время, пока запись пройдёт по кругу, все изменения должны быть записаны в базу данных, а сам файл журнала при необходимости скопирован в архив. Если «голова» догнала «хвост» и в файлах оперативного журнала не осталось свободного места, система перестаёт выполнять транзакции.

Архивные журналы (archive redo log) не являются частью базы данных и не требуются при восстановлении после сбоя экземпляра, но без них невозможно резервное копирование и восстановление из резервной копии.

В качестве оперативных журналов Oracle может использовать не только файлы, но и наборы файлов, дублирующих друг друга.



В Microsoft SQL Server журнал также представляет собой фиксированный набор файлов, причём этот файл может быть даже один. Всё журнальное пространство разбито на части, называемые «виртуальными файлами» (virtual files). Так же, как и Oracle, Microsoft SQL Server использует виртуальные файлы по принципу кольцевого буфера. Платформенные утилиты могут создавать резервную копию журнальных файлов, образуя цепочку архивных журналов (log chain). При переполнении файлов журнала Microsoft SQL Server может увеличить размер файла, если это разрешено настройками и на диске есть свободное место.



Db2 в зависимости от настроек может вести себя либо как Oracle, используя по кругу фиксированный набор файлов, но без возможности резервного копирования, либо как PostgreSQL, создавая для каждого нового журнального файла новый физический файл.

Структуры без журналирования

Практически все современные СУБД используют журналирование, однако есть и структуры, позволяющие обходиться без журналов.

LSM-деревья без журналирования применяются в аналитических платформах, которые выполняют относительно редкие, но длинные транзакции и могут

позволить себе потерять содержимое буфера в памяти. Примером такой системы может служить ClickHouse.

LSM-дерево устроено таким образом, что в нём нет физических изменений и удалений — данные только добавляются. По окончании записи данных на диск система отмечает в метаданных, что новые файлы теперь являются частью базы данных, и этот момент считается окончанием транзакции. В случае сбоя сервера в середине транзакции достаточно игнорировать файлы, созданные этой транзакцией.

Массовая загрузка данных без использования журнала существенно ускоряется, т. к., во-первых, объём записанных данных уменьшается вдвое, а во-вторых, записывать данные в таблицы, в отличие от журнала, можно несколькими параллельными процессами. Этой возможностью пользуются и платформы, где журналирование применяется. Например:

- Oracle поддерживает операцию прямой загрузки (direct path load), используемую утилитой SQL*Loader и командой `INSERT` с подсказкой[1] `APPEND`;
- PostgreSQL и Db2 позволяют отключить журналирование операций над конкретной таблицей на время загрузки данных;
- с Db2 поставляется утилита LOAD, позволяющая быстро загрузить данные без журналирования (`NONRECOVERABLE`).

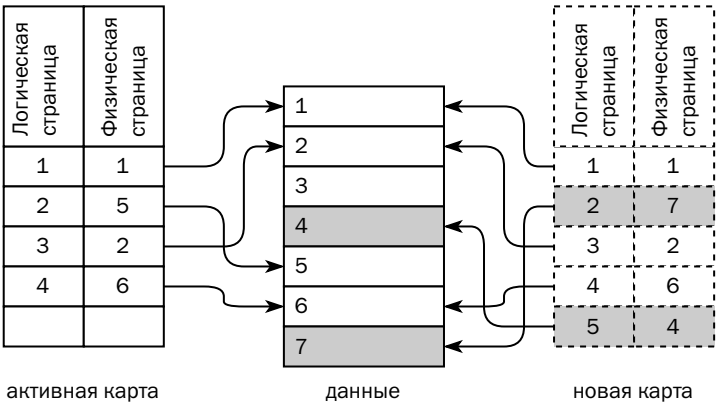
Загрузка без журналирования широко используется в аналитических системах. Однако прежде, чем воспользоваться такой заманчивой возможностью, необходимо представлять себе все её подводные камни:

- Если во время операции произойдёт сбой, то в нежурналируемой таблице PostgreSQL и Db2 будут потеряны все данные, включая загруженные ранее.
- В PostgreSQL данные нежурналируемой таблицы при сбое экземпляра будут потеряны, даже если за время работы экземпляра никакой записи в таблицу не было. Если же включить журналирование на нежурналируемой таблице, то все её данные будут скопированы в журнал.
- Данные, загруженные без журналирования, не будут реплицированы на резервный сервер.
- В Oracle и Db2 при успешном окончании транзакции данные будут доступны даже после сбоя сервера, а вот резервную копию (как минимум инкрементальную) надо будет обновить, иначе восстановить загруженные данные из резервной копии не получится.

[1] В профессиональной среде чаще употребляется термин «хинт» — калька с английского «hint». Подробнее о подсказках оптимизатору — в разделе 4.3 «Оптимизация запросов».

Ещё один способ обойтись без журналирования — использование **теневых страниц (shadow pages)**, которые можно применять при страничной организации памяти. Иногда этот механизм называют копированием при записи (copy-on-write).

Пусть у нас есть виртуальные страницы с данными и карта, которая ставит в соответствие физические страницы логическим страницам. При необходимости изменить данные система копирует изменяемые физические страницы в новое место и поддерживает в памяти копию карты, где логические страницы связаны с новыми физическими страницами вместо существующих.



Все изменения транзакция делает в новых («теневых») страницах, которые не видны другим транзакциям, а затем записывает изменённые страницы на диск. Завершением транзакции считается замена текущей карты на новую.

Подход с теневыми страницами чрезвычайно надёжен: при сбое сервера повреждение сохранённых данных невозможно, а при восстановлении откат незавершённых транзакций происходит мгновенно. Этот подход используется в таких системах, как CouchDB и TokyoCabinet.

Широкого распространения подход с теневыми страницами не получил из-за ряда существенных недостатков:

- каждая транзакция целиком перезаписывает изменённые страницы, что ведёт к большому объёму ввода-вывода;
- очень сложно организовать конкурентное обновление данных, чтобы одновременно выполняющиеся транзакции меняли данные согласованно;
- операция записи карты страниц на диск должна быть атомарной. В противном случае при сбое сервера в процессе перезаписи карты база данных будет повреждена безвозвратно.

Общий алгоритм выполнения транзакции

Подробно рассмотрев устройство журнала, давайте проследим работу транзакции по шагам. Итак, получив запрос на изменение данных, сервер выполняет следующие шаги:

1. Сохраняет текущее состояние базы данных на случай, если транзакцию придётся откатить. О том, где и как хранится это состояние, мы ещё поговорим, но так или иначе это сохранение отражается в журнале.
2. Вносит изменения в данные. Как правило, за время транзакции все изменения выполняются исключительно в памяти, а на диск они записываются асинхронно.
3. Записывает в журнал информацию, позволяющую повторить внесённые изменения.
4. Записывает в журнал маркер окончания транзакции и дожидается, пока журнал будет записан на диск.

Если по каким-то причинам принято решение откатить транзакцию, то между третьим и четвёртым шагами сервер выполняет ещё два действия:

- возвращает данные к исходному состоянию, пользуясь информацией, сохранённой на первом шаге;
- записывает в журнал информацию, позволяющую повторить возврат данных к исходному состоянию.

Поскольку окончанием транзакции считается атомарная операция записи маркера в журнал, можно сказать, что вся транзакция выполняется атомарно: нет маркера — нет транзакции. По этой же причине можно сказать, что журналирование обеспечивает и согласованность базы данных: перед тем как пометить транзакцию успешно завершённой, сервер имеет возможность проверить, все ли внутренние структуры обновлены и все ли ограничения целостности соблюдены[1].

Записав информацию о транзакции в журнал, сервер имеет возможность восстановить все изменения, сделанные в этой транзакции, даже если сбой произойдёт немедленно. Таким образом журналирование обеспечивает и надёжность хранения.

---

[1] Как правило, реляционные платформы проверяют ограничения немедленно после выполнения команды, не дожидаясь окончания транзакции. Однако Oracle, PostgreSQL и Microsoft SQL Server дают возможность отложить проверку ограничений до окончания транзакции. Механизм так и называется — отложенные проверки, deferred constraints.

Единственное свойство транзакции, которое не может обеспечить журнал, — изоляция. Для обеспечения изоляции применяются другие средства — блокировки и версионирование данных, о которых мы и поговорим в следующих разделах.

5.3. Блокировки

Блокировкой (lock) называется отметка о том, что объект используется транзакцией, и доступ других транзакций к этому объекту должен быть ограничен или вообще запрещён. Блокировки являются старейшим механизмом изоляции транзакций.

Впервые применение блокировок для изоляции транзакций было предложено в 1976 году инженером компании IBM, одним из создателей платформы System R и авторов языка SQL Капали Эсвараном (Kapali Eswaran). Теорема, сформулированная в его работе «Заметки о согласованности баз данных и о предикатных блокировках» («The Notions of Consistency and Predicate Locks in a Database System»), гласит, что транзакции в базе данных могут быть упорядочены, если каждая транзакция следует двухфазному протоколу блокировок.

На первой фазе транзакция блокирует объекты, затем изменяет их и на второй фазе снимает блокировки. Важно, что после того, как транзакция сняла хотя бы одну блокировку, она уже не может накладывать новые, поэтому во всех СУБД, реализующих двухфазный протокол[1], вторая фаза выполняется непосредственно перед завершением транзакции.

В работе Эсварана все блокировки одинаковы, а для описания подмножества блокируемых объектов используются логические выражения — предикаты. Практические реализации блокировок в современных платформах гораздо разнообразнее, и именно о них мы поговорим в этом разделе.

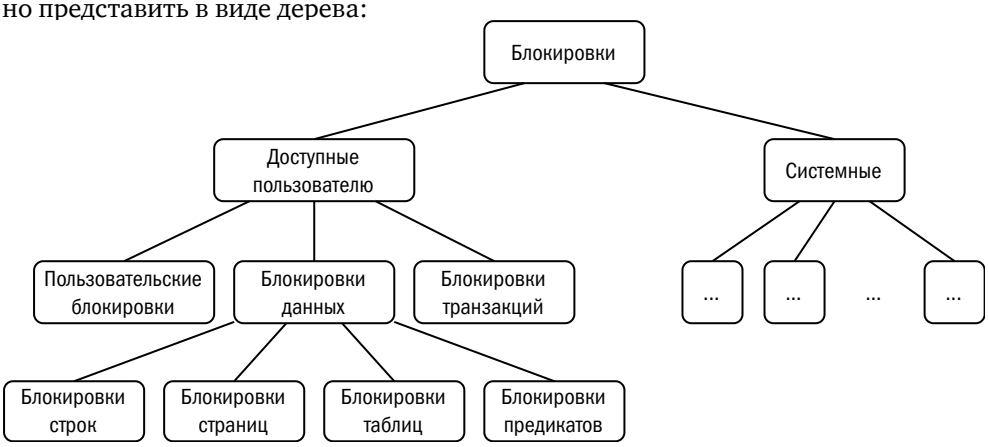
Классификация блокировок

Блокировки можно классифицировать:

- по области действия;
- по строгости — на исключительные и разделяемые;
- по логике реализации — на оптимистические и пессимистические.

[1] Двухфазный протокол блокировок (two-phase locking, 2PL) не имеет никакого отношения к двухфазному протоколу фиксации транзакций, 2PC (two-phase commit), о котором речь пойдёт в главе 8 «Изменение данных в распределённых системах».

При помощи механизма блокировок СУБД решают самые разные задачи, связанные с конкурентным доступом к ресурсам, и для разных ресурсов реализованы разные блокировки. Классификацию блокировок **по области действия** можно представить в виде дерева:



Системные блокировки защищают внутренние объекты, которыми оперирует СУБД, — файлы, табличные пространства, очереди, роли... В Oracle насчитывается около 300 типов системных блокировок. Для обозначения системных блокировок разные платформы используют разные термины — например, в Oracle это `IS_USER = NO`, а в PostgreSQL — блокировка типа `Object`.

Блокировки, доступные пользователю, возникают при обработке запросов.

Пользовательские блокировки (user-defined locks) представляют собой блокировки абстрактных объектов. Они могут использоваться для синхронизации исполнения хранимых процедур и по функциональности близки к мьютексам (mutex) в операционных системах.

Блокировки данных (DML[1] lock) защищают непосредственно пользовательские данные, являясь одним из средств изоляции транзакций, и именно об этих блокировках мы и будем говорить в дальнейшем.

Блокировки транзакций (transaction lock) — специальный вид блокировок, которые некоторые платформы генерируют для защиты данных. О них подробнее будет рассказано в подразделе, посвящённом реализации блокировок.

Основной вид блокировки — блокировка строки[2] (row lock), указывающая на то, что транзакция работает с этой строкой. Иногда случается, что транзакции

[1] DML = Data Manipulation Language, в реляционных платформах этим термином обозначают набор команд `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`.

[2] Или ключа (key lock) в нереляционных платформах.

блокируют слишком много строк, и экземпляр не может управлять таким количеством блокировок. В этом случае происходит повышение уровня блокировки (lock escalation), и вместо того чтобы блокировать много строк, экземпляр блокирует страницу, где находятся заблокированные строки (page lock), или даже всю секцию либо таблицу (table lock). При повышении уровня заблокированными оказываются не только те строки, с которыми работает транзакция, но и все остальные строки, находящиеся в той же странице или в той же таблице.

Блокировка таблицы требуется при изменении структуры этой таблицы, а также при любой операции над строками этой таблицы — чтобы транзакция была уверена, что за время её выполнения структура таблицы не изменилась.

Ряд аномалий, например фантомные чтения или несогласованную запись, невозможно предотвратить, используя только блокировки строк, поскольку эти аномалии возникают при создании параллельными транзакциями новых строк. Для их предотвращения при сериализации применяются блокировки предикатов (predicate lock), или, что то же самое, интервальные блокировки (gap lock). Вместо блокировки конкретных записей транзакция при чтении блокирует предикат — например, `SALARY > 1000`. Если конкурирующая транзакция попытается изменить (создать, удалить, обновить) строки, удовлетворяющие этому предикату, она будет вынуждена ждать снятия блокировки.

По строгости блокировки делятся на исключительные (exclusive) и разделяемые[1] (shared). Первые подразумевают, что к объекту не может обращаться никто, кроме транзакции, установившей блокировку, а вторые — что к объекту могут обращаться и другие транзакции, которым нужна разделяемая блокировка, но не могут обращаться транзакции, требующие исключительного доступа. Матрица совместимости блокировок выглядит так:

	Разделяемая	**Исключительная**
Разделяемая	+	–
Исключительная	–	–

Теорема о корректности двухфазного протокола блокировок доказана только для исключительных блокировок, однако автор статьи считает, что доказательство теоремы очевидным образом обобщается на случай использования и исключительных, и разделяемых блокировок.

[1] Изредка используется термин «совместные».

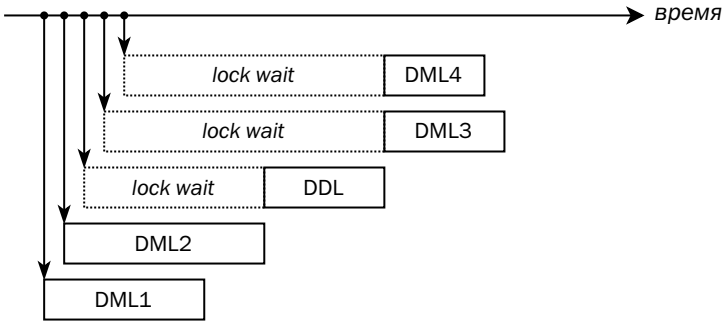
Если транзакция читает строку, то для низших уровней изоляции (read uncommitted, read committed, cursor stability) ей вообще не нужна никакая блокировка. Но если транзакция желает избежать неповторяющегося чтения, то ей может понадобиться наложить на прочитанную строку разделяемую блокировку, чтобы до завершения этой транзакции никто не смог изменить данные в прочитанной ей строке.

Если транзакция изменяет данные в строке, то ей необходима исключительная блокировка на эту строку, чтобы никто до завершения транзакции не смог не только изменить данные, но и прочитать неподтверждённые изменения.

Платформы, где изоляция транзакций обеспечивалась только блокировками, показывали крайне низкую производительность при конкурентном доступе из-за многочисленных ожиданий, поэтому современные платформы вместе с блокировками обеспечивают версионирование данных, о котором речь пойдёт в следующем разделе.

Вместе с блокировкой изменяемых или читаемых данных транзакциям необходима разделяемая блокировка на таблицу для уверенности, что за время выполнения транзакции структура таблицы не изменилась и все запросы к этой таблице по-прежнему корректны.

Операции изменения структуры таблицы (DDL, data definition language) требуют исключительной блокировки таблицы, и во время их исполнения никакие операции над данными в таблице невозможны. Именно поэтому выполнение DDL крайне не рекомендуется в периоды высокой нагрузки на базу данных. Пауза в работе системы может быть гораздо длиннее, чем время выполнения DDL-команды, поскольку команда сначала должна дождаться снятия всех ранее наложенных блокировок и только затем будет исполнена. Все команды, запущенные после DDL-операции, встанут за ней в очередь и будут ждать её окончания, вместо того чтобы «быстренько проскочить»:



Влияние блокировок на производительность обработки данных настолько сильно, что разработчики прилагают немалые усилия, чтобы уменьшить количество несовместимых блокировок.

Например, Microsoft SQL Server и Db2, кроме исключительной и разделяемой блокировок строк, поддерживает промежуточный тип блокировок — UPDATE.

Такую блокировку может наложить на строку транзакция, которая собирается менять данные. Непосредственно перед изменением блокировка UPDATE должна быть преобразована в исключительную, но пока это не произошло, другие транзакции могут читать строку, накладывая на неё разделяемую блокировку.

В PostgreSQL пишущая транзакция вместо исключительной блокировки строки (FOR UPDATE) может наложить менее строгую блокировку FOR NO KEY UPDATE, не допускающую удаления строки и изменения её уникальных ключей. В свою очередь, если читающей транзакции необходима блокировка строки и важно не всё её содержимое, а просто факт наличия (или только ключевые поля), то вместо разделяемой блокировки (FOR SHARE) она может наложить менее строгую блокировку FOR KEY SHARE. Блокировки FOR KEY SHARE и FOR NO KEY UPDATE взаимно совместимы, то есть чтение ключа и обновление неключевой информации не мешают друг другу.

Поскольку конкуренция между транзакциями за таблицу гораздо сильнее, чем за отдельную строку, в современных платформах реализовано не два режима блокировки таблиц, а гораздо больше — как правило, 6–8, и каждая команда накладывает минимально возможную блокировку. Полный список режимов блокировки таблиц, матрицу совместимости и карту соответствия команд и режимов блокировки можно найти в фирменной документации.

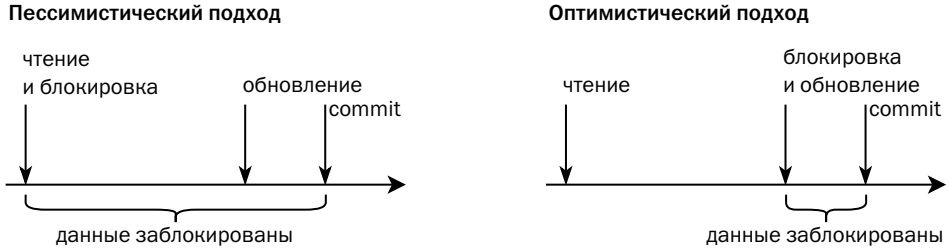
Наконец, **по логике реализации** блокировки делятся на оптимистические и пессимистические.

Пессимистический подход предполагает, что конкуренция за строку высока, и кто-то попытается её изменить в ближайшее время. Транзакция при таком подходе блокирует данные в тот момент, когда понимает, что их надо будет менять. При оптимистическом подходе, напротив, считается, что строка, скорее всего, не изменится. Оптимистическая транзакция не блокирует строку до тех пор, пока не настанет непосредственно момент изменения.

У каждого из подходов есть преимущества и недостатки:

	Пессимистический	**Оптимистический**
Неизменность данных	Гарантирована	Требуется проверка

	Пессимистический	Оптимистический
Риск отказа фиксации транзакции после проведения изменений	Нет	Есть
Возможность блокировки лишних данных	Есть	Нет
Продолжительность блокировки	Длинная	Короткая
Обращение к механизмам блокирования	Два раза — протокол 2PL	Один раз



Важно понимать разницу между командой изменения данных и бизнес-транзакцией.

В подавляющем большинстве современных платформ каждое обновление строки представляет собой «пессимистическую транзакцию» в миниатюре, то есть строка блокируется в момент изменения и остаётся заблокированной до конца транзакции (подтверждения или отмены). Платформ, использующих оптимистические механизмы блокирования, немного — например, Apache Ignite/GridGain, YDB, TiDB, FoundationDB. Эти платформы накапливают изменения в буфере транзакции и в момент фиксации проверяют, не были ли изменённые текущей транзакцией строки изменены другими транзакциями.

Для реализации бизнес-транзакций, которые могут состоять из нескольких команд чтения и изменения данных, неопытные программисты используют уровень изоляции, предоставляемый платформой по умолчанию (чаще всего — read committed), читают данные, выполняют вычисления и записывают в базу данных результат. Зачастую, особенно в системах, связанных с обслуживанием людей (то есть с относительно низкой конкуренцией за каждую конкретную строку), такие программы работают, как задумано. Однако порой такой подход приводит к непонятным и трудноуловимым ошибкам — либо к странным результатам, вычисленным на основе некорректно прочитанных данных («потеря изменений»), либо к потере результатов вычислений («несогласованная запись»).

Для корректной работы с данными необходимо использовать один из трёх подходов:

1. Выставлять транзакциям уровень изоляции serializable. Платформа берёт на себя обеспечение целостности данных, накладывая либо построчные разделяемые, либо интервальные блокировки на прочитанные данные. Производительность базы данных при этом снижается за счёт увеличения конкуренции между транзакциями.
2. Вручную блокировать прочитанные строки, содержимое которых влияет на результат вычисления, командой `SELECT` с фразой `FOR UPDATE`[1], реализуя тем самым пессимистическую логику. Этот вариант также довольно сильно влияет на производительность базы данных, поэтому явную ручную блокировку рекомендуется использовать только при высокой конкуренции за одни и те же данные.
3. Проверять неизменность данных при обновлении, что соответствует оптимистической логике. Такой подход при высокой конкуренции за одни и те же данные ещё хуже, чем пессимистические блокировки. Однако если обычно транзакции работают с разными данными, как при обслуживании клиентов, то этот подход практически не влияет на производительность, позволяя избежать несогласованной записи и потерянных изменений.

Проверка неизменности данных состоит из следующих шагов:

- В таблице заводится колонка «версия», значение в которой увеличивается при каждом обновлении строки.
- Считывая строку, транзакция обязательно считывает её версию.
- Обновляя строку, транзакция проверяет, что обновляет именно ту версию, которую считала, добавляя в условие обновления к значению ключа номер версии, и в этой же команде обновляет номер версии. Пусть обновление данных выполняется командой `update employee set salary = 1200 where id=42`. Команда с проверкой будет чуть длиннее: `update employee set salary = 1200, _VERSION = 89 where id=42 AND _VERSION=88`. В момент успешного выполнения команды строка блокируется.
- Если обновление не получилось, значит, обновляемая запись была либо удалена (и тогда в таблице не будет строки с ключом 42), либо изменена (и тогда у строки изменится номер версии, и условие в `where` также не найдёт ни одной строки). В этом случае бизнес-транзакция должна забыть все ре-

[1] Точный синтаксис команды зависит от платформы.

зультаты, откатить транзакцию в базе данных и повторить чтение, расчёты и попытку записи.

Управление блокировками

Очевидный подход к управлению блокировками строк — завести структуру в памяти сервера БД, в которой каждая блокировка будет представлена записью. Такой подход используется в Microsoft SQL Server и Db2.

У этого подхода есть несомненное преимущество: из монолитной платформы можно сделать распределённую, почти не меняя архитектуру, а общий менеджер блокировок вынести на выделенный узел. Так, например, устроен кластер Db2 Parallel Sysplex — кластерная версия Db2 for z/OS и его версия для Linux под торговой маркой IBM PureData System for Transactions.

С другой стороны, структура, описывающая блокировку, получается весьма объёмной — она должна содержать идентификатор заблокированной строки, строгость блокировки и идентификатор транзакции, наложившей блокировку. В Microsoft SQL Server такая структура занимает 96 байт, а в Db2 — около 500 байт. По умолчанию экземпляр Db2 способен поддерживать до 10 000 блокировок, в то время как высоконагруженная система может одновременно обновлять несколько сотен тысяч строк.

Если требуется заблокировать больше строк, чем имеется слотов для блокировки, платформа повышает уровень блокировки (lock escalation), то есть захватывает более крупный объект, содержащий в себе несколько блокируемых объектов, с одновременным освобождением «мелких» блокировок.

Платформы со страничной организацией памяти от блокировки записей могут переходить не к блокировке таблицы, а к блокировкам страниц. Не следует путать блокировку страницы, обеспечивающую изоляцию данных (page lock), с кратковременной блокировкой страницы в буферном кеше для получения её согласованного образа (latch или spin-lock).

При повышении уровня освобождаются ресурсы менеджера блокировок, но строки, которые находятся в заблокированных страницах, становятся недоступны (или ограниченно доступны) другим транзакциям, что увеличивает конкуренцию. Если транзакция блокирует так много строк, что даже блокировки страниц не помещаются в память, платформа может повысить уровень блокировки до секции или даже таблицы.

MySQL[1] использует более компактное представление блокировок: у каждой транзакции есть битовые карты, в которых отмечаются заблокированные строки. Такой подход требует от трёх до восьми бит на каждую заблокированную строку, причём если строки физически находятся рядом друг с другом, то представление блокировок получается более компактным, чем для разреженной выборки. MySQL никогда не повышает уровень блокировок. Тем не менее возможны ситуации, когда оперативной памяти для хранения блокировок не хватает.

Что же делает транзакция, если оказывается, что строка (или другой объект), которая ей требуется, заблокирована другой транзакцией в несовместимом режиме?

Для начала транзакция может предположить, что конкурирующая транзакция быстро закончится, и несколько раз попытаться повторно захватить блокировку. Если повторная блокировка удалась, то транзакция продолжается, в противном случае она приостанавливается и встаёт в очередь, ожидающую снятия блокировки.

Возможна ситуация, когда первая транзакция ожидает освобождения блокировки, наложенной второй транзакцией, а вторая в свою очередь ожидает освобождения блокировки, наложенной первой, и в результате ни одна из транзакций не может выполняться дальше. Такая ситуация называется **взаимоблокировкой (deadlock)**:

Транзакция 0xDEAD	Транзакция 0xBEEF
`update client set shoe_size = 8.5 where id=42`	
	`update client set eye_color = 'blue' where id=88`
`update client set shoe_size = 11 where id=88`	
	`update client set eye_color = 'green' where id=42`

При попытке выполнить вторую команду транзакция 0xDEAD останавливается в ожидании, пока транзакция 0xBEEF не отпустит блокировку строки с ключом 88. Вместо этого транзакция 0xBEEF пытается обновить строку с ключом 42

[1] С движком InnoDB.

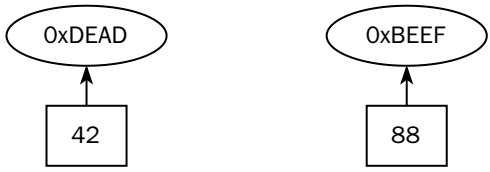
и останавливается в ожидании окончания транзакции 0xDEAD. Согласно классическому определению, взаимоблокировка возникает, когда выполнены четыре условия:

1) **взаимоисключение**, то есть ситуация, когда несколько процессов пытаются одновременно использовать ресурс, который может использовать только один процесс.
2) **удержание и ожидание**, то есть ситуация, когда процесс удерживает какой-то ресурс и одновременно пытается получить доступ к другому;
3) **отсутствие вытеснения**, то есть невозможность «отобрать» ресурс у удерживающего процесса;
4) **циклическая зависимость** между процессами.

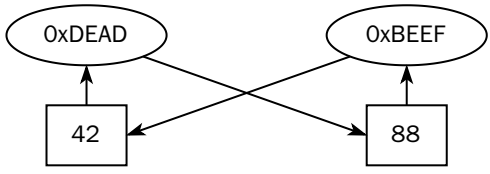
Когда не выполнено хотя бы одно из этих условий, взаимоблокировки нет.

Разрешение взаимоблокировок в базах данных основано на поиске циклических зависимостей (выполнение условия 4) и принудительном завершении одной из транзакций (нарушение условия 3). При этом условие 1 выполняется для любой блокировки, а если транзакция пытается получить доступ к нескольким объектам, то для неё выполняется и условие 2.

СУБД либо периодически, либо при превышении одной из транзакций порогового времени ожидания строит граф ожиданий, вершинами в котором являются транзакции и строки, а рёбрами — блокировки. После того как в описанном выше примере каждая из транзакций выполнит свою первую команду, граф ожиданий примет такой вид:



Граф направленный, и если строка заблокирована транзакцией, то ребро выходит из строки и входит в транзакцию. Если же транзакция ожидает доступа к строке, то ребро, наоборот, выходит из транзакции и входит в строку. После выполнения каждой из транзакций своей второй команды граф ожиданий примет вид:



Появление направленного цикла в этом графе, то есть выполнение условия 4, означает наличие взаимоблокировки. На рисунке выше изображена взаимоблокировка двух транзакций, но цикл может включать в себя три и более транзакций.

Обнаружив взаимоблокировку, СУБД должна освободить какой-то ресурс, то есть нарушить условие 3. Для этого какая-то из транзакций, входящих в цикл, должна быть отменена. Как правило, отменяется последняя транзакция, замкнувшая цикл (в нашем примере — транзакция 0xBEEF), но рассчитывать на это нельзя. Например, Db2 отменит произвольную транзакцию.

Другой механизм управления блокировками — хранение отметок о блокировке непосредственно в области данных, то есть в страницах. Разные варианты такого подхода используют Oracle и PostgreSQL.

У такого подхода есть ряд преимуществ:

- Количество блокировок, накладываемых одной транзакцией, не ограничено ничем. Транзакция может заблокировать все записи в таблице, и при этом не возникнет необходимости повышения уровня.
- После завершения транзакции нет необходимости снимать блокировки — они могут быть сняты позже транзакциями, которые будут вносить в данные последующие изменения.

Преимущества подхода Oracle и PostgreSQL в полной мере реализуются в системах с ограниченным объёмом оперативной памяти, а на современных серверах, где памяти достаточно, эти преимущества уже не так заметны.

Физическая реализация блокировок у Oracle и PostgreSQL достаточно сильно отличается, и эти отличия будут подробно рассмотрены в разделе, посвящённом версионированию данных.

К недостаткам хранения блокировок рядом с данными можно отнести невозможность накладывать блокировки предикатов. У PostgreSQL есть механизм блокировки предикатов в памяти, а у Oracle таких блокировок вообще нет, из-за чего платформа не способна обеспечить уровень изоляции serializable, ограничиваясь уровнем snapshot isolation[1].

Обнаружив, что строка, которую транзакция собирается блокировать, уже заблокирована в несовместимом режиме, система создаёт блокировку типа «транзакция», которая располагается в оперативной памяти. В отличие от ранее описанного подхода, таких записей в памяти столько, сколько транзакций ожидает

[1] Который тем не менее в документации и диалекте SQL называется `serializable`.

завершения другой транзакции. Заблокированная транзакция зависит от заблокировавшей и будет продолжена после её окончания.

Рассмотрим ещё раз пример с двумя транзакциями выше. К тому моменту, как каждая транзакция выполнила свою первую команду, таблица блокировок в памяти пуста!

После выполнения первой транзакцией второй команды в памяти возникает блокировка:

И наконец, после выполнения второй транзакцией второй команды возникает вторая блокировка:

Взаимоблокировка определяется так же, как и в предыдущем случае, по наличию циклов, но сам цикл получается минимум вдвое короче, поскольку в графе нет вершин, соответствующих строкам, а есть только вершины, соответствующие транзакциям.

Механизм хранения информации о блокировках рядом с блокированными данными принципиально непригоден для хранилищ с неизменяемыми данными на основе LSM-деревьев.

Конкурентные транзакции без блокировок

При сериализации транзакций результат их параллельного исполнения эквивалентен результату их какого-либо последовательного исполнения.

В конце 2000-х годов нескольким разработчикам пришла в голову одна и та же идея: а что, если исполнять транзакции действительно последовательно? Так устроены Redis (2009), Tarantool (2010), VoltDB (2010).

Если все изменения выполняются последовательно одним потоком, то блокировки становятся не нужны. Выполнение транзакций за счёт отсутствия накладных расходов на блокировки кратно ускоряется.

Однако использование однопоточного сервера накладывает на архитектуру платформы ряд ограничений:

- Платформа должна хранить все данные в памяти. Производительность системы, читающей в один поток данные с диска, получается неприемлемо низкой.
- Интерактивные транзакции невозможны. Другими словами, транзакция представляет собой одну команду, или хранимую процедуру, или же набор команд, между выполнением которых система не возвращает управление клиенту[1].
- Производительность одного экземпляра платформы ограничена скоростью процессора, поэтому платформа должна быть распределённой.
- Алгоритм распределённых транзакций должен быть неблокирующим, то есть единственный способ взаимодействия нескольких экземпляров — сага[2].

5.4. Версионирование данных

Ещё один важнейший механизм обеспечения изоляции транзакций — версионирование (multiversion concurrency control, MVCC).

Обработка данных без версионирования

Рассмотрим транзакцию, изменяющую строку с ключом 1024. После того как система получила команду на изменение строки, она должна:

- заблокировать изменяемую строку;
- сохранить новое состояние строки;
- сохранить старое состояние строки на случай отмены транзакции.

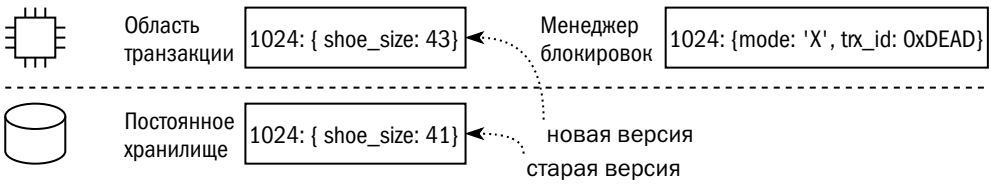
Новое состояние записи сохраняется в области памяти, принадлежащей данной транзакции.

Такая архитектура была реализована в ранних версиях Microsoft SQL Server. Из современных платформ без версионирования обходятся, например, Db2, NonStop SQL, YDB и Apache Ignite/GridGain.

---

[1] В Tarantool 2.6 и старше интерактивные транзакции реализованы за счёт версионириования данных и оптимистических блокировок.

[2] Подробнее о сагах — в главе 8 «Изменение данных в распределённых системах».



В случае отмены транзакции память транзакции просто очищается, а при подтверждении новое значение записывается на место старого.

У описанной схемы есть весьма серьёзные недостатки:

- Всё время между выполнением команды изменения и подтверждением транзакции изменяемая строка остаётся заблокированной в исключительном режиме, что останавливает конкурирующие транзакции, которые пытаются прочитать изменяемую строку. Для решения проблемы платформа либо разрешает грязное чтение (ранние версии Db2), либо обеспечивает механизм оптимистической блокировки (Apache Ignite/GridGain).
- Если в момент подтверждения транзакции произойдёт сбой, это приведёт к остановке экземпляра и восстановлению данных из журнала транзакций: после начала записи изменений старое состояние данных безвозвратно утрачено.
- Если транзакция запросила уровень изоляции, исключающий неповторяющееся чтение (то есть любой, кроме read uncommitted, read committed и cursor stability), то при попытке прочитать строки, изменённые после её начала параллельной транзакцией, она неизбежно будет прервана.
- Расход оперативной памяти и время подтверждения транзакции зависят от объёма изменённых данных.

При использовании транзакционных платформ без версионирования данных не рекомендуется выполнять в них сколько-нибудь продолжительные транзакции — как читающие, так и пишущие.

Аналитические платформы, для которых не характерна высокая конкуренция между транзакциями, могут обходиться без механизма версионирования. К таким платформам относятся, например, Teradata, Vertica, ClickHouse.

Версионирование в страничных хранилищах

Версионирование данных в страничных хранилищах основано на одной из двух идей:

1) новая версия данных записывается поверх старой, а старая какое-то время сохраняется в специальной области;
2) новая версия данных записывается рядом со старой, и старые версии удаляются фоновыми процессами.

Рассмотрим оба способа подробнее.

Сохранение старой версии в специальной области было впервые реализовано в 1984 году в четвёртом релизе Oracle, а современный вид схема приобрела в 1988 году в шестом релизе. Позже похожие схемы были реализованы в MySQL и Microsoft SQL Server.

При попытке изменить данные транзакция:

- накладывает на изменяемые данные исключительную блокировку;
- сохраняет в сегменте отката (rollback segment[1]) текущую версию данных;
- записывает новую версию данных поверх текущей.

Для фиксации транзакции достаточно просто удалить идентификатор транзакции из списка активных транзакций в памяти.

Старые версии строк в сегменте отката привязаны к конкретной транзакции, и когда место для старых версий заканчивается, система может удалять ненужные версии, освобождая место для информации отката новых транзакций.

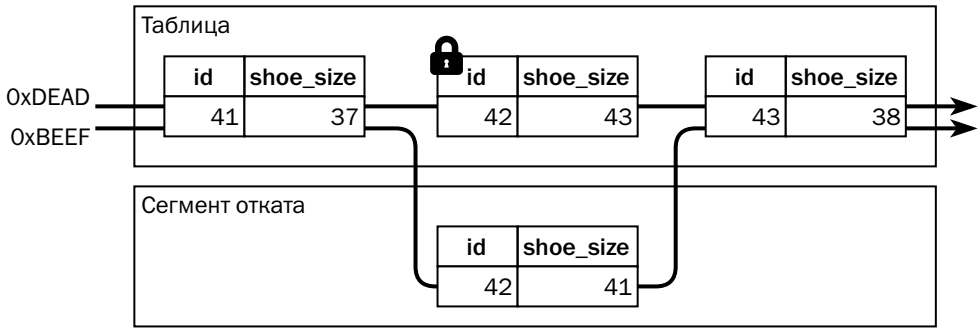
При отмене транзакции информация из сегмента отката переписывается обратно в сегмент данных. Таким образом, откат транзакции может занимать столько же времени, сколько занимала сама транзакция. Поскольку в промышленных базах данных подавляющее большинство транзакций завершаются успешно, этот недостаток можно считать несущественным.

Важно, что в журнал записывается информация как об изменениях данных, так и о записи в сегмент отката. Это необходимо, чтобы после сбоя сервера система могла восстановить содержимое сегментов отката и корректно отменить незавершённые транзакции. К сожалению, такое восстановление может занимать продолжительное время, поскольку системе необходимо откатить все транзакции, которые были активны в момент отказа сервера.

Если читающая транзакция видит, что требуемая строка заблокирована, то она делает вывод, что там находятся изменения, внесённые незавершённой транзак-

[1] Термин rollback segment используется в Oracle до релиза 9i включительно и в MySQL. В релизах 10g и старше Oracle изменил механизм управления сегментами отката, и термин rollback был заменён на undo. Конкурентное исполнение транзакций действительно улучшилось, хотя концептуально ничего не изменилось. Microsoft SQL Server сохраняет старые версии данных в системной базе данных tempdb.

цией, ищет сегмент отката, приписанный этой незавершённой транзакции, и читает предыдущую версию записи оттуда:



Если для старых версий данных выделить достаточно большое пространство, то можно выполнять запросы к базе данных по состоянию на момент в прошлом. На этом принципе основана технология «воспоминаний» (flashback) в Oracle.

В отличие от платформ, хранящих информацию о блокировках в выделенной области оперативной памяти, у успешной транзакции в Oracle нет необходимости снимать все блокировки. Транзакция снимает блокировки только с тех строк, которые находятся в буферах в оперативной памяти[1]. Если какой-то транзакции в будущем понадобится заблокированная строка, она попытается найти активный сегмент отката, связанный с блокирующей транзакцией. Поскольку блокирующая транзакция завершена, сегмент отката, связанный с ней, или не активен, или затёрт последующими транзакциями. Не найдя активного сегмента отката, новая транзакция понимает, что блокирующая транзакция закончена, и сама снимает блокировку, даже если она только читает данные[2].

Альтернативный подход к версионированию заключается в **сохранении новых версий данных рядом со старыми**. Этот подход впервые был применён в релизе 6.5 СУБД PostgreSQL, увидевшем свет в 1999 году, а позднее был реализован в Firebird.

PostgreSQL хранит не строки, а версии строк, то есть данные хранятся в таблице столько раз, сколько они были изменены. Чтобы транзакция могла понять,

---

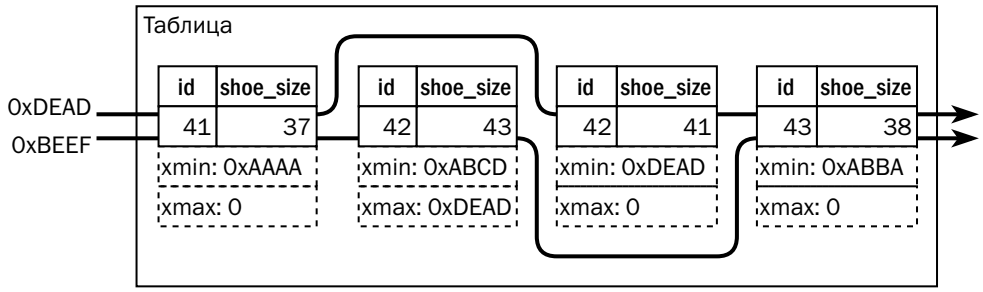
[1] Если этих буферов больше, чем 10 % буферного кеша, то блокировки не снимаются даже в оставшихся буферах.

[2] С этим механизмом связан эффект «отложенной очистки страниц» (delayed block cleanup) в Oracle, когда читающие транзакции порождают массу изменённых страниц, которые должны быть заново записаны на диск.

какая версия актуальна, и можно ли её изменять, или она заблокирована, в заголовке каждой версии содержатся следующие поля:

- `xmin` — идентификатор транзакции, создавшей эту версию. Версии создаются командами `INSERT` и `UPDATE`, и пишущая транзакция записывает в `xmin` свой идентификатор;
- `xmax` — идентификатор транзакции, в результате которой данная версия перестала существовать. Версии прекращают своё существование после выполнения `DELETE` или `UPDATE`, записывающих свои идентификаторы в `xmax`;
- флаги, показывающий статус транзакций `xmin` и `xmax`. Каждый флаг может принимать значения «неизвестно», «подтверждена» или «отменена».

Все версии строки объединены в односвязный список, где каждая предыдущая версия указывает на следующую, а самая свежая указывает сама на себя[1]. Таким образом любая транзакция может найти актуальную версию строки. У актуальной версии `xmax` равен нулю.



При завершении транзакции система удаляет её идентификатор из списка активных транзакций и записывает её исход (подтверждение или отмену) в специальную структуру под названием CLOG[2].

Для поиска актуального состояния строки транзакция находит её последнюю версию и анализирует состояние флагов:

	xmin_status	xmax_status	Действия и выводы
1	committed	aborted	Последняя версия актуальна, строка не заблокирована
2	committed	committed	Строка удалена

[1] Как правило, транзакция читает либо все версии, либо сразу актуальную версию, итерация по списку используется в специальных случаях.

[2] В некоторых источниках называется XACT — по названию каталога, где хранится структура.

	xmin_status	xmax_status	Действия и выводы
3	aborted	aborted	Последняя версия не актуальна, т. к. транзакция, создавшая эту версию, была отменена. Актуальной является предпоследняя версия (или, если её нет, строка вообще не существует). Строка не заблокирована
4	unknown	aborted	Транзакция ищет xmin в списке активных транзакций. Если нашла, значит, строка заблокирована. Если предыдущая версия есть, значит, она актуальна, если нет, значит, строки ещё нет. Если не нашла, то ищет исход транзакции xmin в CLOG, заполняет флаг[1] и действует в соответствии с п. 1 или 3
5	committed	unknown	Транзакция ищет xmax в списке активных транзакций. Если нашла, значит, строка удаляется (или изменяется) параллельной транзакцией, но пока не удалена (не изменена), а заблокирована, то есть именно эта версия является актуальной. В случае выполняющейся команды UPDATE такое состояние будет у предпоследней версии строки, и именно она будет актуальной. Если не нашла, то ищет исход транзакции xmax в CLOG, заполняет флаг и действует в соответствии с п. 1 или 2

Поскольку номера транзакций возрастают, транзакция может прочитать не только актуальную версию строки, но и ту версию, которая была актуальна на момент начала команды (при уровне изоляции read committed) или начала транзакции (при уровне изоляции repeatable read или serializable) — так обеспечивается согласованность данных.

Ещё один источник накладных расходов на поддержку многоверсионности в PostgreSQL — индексы. Индексная запись указывает на определённую версию строки, и когда строка меняется, индекс тоже должен меняться, даже если изменённые поля не входят в индекс. Разумеется, есть ряд механизмов, оптимизирующих такое обновление.

За счёт хранения всех версий записей файлы данных PostgreSQL раздуваются, поэтому периодически запускается процесс очистки (vacuum), удаляющий из файлов данных ненужные версии строк.

Таким образом, оба описанных подхода эффективно решают задачу изоляции транзакций, но **оба подхода сопряжены с накладными расходами**. Разница представлена в таблице ниже:

[1] Что, как и в Oracle, приводит к ситуации, когда читающая транзакция физически меняет данные.

	Сегменты отката (подход Oracle)	Версии в области данных (подход PostgreSQL)
Объём журнала	– Увеличенный (undo и данные)	+ Равен объёму изменений
Подтверждение транзакции	+ Мгновенное	+ Мгновенное
Отмена транзакции	– Долгая	+ Мгновенная
Хранение информации об исходе транзакций	+ Нет	– Необходимо (CLOG)
Объём хранимых версий	– Ограничен	+ Не ограничен
Обновление индексов	+ Не требуется	– Требуется
Периодическое сжатие таблиц	+ Не требуется	– Требуется
«Изменение» данных читающими транзакциями	– Возможно	– Возможно

Оба подхода на практике показали свою жизнеспособность, однако автору всё же больше по душе подход с выделенными сегментами отката.

Версионирование в хранилищах на основе LSM-деревьев

Поскольку в LSM-деревьях при записи изменённых данных предыдущее состояние не уничтожается, для обеспечения многоверсионности достаточно просто добавить к ключу временну́ю метку транзакции, изменившую этот ключ.

Рассмотрим механизм версионирования на примере CockroachDB.

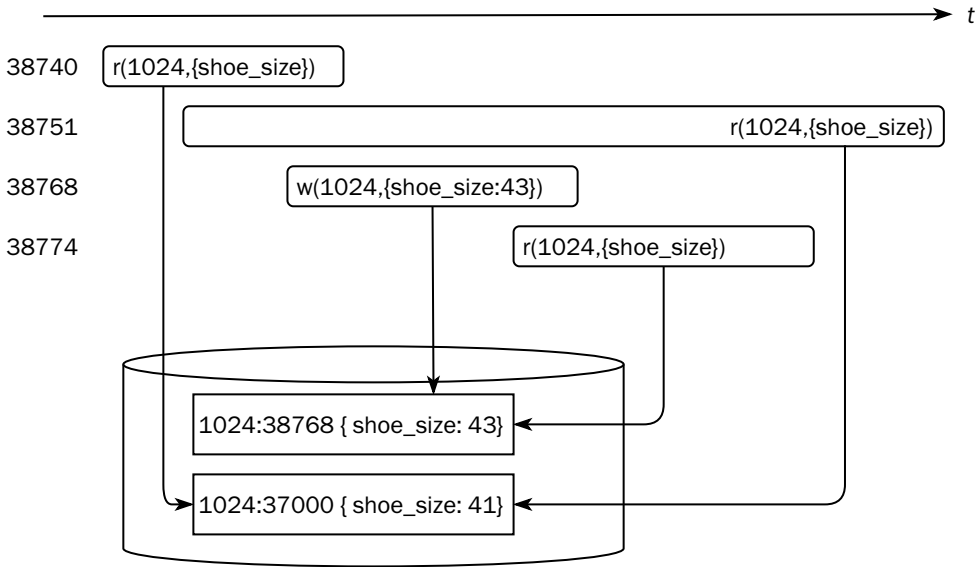
Когда транзакция, в том числе читающая, начинается, сервер приписывает ей временну́ю метку. Временная метка представляет собой комбинацию из показаний часов реального времени и «логических часов»[1].

При записи данных вместе с ключом записывается и временна́я метка транзакции.

Если временна́я метка читающей транзакции больше, чем временна́я метка ключа, то чтение происходит как обычно. В противном случае читающая транзакция игнорирует найденный ключ и продолжает чтение на следующих уровнях LSM-дерева. На рисунке ниже транзакция с временно́й меткой 38751 пытается

[1] Механизм, применяемый для генерации временных меток в CockroachDB, называется Hybrid Logical Clock (HLC) и описан в статье группы исследователей из Мичиганского университета и Университета Буффало «Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases».

прочесть данные с ключом 42. Обнаружив, что данные изменены после начала транзакции параллельной транзакцией с меткой 38768, она продолжает поиск ключа и находит предыдущую версию с временно́й меткой 37000.



Поскольку откат транзакции, результат которой записан в неизменяемую структуру данных, невозможен, платформа с хранилищем на основе LSM-дерева до подтверждения вынуждена сохранять планируемые изменения в памяти, что накладывает ограничения на максимальный объём изменений в транзакции.

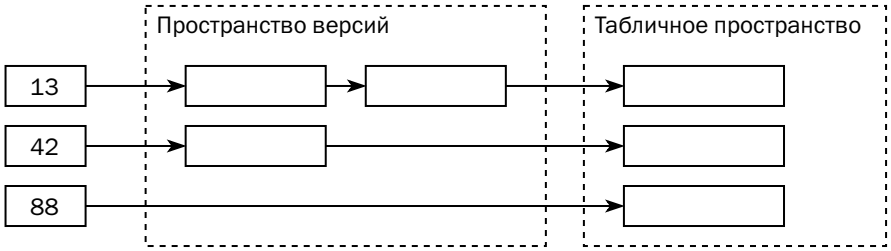
В CockroachDB механизм временны́х меток служит не только для версионирования данных, но и для упорядочивания (сериализации) транзакций.

Сервер поддерживает т. н. «кеш временны́х меток»: каждый раз, когда транзакция читает какие-то данные, она обновляет значение временно́й метки в кеше для каждого прочитанного ключа. Прежде чем изменить данные, пишущая транзакция должна убедиться, что метки чтения всех изменяемых ключей меньше, чем метка самой транзакции. Так, например, на рисунке выше к моменту попытки изменения объекта с ключом 1024 максимальная метка чтения для него будет 38740, то есть меньше, чем временна́я метка пишущей транзакции. Если же окажется, что данные были прочитаны после начала пишущей транзакции, то система пытается сдвинуть временну́ю метку пишущей транзакции вперёд. В некоторых случаях это может приводить к перезапуску транзакции.

Версионирование в СУБД в памяти

В отличие от дискового пространства, разделённого на файлы, деление оперативной памяти на «области» достаточно условно: оно требуется только для того, чтобы оптимизировать распределение памяти.

Так, например, в SAP HANA память разделена на табличное пространство (table space) и пространство версий (version space). Команда `INSERT`, создавая новые строки, сохраняет их непосредственно в табличное пространство, а вот команда `UPDATE`, обновляя данные, сохраняет новые версии строк в пространство версий. В памяти системы поддерживается хеш-таблица, где для каждого ключа хранится указатель на текущую версию строки. Текущая версия, в свою очередь, является началом односвязного списка, в котором содержатся все версии строки.



Сборщик мусора (garbage collector) удаляет из памяти неактуальные версии строк, перенося новые версии из пространства версий в табличное пространство. В простейшем случае сборка мусора основана на временны́х метках: версия удаляется, когда её временна́я метка становится меньше, чем временна́я метка самой старой из исполняемых в системе транзакций. Однако такой подход ведёт к росту потребления памяти, поскольку удаляет не все ненужные версии.

Пусть, например, в системе выполняются транзакции с временны́ми метками 3, 100 и 105, а у записи есть версии с метками 1, 2, 5, 6, 7, 13, 45, 99, 103. Основываясь на временно́й метке самой старой транзакции, сборщик мусора может удалить только версию с меткой 1, хотя версии с метками 5, 6, 7, 13, 45 также никому не нужны, поскольку транзакции с метками 100 и 105 должны видеть версии с метками 99 и 103 соответственно.

В последних версиях SAP HANA применяются более сложные алгоритмы сборки мусора, подробное рассмотрение которых выходит за рамки книги.

Устройство механизма версионности в Tarantool сложнее, но элегантнее. Как мы помним, при традиционной работе с этой СУБД версионность вообще не нужна, т. к. транзакции выполняются действительно последовательно. Если же

администратор базы данных считает, что в системе необходимы интерактивные транзакции, где в промежутках между выполнением команд управление будет возвращено приложению, версионность можно включить.

В каждом кортеже[1] появляется бит `is_dirty`, который устанавливается, если кортеж изменён. Увидев «грязный» кортеж, транзакция может посмотреть историю его изменений. Изменения незавершённых транзакций также хранятся в истории, что позволяет параллельным транзакциям видеть актуальное состояние кортежей. Напомню, что в Tarantool реализован оптимистический подход к блокировкам, поэтому ничто не мешает нескольким транзакциям изменять один и тот же кортеж, но успешно завершится только та, которая первой выполнит команду `COMMIT`. Наконец, история чтения тоже хранится, что позволяет менеджеру конфликтов прервать любую транзакцию, если её сериализация невозможна.

Данные из истории удаляются сборщиком мусора. В снимок базы данных, записываемый на диск, история не попадает, поскольку если придётся восстанавливаться с диска, то все новые транзакции начнутся уже после внесения изменений, и им понадобятся только последние версии.

Литература

- Dan Pritchett. Base: An Acid Alternative.
  queue.acm.org/detail.cfm?id=1394128
- Стандарт SQL 92.
  www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt
- Overview of Oracle Database Transaction Isolation Levels.
  docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/data-concurrency-and-consistency.html
- Db2 Isolation Levels.
  www.ibm.com/docs/en/db2/11.5?topic=issues-isolation-levels
- SET TRANSACTION ISOLATION LEVEL (Transact-SQL).
  docs.microsoft.com/ru-ru/sql/t-sql/statements/set-transaction-isolation-level-transact-sql?view=sql-server-ver15
- PostgreSQL: изоляция транзакций.
  postgrespro.ru/docs/postgresql/14/transaction-iso

[1] Tarantool — не реляционная платформа, и единицей хранения в ней является не строка (row), а кортеж (tuple).

- MySQL Transaction Isolation Levels.
  dev.mysql.com/doc/refman/8.0/en/innodb-transaction-isolation-levels.html
- Isolation In SQLite.
  www.sqlite.org/isolation.html
- CockroachDB's Consistency Model.
  www.cockroachlabs.com/blog/consistency-model/
- MongoDB transactions.
  www.mongodb.com/docs/v4.2/core/transactions/
- YugabyteDB transaction isolation level.
  docs.yugabyte.com/preview/architecture/transactions/isolation-levels/
- Jepsen: consistency models.
  jepsen.io/consistency
- Владислав Родин. К чему может привести ослабление уровня изоляции транзакций в базах данных.
  habr.com/ru/company/otus/blog/501294/
- Deeply understand Isolation levels and Read phenomena in MySQL & PostgreSQL.
  dev.to/techschoolguru/understand-isolation-levels-read-phenomena-in-mysql-postgres-c2e
- Adrian Colyer. A Critique of ANSI SQL Isolation Levels.
  blog.acolyer.org/2016/02/24/a-critique-of-ansi-sql-isolation-levels/
- Db2 log records.
  www.ibm.com/docs/en/db2/11.5?topic=apis-db2-log-records
- Alex Sorokoumov. Learn How CommitLog Works in Apache Cassandra.
  cassandra.apache.org/_/blog/Learn-How-CommitLog-Works-in-Apache-Cassandra.html
- PostgreSQL. Надёжность.
  postgrespro.ru/docs/postgrespro/15/wal-reliability
- Руководство по архитектуре журнала транзакций SQL Server и управлению им.
  docs.microsoft.com/ru-ru/sql/relational-databases/sql-server-transaction-log-architecture-and-management-guide?view=sql-server-ver15
- Db2 Log file allocation and removal.
  www.ibm.com/docs/en/db2/10.5?topic=management-log-file-allocation-removal
- Внутреннее устройство WAL в PostgreSQL.
  postgrespro.ru/docs/postgresql/14/wal-internals

- C. Mohan, Don Haderle et al. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging.
  dl.acm.org/doi/pdf/10.1145/128765.128770
- @itskawal2000. Introduction of Shadow Paging.
  www.geeksforgeeks.org/shadow-paging-dbms/
- K. P. Eswaran et al. The Notions of Consistency and Predicate Locks in a Database System.
  people.csail.mit.edu/tdanford/6830papers/eswaran-notions-of-consistency.pdf
- Nikola Dimitrijevic. All about locking in SQL Server.
  www.sqlshack.com/locking-sql-server/
- Steve Thomas. Db2 Beginners Guide to Locks, Latches, Claims and Drains.
  bobbydreamer.com/108-db2-a-beginners-guide-to-locks-latches-claims-and-drains
- Subham Datta. Deadlock: What it is, How to Detect, Handle and Prevent?
  www.baeldung.com/cs/os-deadlock
- Peter Zaitsev. Innodb Architecture and Internals.
  www.percona.com/sites/default/files/PLDC2012-innodb-architecture-and-internals.pdf
- Nilnandan Joshi, Valerii Kravchuk. Understanding InnoDB Locks and Deadlocks.
  docs.huihoo.com/mysql/percona/live/mysql-conference-2015/Understanding-InnoDB-locks-and-deadlocks.pdf
- Руководство по блокировке и управлению версиями строк транзакций в Microsoft SQL Server.
  docs.microsoft.com/ru-ru/sql/relational-databases/sql-server-transaction-locking-and-row-versioning-guide?view=sql-server-ver15
- CockroachDB Transaction Layer.
  www.cockroachlabs.com/docs/stable/architecture/transaction-layer.html
- Kirill Loifman. Roadmap of Oracle Database releases.
  www.dadbm.com/roadmap-oracle-database-releases/
- PostgreSQL Release Notes.
  www.postgresql.org/docs/release/
- Егор Рогов. PostgreSQL 15 изнутри. М.: ДМК Пресс, 2023.
  postgrespro.ru/education/books/internals
- Juchang Lee, Hyungyu Shin et al. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA.
  www.researchgate.net/profile/Seongyun-Ko/publication/304021444_Hybrid_Garbage_Collection_for_Multi-Version_Concurrency_Control_in_SAP_HANA

- Александр Ляпунов. Менеджер транзакций для базы данных в оперативной памяти.
  habr.com/ru/company/vk/blog/540842/

Глава 6
Устройство СУБД

> Чти Догмат, но включай и Голову. Ибо не всякий провод, обозначенный на схеме «чёрным», воистину чёрен, и не всякий «белый» на самом деле бел. Помни — неисповедимы пути Конвейера, и Руководство По Ремонту следует уважать, но понимать метафорически.
>
> *Павел Иевлев, УАЗДао 2:15*

В книге периодически встречаются термины «база данных» и «экземпляр». Пришло время дать им определение, поскольку все компоненты по отдельности мы рассмотрели.

6.1. Экземпляр

Экземпляр (instance) — набор ресурсов операционной системы, выделенных для работы с общими данными в памяти и на диске. Экземпляр состоит из процессов и области памяти, общей для всех процессов.

Структура экземпляра

В подавляющем большинстве платформ экземпляр представляет собой процесс операционной системы. Такой экземпляр, в свою очередь, состоит из параллельных потоков выполнения, таких как поток операционной системы (thread) либо легковесный пользовательский поток (fiber)[1], а общая память экземпляра — просто область в памяти процесса. Однако в таких платформах, как Oracle и PostgreSQL, разработка которых началась в окружениях, не поддерживающих многопоточность, каждый поток экземпляра реализован как отдельный процесс

[1] Поток является объектом вытесняющей многозадачности (preemptive multitasking) и может быть прерван планировщиком ОС в любое время, за исключением выполнения критической секции. Задача — субъект кооперативной многозадачности (cooperative multitasking), и она выполняется до тех пор, пока сама не передаст управление.

операционной системы, а общая память — как сегмент разделяемой памяти (shared memory).

Любопытно, что версия Oracle для Microsoft Windows реализована как многопоточный процесс, поскольку Windows не поддерживает разделяемую память[1], а Db2, несмотря на почтенный возраст, даже в версии для Unix и Linux реализована как многопоточное приложение.

Для обозначения потока как части экземпляра применяются разные термины — процесс (process), задача (task) или даже диспетчеризуемый элемент (EDU, engine dispatchable unit). Далее мы будем использовать в значении «часть экземпляра» термин «процесс», а процесс как набор ресурсов операционной системы будем называть «процесс ОС».

Общая область памяти в большинстве платформ не называется вообще никак, поскольку с точки зрения реализации эта область ничем не отличается от локальной памяти процесса ОС[2]. Тем не менее для понимания архитектуры СУБД важно видеть эту разницу.

Понятие «экземпляр» неприменимо к встраиваемым базам данных, таким как SQLite и Berkeley DB (полностью встраиваемые) или H2, Interbase, GridGain/Apache Ignite, Oracle Coherence и другим, которые могут использоваться как встраиваемые. В этом случае код СУБД подключается к приложению в виде библиотек и выполняется непосредственно в процессе приложения.

Общая память экземпляра

В общей памяти экземпляра находятся структуры данных, доступ к которым необходим всем процессам. К ним относятся:

- всевозможные кеши;
- буферы записи;
- структуры, описывающие текущее состояние экземпляра;
- области памяти для внешних приложений.

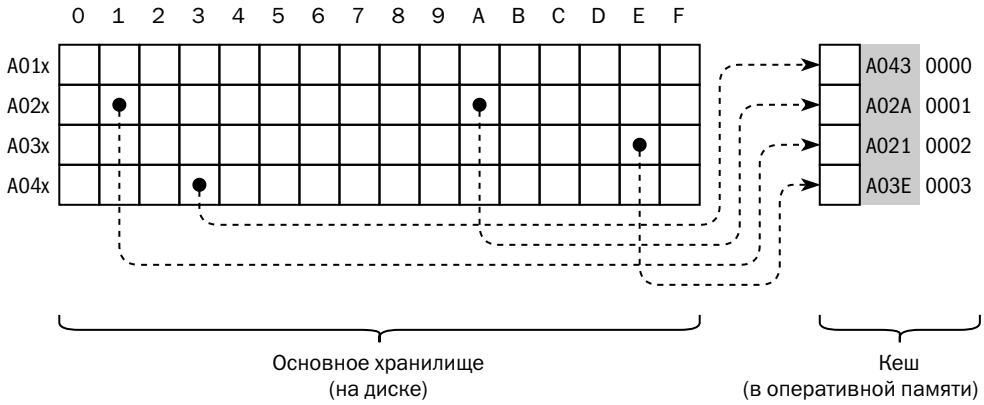
Кеш (cache) — область памяти, содержащая копию данных, повторный доступ к которым наиболее вероятен.

---

[1] Единственным механизмом совместного доступа нескольких процессов Windows к памяти является отображение файла в память (memory-mapped file), и PostgreSQL для Windows использует именно его.

[2] В Oracle, где экземпляр представляет собой набор процессов ОС, общая память называется SGA, system global area.

Как правило, в платформах, использующих блочные хранилища, большую часть общей памяти занимает **буферный кеш (buffer cache)**, то есть кеш, элементом которого является страница хранилища:



Такой кеш можно рассматривать как ассоциативный массив, ключом в котором является физический адрес дисковой страницы. Каждое обращение к странице приводит к поиску страницы в буферном кеше. Если образ страницы в кеше не найден, процесс, которому страница потребовалась, считывает её с диска и помещает в кеш, где этим образом может воспользоваться другой процесс.

Буферный кеш, как следует из названия, помимо обеспечения быстрого доступа к одним и тем же данным несколькими параллельными процессами, обеспечивает буферизацию записи: экземпляру достаточно изменить данные в кеше и вернуть ответ пользователю, а затем асинхронно записать изменённую страницу на диск.

Аналитической базе данных не требуется большой буферный кеш, поскольку вероятность конкурентного доступа к данным относительно низка, а изменения носят пакетный характер и практически не нуждаются в буферизации. При использовании платформ общего назначения (Oracle, Db2, Microsoft SQL Server) для аналитики память перераспределяется от буферного кеша в пользу локальной памяти процессов, используемой для сортировки или хеширования.

В платформах, использующих хранилища на основе LSM-деревьев, кеш также занимает значительную часть общей памяти. В отличие от платформ с блочным хранилищем, элементами такого кеша являются не страницы, а операции (mutations в терминологии Apache Cassandra, или tuple ops в Tarantool), состоящие из ключей и ассоциированных с ними значений.

Помимо основного кеша для данных, экземпляр может поддерживать ещё несколько кешей для ускорения доступа к информации.

Буфер изменений (change buffer) — хранит изменения вторичных индексов, если требуемых страниц нет в буферном кеше. Изменения вторичного индекса могут быть записаны асинхронно — например, когда какой-то из запросов обратится к индексу и поместит страницу в буферный кеш.

Дисковый кеш предназначен для быстрого доступа к некоторым областям на диске. Большинство платформ учитывают наличие дискового кеша в операционной системе, и документация к этим платформам содержит инструкции по его настройке. Некоторые платформы, например Oracle, рекомендуют вообще отключить дисковый кеш в операционной системе и отдать освободившееся место под буферный кеш экземпляра. Платформам с хранилищами на основе LSM-деревьев необходимо держать в памяти части файлов данных — карты ключей и фильтры Блума.

Кеш словаря данных (dictionary cache) содержит информацию о структуре данных, содержащихся в базе. Если для хранилищ «ключ—значение» этот кеш содержит только названия пространств ключей, то в реляционных платформах он содержит гораздо больше информации — имена и типы колонок таблиц, декларативные ограничения целостности и т. д.

Реляционные платформы могут поддерживать **кеш скомпилированных запросов**, позволяющий не компилировать одинаковые запросы повторно.

Наконец, некоторые платформы (Oracle, MySQL, Google BigTable, Solr, Azure Data Explorer) поддерживают **кеш результатов (query result cache)**. Если пользователь запускает запрос, идентичный уже обработанному, то вместо повторного выполнения запроса он может мгновенно получить кешированный результат.

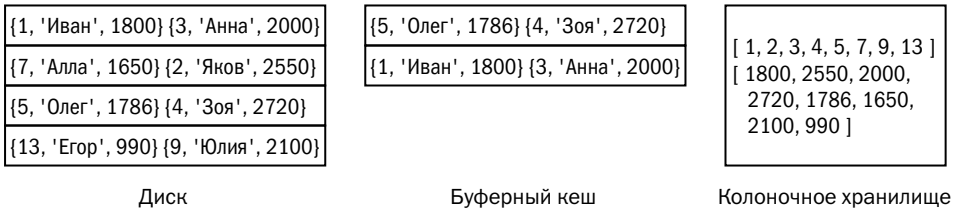
В отличие от дисковых платформ, основной объём памяти экземпляра СУБД в памяти занимает не кеш, а **основное хранилище данных**. Для этой области нет общепринятого термина: в документации к SAP HANA она называется heap memory, в документации к Tarantool/Picodata — arena, а в документации к большинству СУБД в памяти не называется вообще никак.

Есть два принципиальных различия между основным хранилищем и кешем, которые необходимо понимать:

- Основное хранилище, в отличие от кеша, содержит все данные. В частности, отсутствие каких-либо данных в основном хранилище означает их полное отсутствие в базе, а их отсутствие данных в кеше дисковой БД не означает ничего.

- Структура основного хранилища оптимизирована для размещения в памяти, а не для обмена с диском. Если набор обрабатываемых данных помещается в память, то гораздо выгоднее использовать СУБД в памяти, чем поместить все обрабатываемые данные в кеш дисковой платформы.

Отдельного упоминания заслуживает колоночное хранилище (column store), которое поставляется в виде опции (In-memory Option) к платформе Oracle. Это особая область памяти, не связанная с буферным кешем.



Колоночное хранилище представляет собой нечто среднее между кешем и основным хранилищем. Как и кеш, оно содержит только те данные, которые есть в основном хранилище на диске. В то же время оно обладает и свойствами основного хранилища. В колоночном хранилище содержатся все строки таблиц, отмеченных для размещения в памяти, но могут быть исключены некоторые колонки. Формат данных в колоночном хранилище отличается от формата хранения на диске и оптимизирован для выполнения аналитических запросов[1].

Помимо буферного кеша, в системах управления базами данных используются специальные структуры в памяти для буферизации записи.

В каждой платформе, использующей журналирование, обязательно есть **журнальный буфер (log buffer)**. Это небольшая (мегабайты, максимум десятки мегабайт) область памяти, куда транзакции сохраняют журнальную информацию. Информация из журнального буфера записывается на диск и передаётся на дочерние серверы (реплики) отдельными процессами. Буферизация позволяет радикально уменьшить количество операций ввода-вывода за счёт пакетной записи изменений.

В платформах, использующих хранилища на основе LSM-деревьев, перед записью в файл данные накапливаются в специальной структуре под названием **memtable** — это тоже не что иное, как буфер.

[1] Данные в Oracle In-memory Option хранятся по колонкам; подробнее об этой технологии — в главе 3 «Структуры хранения данных».

Часть памяти экземпляра отводится под хранение **текущего состояния экземпляра**. Примером такой информации могут служить:

- список процессов экземпляра;
- список пользовательских сеансов;
- список активных транзакций;
- таблица блокировок (объём и содержимое зависят от реализации менеджера блокировок);
- статистика ввода-вывода, использования памяти и др.

Информация о текущем состоянии доступна пользователю через системные представления или хранимые процедуры. Например, в Oracle насчитывается более 600 динамических представлений (dynamic performance views), отображающих состояние экземпляра, а в PostgreSQL — около 150 процедур и функций.

Ещё одна область общей памяти — **буфер для обмена информацией** между процессами, используемый, когда несколько процессов выполняют один и тот же запрос. Такая ситуация характерна для аналитических запросов, когда дочерние процессы читают данные с диска и выполняют предварительную обработку (фильтрацию, сортировку подмножества), а процесс-координатор объединяет полученные результаты, выполняет финальную обработку и возвращает результат пользователю.

Наконец, в общей памяти экземпляра могут находиться **области памяти внешних приложений**. Как правило, это приложения, разработанные поставщиком платформы, а иногда и входящие в комплект поставки СУБД — например, утилиты резервного копирования. Обмен данными между приложением и базой данных настолько интенсивен, что выгода от интеграции через память перевешивает риски. Примером таких областей могут служить область Java (Java pool), или область логической репликации (Streams pool) в Oracle, или память для утилит (util_heap) в Db2.

В заключение — несколько замечаний о связи описанной теории с реальными продуктами:

- Совсем не обязательно у каждой платформы есть все перечисленные области памяти. Безусловно необходимым является лишь журнальный буфер (если, конечно, платформа не использует нежурналируемые структуры), а также, в зависимости от устройства хранилища, буферный кеш, основное хранилище или memtable.
- Не всегда у администратора есть возможность влиять на распределение памяти. Если в ранних версиях Oracle ещё была необходимость явно зада-

вать значения параметров, устанавливающих размеры каждой области, то в Microsoft SQL Server возможности управления с самого начала сведены к двум параметрам — `max_server_memory` и `min_server_memory`. По мере развития все платформы движутся в сторону упрощения настроек и автоматического перераспределения памяти.

- Зачастую архитектура памяти детально не документирована. Разработчики считают, что либо пользователю это не нужно, либо он при необходимости посмотрит исходный код. Эпоха фундаментальной документации типа Oracle Concepts или IBM Redbooks уходит в прошлое.

Пользовательские процессы

Процессы, составляющие экземпляр, делятся на две большие группы — процессы, обеспечивающие обработку пользовательских запросов, и процессы, решающие внутренние задачи экземпляра.

Чтобы пользователь имел возможность подключиться к базе данных, в составе экземпляра должен быть процесс, принимающий пользовательские соединения. В PostgreSQL этим занимается процесс-почтмейстер (postmaster), в Microsoft SQL Server — безымянный поток, а в Db2 и Oracle — процесс-слушатель (listener), причём в Oracle слушатель формально даже не является частью экземпляра.

Установив соединение с пользовательским приложением по одному из доступных протоколов, чаще всего — TCP для удалённых клиентов или именованные каналы (named pipes) либо сокеты (Unix sockets) для локальных, — процесс-слушатель запускает серверный процесс, поддерживающий соединение и выполняющий операции над данными от имени клиентского приложения.

Существует два подхода к обработке пользовательских запросов.

При первом подходе все операции над данными выполняются непосредственно тем процессом, который поддерживает соединение с клиентом. При втором процесс-диспетчер только поддерживает соединения с пользователями, а когда необходимо выполнить действия над данными, обращается к рабочему процессу (worker) из пула. Если в пуле нет свободных процессов, то диспетчер в зависимости от настроек либо порождает новый рабочий процесс, либо ждёт освобождения одного из существующих.

Подход с выделенным процессом для каждого соединения проще в реализации и в эксплуатации. Он реализован во всех современных платформах.

Большинство современных бизнес-приложений написаны на Java и других языках, использующих JVM, в стандартную библиотеку которой входит пул соединений с базой данных. Таким образом приложение поддерживает минимально необходимое количество соединений.

Некоторые платформы позволяют организовать пул соединений на стороне базы данных. Microsoft SQL Server переходит в режим использования пула автоматически при достижении определённого количества клиентских соединений; Oracle требует настройки режима разделяемого сервера (shared server) — в отличие от традиционного режима выделенного сервера (dedicated server); в MySQL Enterprise или MariaDB доступна опция ThreadPool. Для PostgreSQL используются внешние пулы[1] — например, PgBouncer или Pgpool-II. Пулинг соединений на стороне сервера может быть полезен, если по каким-то причинам количество соединений слишком велико — например, количество пользователей приложения выросло сильнее, чем рассчитывали разработчики, или с одним экземпляром работает несколько приложений.

Важно отметить, что у пользовательского сеанса может быть состояние, сохраняемое между запросами, — например, значения глобальных переменных. Если для каждого сеанса выделен процесс, то это состояние хранится в локальной памяти процесса, а при использовании пула процессов состояние необходимо хранить где-то вовне. Для платформ, реализованных как многопоточный процесс ОС, разницы между глобальной и локальной памятью нет, а вот для многопроцессных — есть. Oracle в режиме разделяемого сервера создаёт специальную область в общей памяти для хранения состояния сеансов (user global area, UGA), а PostgreSQL в документации предупреждает об ограничениях, накладываемых использованием пула.

В аналитических базах данных нередка ситуация, когда запрос должен прочитать большой объём данных или выполнить операцию над большим количеством объектов. В этом случае серверный процесс порождает несколько рабочих процессов и распределяет работу между ними, а сам выступает в роли координатора, а также выполняет ту часть запроса, которую необходимо выполнить в один поток, — например, финальную сортировку. Впервые такой подход был применён в СУБД Teradata: с пользователем взаимодействует процесс PE (parsing engine), а данные обрабатываются процессами AMP (access module processor) и проходят финальную обработку в транспорте BYNET. Как обычно, каждый произво-

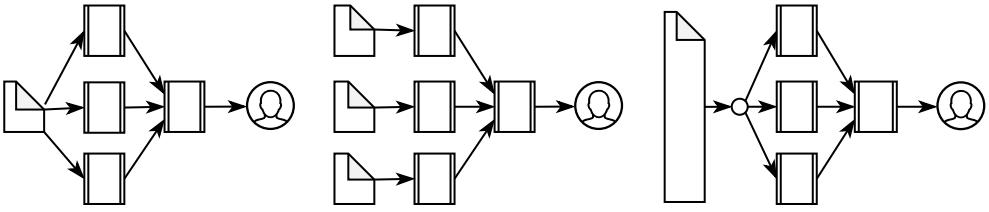
[1] В СУБД Postgres Pro Enterprise реализован встроенный пул соединений.

дитель придумывает свою терминологию: в Oracle пользовательский процесс (user process) может порождать параллельные серверные процессы (PX servers), в Db2 агент (db2agent) порождает вспомогательные процессы-агенты (db2agntp и db2agnts), в Microsoft SQL Server поток (thread) использует для параллельного исполнения множество рабочих потоков (worker threads), в PostgreSQL есть ведущий процесс (leader process) и рабочие процессы (worker processes).

При компиляции запроса сервер отмечает места, где работа может быть сделана параллельно, и отражает параллельное выполнение в плане запроса. Фактическое количество параллельных процессов зависит от общей мощности и текущей загрузки сервера, а также от настроек экземпляра.

В Teradata и других массивно-параллельных системах (Greenplum, Microsoft PDW и др.) каждый процесс работает с данными, которые хранятся на узле, где он запущен. Если все данные одинаково доступны всем процессам, сервер может применять разные стратегии распределения данных между ними:

- тиражирование (broadcast);
- деление по диапазонам (range);
- деление по хеш-сумме (hash).

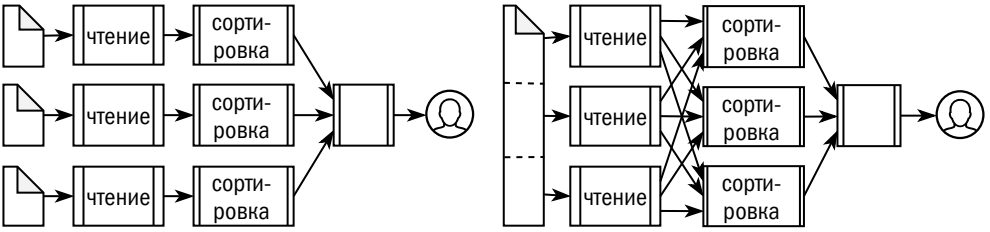


Тиражирование данных во все параллельные процессы используется, когда их объём небольшой. Например, если большая таблица соединяется с маленьким справочником, то проще всего маленький справочник скопировать в память каждого процесса.

Деление по диапазонам применяется, когда данные разделены на несколько физически независимых частей. Например, если требуется подсчитать количество записей в таблице или соединить таблицу со справочником, каждый процесс может взять себе произвольный набор страниц. Особенно хорошо работает деление по диапазонам в сочетании с секционированием таблиц, когда каждый процесс работает со своим набором секций.

Распределение по хеш-сумме необходимо, когда данные из разных физических диапазонов должны обрабатываться совместно, — например, если требуется группировка по колонке, не являющейся ключом секционирования. В этом случае один или несколько процессов читают данные, вычисляют хеш-сумму от значения колонки или набора колонок и передают запись процессу, отвечающему за соответствующее значение хеш-суммы.

При выполнении запроса возможно сочетание параллельной обработки с конвейеризацией, когда один процесс отвечает за чтение данных, а другой — за обработку:



деление по диапазонам деление по хеш-сумме

Процессы могут обмениваться большими объёмами информации; для этого обмена используются области в общей памяти.

Служебные процессы

Помимо процессов, обеспечивающих непосредственно обработку данных, существует набор процессов, обеспечивающих функционирование экземпляра. Такие процессы называют фоновыми (background) или служебными (utility).

У каждой платформы есть десяток-другой обязательных служебных процессов и примерно столько же необязательных. Названия процессов и распределение конкретных функций между ними зависят от платформы и могут быть найдены в документации или в исходном коде. Перечислим лишь их основные функции.

1. Управление остальными процессами экземпляра: порождение новых процессов, мониторинг, освобождение ресурсов после аварийного завершения процессов и восстановление экземпляра.
2. Управление памятью: выделение и освобождение общей памяти, освобождение временных дисковых сегментов.
3. Восстановление данных после сбоя экземпляра.

4. Асинхронная запись изменённых данных (страниц или заполненных сортированных таблиц) на диск; установка контрольных точек, то есть отсечение ненужной журнальной информации.
5. Запись журнала БД на диск, а также архивирование файлов журнала и передача журнальной информации на ведомые экземпляры в процессе репликации.
6. Сбор статистики — как по распределению данных, так и по нагрузке на экземпляр.
7. В распределённых платформах — восстановление распределённых транзакций после сбоев.
8. Фоновое выполнение пользовательских процедур.
9. Оптимизация дискового хранилища — например, объединение файлов в LSM-дереве или очистка неактуальных версий строк (vacuum) в PostgreSQL.
10. Поиск взаимоблокировок.

6.2. База данных

База данных (database) — это набор логически связанной информации, хранимой в электронном виде. В подавляющем большинстве случаев это означает набор файлов, однако иногда это могут быть данные на «сырых» устройствах или набор произвольных объектов в объектном хранилище S3.

Связь базы данных и экземпляра

Термины «экземпляр» и «база данных» зачастую употребляют как синонимы, поскольку обычно по контексту ясно, что именно имеется в виду. Тем не менее это разные вещи.

В отличие от экземпляра, который можно запустить и остановить, база данных существует независимо от того, запущен ли экземпляр и есть ли у пользователей доступ к данным.

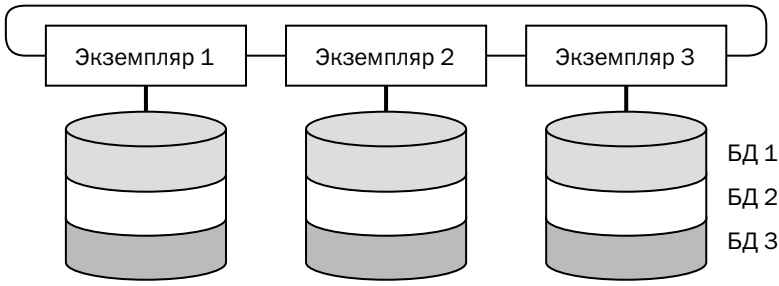
Один экземпляр может обслуживать несколько баз данных, и наоборот — несколько экземпляров могут обслуживать одну и ту же базу. Связь между базой и экземпляром «один к одному» характерна только для Oracle 8i и более ранних релизов.

В монолитных платформах — как традиционных (PostgreSQL, Microsoft SQL Server, MySQL, Db2, Oracle 12c и новее с технологией CDB — container database), так и более новых (OrientDB, Neo4j) — один экземпляр может обслуживать несколько баз данных одновременно.

Точное определение понятия «база данных» и степень изоляции одной базы от другой в рамках экземпляра зависят от платформы. Так, например, в MySQL и PostgreSQL понятие «база данных» — скорее логическое, чем физическое, и данные разных баз могут физически располагаться в одних и тех же файлах. В Oracle базы данных физически отделены друг от друга, и администратор даже может переносить базы данных между экземплярами (технология pluggable databases), но при этом экземпляр использует общий журнал для всех обслуживаемых баз. А вот в Microsoft SQL Server даже журналы у каждой базы данных свои.

В таких платформах, как Redis, Tarantool/Picodata, GridGain/Apache Ignite (а также в Oracle Real Application Cluster релизов с 9i по 11g), одна база данных обслуживается несколькими экземплярами.

В большинстве же современных распределённых платформ (Oracle RAC 12c+, Apache Cassandra, MongoDB, CockroachDB, YugabyteDB, ClickHouse, SAP HANA и др.) кластер может обслуживать несколько баз данных, то есть однозначного соответствия между базой данных и экземпляром нет.



Состав базы данных

Из чего же состоит обсуждаемый «набор информации»? Даже простой текстовый файл, содержащий значения, разделённые запятыми, можно рассматривать как базу данных, однако структура реальной базы данных значительно сложнее.

База данных состоит из следующих компонентов:

1. Каталог — описание физической структуры базы данных. Ссылка на каталог передаётся экземпляру при старте через параметры или через переменную среды. Прочитав каталог, экземпляр знает, из каких файлов состоит база

данных. В некоторых платформах роль каталога может играть каталог файловой системы.

2. Словарь данных — описание логической структуры базы данных. Обратившись к словарю данных, экземпляр узнаёт, какие объекты — таблицы, коллекции — хранятся в базе, где располагаются данные этих объектов и какова структура этих объектов (если модель данных платформы предусматривает описание структуры).
3. Собственно файлы данных, включая файлы, содержащие прошлые версии объектов.
4. Оперативный журнал, то есть журнальные файлы, необходимые для восстановления согласованного состояния данных. Большинство платформ требуют сохранения оперативного журнала даже в том случае, если экземпляр остановлен корректно и файлы базы данных находятся в согласованном состоянии.
5. Временные файлы, то есть файлы, в которых хранятся области сортировки и хеширования, не помещающиеся в оперативную память, а также данные временных таблиц, то есть таблиц, содержимое которых актуально только на время сеанса. Экземпляр при старте игнорирует всю информацию во временных файлах, оставшуюся в базе данных со времени предыдущего запуска экземпляра, а резервные копии БД не включают в себя временные файлы. И тем не менее временные файлы являются частью базы данных.
6. Вспомогательные физические структуры, специфические для конкретной платформы. Так, например, в MySQL можно сконфигурировать так называемый «буфер двойной записи» (doublewrite buffer). Все изменённые страницы сначала записываются в этот буфер одним непрерывным фрагментом, а позже фоновый процесс переносит эти страницы в файлы данных — туда, где им положено находиться. В PostgreSQL имеется множество файлов для хранения состояния транзакций, данных репликации, статистики и др. Полный список файлов, составляющий базу данных, можно найти в документации по конкретной платформе.

Есть и другие файлы, с которыми работает экземпляр, но которые не являются частью базы данных:

- Архивный журнал, то есть журнальные файлы, изменения из которых уже перенесены в файлы данных. Архивный журнал может понадобиться для восстановления резервной копии на точку во времени, но для текущего функционирования базы данных он не нужен.

- Внешние данные, то есть данные, которые не хранятся в базе, но могут использоваться экземпляром при выполнении запроса. Многие платформы позволяют «подключить» к базе CSV-файлы и читать из них данные, но вносить в них изменения запрещено, поскольку экземпляр не может гарантировать целостность данных, не защищённых блокировками и журналированием.
- Файл параметров экземпляра. Этот файл необходим только при запуске экземпляра и содержит различные настройки — объёмы областей памяти, настройки журналов и прочее. Изменение параметров экземпляра не меняет данные в базе.

Составляющие базы данных приведены условно, и конкретная реализация этих составляющих зависит от платформы. Соответствие компонентов и их названий в конкретных платформах и движках приведено в таблице ниже:

	Каталог	**Словарь**	**Файлы данных**	**Оперативные журналы**	**Временные файлы**
Oracle	Управляющий файл (controlfile)	SYS schema	Data files	Online redo logs	Temporary tablespaces
Db2	Table space[1] information files; Storage group control files; History file	SYSTEM tenant	User data table space	Active log; Log control files	Temporary table space
Microsoft SQL Server	БД master; Первичный файл (primary file) каждой БД	БД resource; Первичный файл (primary file) каждой БД	Первичный файл (primary file); Вторичные файлы (secondary files)	Журнал транзакций (transaction log)	БД tempdb
PostgreSQL	Каталог, определяемый переменной PGDATA	Системные каталоги (system catalogs)	Файлы данных (database files) в подкаталогах base, global, tblspc	Журнал предзаписи (write-ahead log), каталог pg_wal	Пространства, указанные в параметре temp_tablespaces

[1] Термин «табличное пространство» приводится в той же транскрипции, что и в фирменной документации. Так, в документации Oracle он пишется в одно слово (tablespace), а в документации Db2 — в два слова (table space).

	Каталог	Словарь	Файлы данных	Оперативные журналы	Временные файлы
MySQL[1]	System tablespace	Data dictionary (`mysql` и `information_schema`)	Tablespaces	Redo logs	Temporary tablespaces
MongoDB[2]	Каталог, определяемый параметром `storage.dbpath`	Metadata (MongoDB и WiredTiger)	Collection & index files	Journal files (write-ahead log)	–
Cassandra	Каталог, определяемый параметром `data_file_directories`	Пространства `system` и `system_distributed`	Data files, index files	Commit log	Кеш (key cache & row cache) в каталоге, определяемом параметром `saved_caches_directory`
LevelDB, RocksDB, Pebble	manifest log	manifest log	SST files	Write-ahead log	–

Литература

- Особенности конфигурирования памяти Db2 для основного хранилища.
  www.ibm.com/docs/ru/psfoa/1.1.0?topic=configuration-db2-memory-configurations-core-warehouse
- Oracle parallel execution concepts.
  docs.oracle.com/en/database/oracle/oracle-database/21/vldbg/parallel-exec-intro.html
- Db2 process model.
  www.ibm.com/docs/en/db2/11.1?topic=architecture-db2-process-model
- Oracle process architecture.
  docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/process-architecture.html
- Dave Thomas. Overview of Postgres Utilities Processes.
  pdfslide.us/download/link/overview-of-postgres-utility-processes

[1] С движком InnoDB.

[2] С движком WiredTiger.

- Microsoft SQL Server. Руководство по архитектуре потоков и задач.

  learn.microsoft.com/ru-ru/sql/relational-databases/thread-and-task-architecture-guide?view=sql-server-ver15
- Files in a WiredTiger database.

  muralidba.blogspot.com/2018/04/files-in-wiredtiger-database.html

Часть IV

Распределённые базы данных

Глава 7
Компромиссы распределённых баз данных

> Быстро, качественно, недорого: выбирайте любые два пункта.
>
> *Треугольник Хопкинса*

Требования к производительности вычислительных систем уже давно превосходят возможности даже самого мощного сервера. Естественное решение этой проблемы — переход к распределённым базам данных.

Распределённая база данных — это набор из нескольких логически связанных баз данных, управляемых связанными экземплярами. Узлы распределённой базы данных совместно обслуживают общий набор данных и могут располагаться на разных серверах.

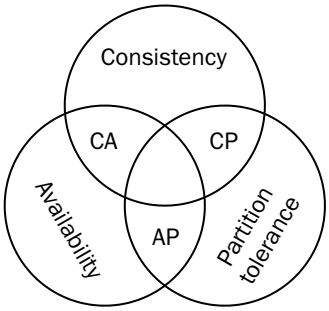
«Естественное» решение не значит «простое»: требования согласованности данных никуда не деваются, а к ненадёжности серверов добавляется ненадёжность каналов связи. Разработчикам, решившим применить в своих системах распределённую базу данных, следует помнить, что распределённой системе приходится жертвовать или согласованностью, или доступностью, или производительностью.

7.1. CAP-теорема

Утверждение Эрика Брюера (Eric Brewer) в докладе на конференции PODC (Principles of Distributed Computing) 2000 года, названное «CAP-теоремой», стало краеугольным камнем теории распределённых систем, вокруг которого вот уже два десятка лет не утихают споры. Не надеясь положить им конец, попытаемся хотя бы разобраться.

Формулировка CAP-теоремы

Профессор Брюер сформулировал своё утверждение следующим образом: распределённая система может обладать одновременно не более чем двумя свойствами из трёх — согласованностью, доступностью, устойчивостью к сетевому разделению (Consistency, Availability, Partition tolerance). Это утверждение было названо «CAP-теоремой» по первым буквам свойств, однако никакого доказательства этой «теоремы» в докладе не содержалось.



Все системы таким образом делятся на три класса — AP, CP, CA[1]. Система класса AP при разделении продолжает функционировать, но может быть не согласована, то есть разные узлы в ответ на один и тот же запрос могут выдавать разные ответы — например, разные версии запрошенного объекта. Система класса CP при разделении продолжает выдавать корректную информацию — например, актуальную версию запрошенного объекта, но при этом часть узлов системы перестаёт отвечать на запросы. Класс CA — системы, которые не подвержены сетевому разделению. Ряд экспертов считает, что такие системы невозможны, однако пример CA-системы содержится даже в исходном докладе Эрика Брюера — к этому классу можно отнести монолитную базу данных.

Спустя два года Сет Гилберт (Seth Gilbert) и Ненси Линч (Nancy Lynch) опубликовали статью «Гипотеза Брюера и возможность существования согласованных, доступных и устойчивых к разделению сервисов» («Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services»), в которой дали определения понятиям «согласованность», «доступность» и «устойчивость к разделению», а также доказали CAP-теорему (теперь уже без кавычек!) с учётом данных определений.

---

[1] Разумеется, система может обладать лишь одним из трёх свойств, но такие системы естественным образом вымерли.

В распределённых системах к аномалиям, возникающим из-за одновременного обращения клиентов к одним и тем же данным, добавляются аномалии, связанные с тем, что клиенты могут обращаться к разным узлам. В работе Гилберта и Линч под **согласованностью** понимается наличие глобального порядка операций, или, другими словами, иллюзия того, что каждая операция одновременно отражается на всех узлах распределённой системы — в точности как это происходит в монолитной системе с единственным узлом. При описании распределённых систем такой уровень согласованности называется линеаризуемостью, однако это не единственный допустимый уровень. Разные уровни согласованности рассмотрим в этой главе далее.

Доступность объясняется так: на каждый запрос, отправленный работающему узлу, должен быть получен ответ. При этом никак не определяется ни содержимое ответа, ни время ожидания — главное, чтобы оно просто было конечным, что в корне противоречит традиционному понятию «высокой доступности» (high availability). Говоря о распределённых системах с позиции здравого смысла, можно ожидать, что узел в ответ на запрос отдаст либо какое-то из значений, соответствующих запрошенному ключу, — не обязательно последнее, но обязательно одно из тех, что было записано, — либо сообщение о том, что ключ недоступен.

В докладе Брюера упоминается два показателя — доля ответов (yield) и доля правильных ответов (harvest). Первое считается как отношение запросов, на которые получен ответ, к общему количеству запросов, но при этом содержимое ответа не важно. Максимизация доли ответов соответствует определению доступности, данному Гилбертом и Линч. Профессор Брюер считает, что для вебсервисов (и других фронтальных систем) важен ответ как таковой, а вот для баз данных важнее правильный ответ. В частности, требования ACID гарантируют 100 % правильных ответов.

Наконец, под **сетевым разделением** понимается ситуация, когда распределённая система распадается на несколько частей, и ни одно сообщение от узлов, оказавшихся в одной части, не доходит до узлов, оказавшихся в другой части. Очевидно, что в любой реальной сети такая ситуация возможна. Если говорить о распределённых системах, то к классу CA можно отнести платформы, связность узлов в которых поддерживается ещё каким-то способом, кроме сети, — например, платформы с разделяемым диском (shared disk); если узел теряет доступ к диску, это равносильно отказу узла.

Работа Гилберта и Линч содержит доказательство CAP-теоремы для асинхронной и полусинхронной моделей сетевого взаимодействия. Асинхронная модель

подразумевает, что у каждого узла есть только сообщения, полученные от других узлов, и результаты собственных вычислений. В полусинхронной модели у каждого узла есть ещё и часы; часы на всех узлах идут с одинаковой скоростью, но между собой не синхронизированы. Полусинхронная модель достаточно точно описывает реальную инфраструктуру.

Критика CAP-теоремы

Профессору Брюеру было необходимо привлечь внимание к проблемам распределённых систем и начать их обсуждение, и публикация броского утверждения, к тому же названного «теоремой», сделала ровно то, что было нужно. Однако достаточно очевидно, что реального положения дел это утверждение не описывает.

Определения, использованные в теореме, допускают слишком вольную трактовку (хотя, напомним, они даны не самим автором, а его последователями). Если следовать букве этих определений, то система, после разделения отправляющая в ответ на ваш запрос случайные данные, может считаться доступной, а система, вообще ничего не отвечающая, — согласованной.

Линеаризуемость изменений, использованная в качестве определения согласованности, практически не достигалась распределёнными СУБД, существовавшими на момент появления CAP-теоремы. Большинство платформ того времени гарантировали причинную согласованность, более слабую, в силу высоких накладных расходов на поддержание линеаризуемости. За прошедшее с тех пор время появились распределённые базы данных, обещающие даже не линеаризуемость, а более сильную модель согласованности — строгую сериализацию.

CAP-теорема рассматривает сетевое разделение, но не рассматривает другие возможные сбои, например отказ дисков или узлов целиком. Кроме того, никак не учитывается способность распределённых баз данных восстанавливаться после ошибок.

Деление систем на три класса CA, AP и CP на самом деле описывает лишь два класса. К первому, CA, можно отнести платформы, не предусматривающие сетевого разделения. Описанию некоторых решений этого класса будет посвящён целый раздел. Второй класс, CP/AP, — «настоящие» распределённые системы без каких-либо общих компонентов (shared nothing). Они подвержены сетевому разделению, а будут ли они после разделения доступными или согласованными, зависит от настроек конкретной системы или даже конкретных транзакций.

Если рассматривать доступность не с точки зрения буквы определения, а с точки зрения здравого смысла, то есть потребовать, чтобы после разделения любой выживший узел возвращал вместо абстрактного ответа реальные данные, то окажется, что некоторые распределённые системы попросту не могут быть доступными. Дело в том, что совсем не обязательно в каждой из образовавшихся частей есть копии всех данных!

И наконец, отдельный вопрос, не затронутый CAP-теоремой, — что делать с изменениями данных после разделения распределённой системы. По букве теоремы первое же изменение делает систему несогласованной, то есть система, допускающая сетевое разделение, при таком разделении должна переходить в режим «только чтение». На практике современные платформы применяют различные компромиссы и механизмы отложенной синхронизации, которым посвящены специальные разделы.

Системы CP и AP и классификация PACELC

Если платформы с общим хранилищем данных устроены по-разному, то большинство платформ без общих компонентов, классифицируемых с точки зрения CAP-теоремы как CP[1] или AP, построены по общим архитектурным принципам:

- Каждый объект, хранящийся в базе данных (строка, кортеж, документ), имеет уникальный ключ. Для платформ класса «ключ—значение» это естественно, а вот таблица в реляционной базе данных может не иметь уникального ключа. В распределённой системе это требование становится обязательным.
- Данные разбиты на шарды. Попадание объекта в какой-либо шард зависит от значения ключа шардирования, который в подавляющем большинстве случаев совпадает с первичным ключом или является его частью.
- Каждый шард хранится на нескольких узлах, то есть избыточность хранения обеспечивается самой платформой, а не нижележащей инфраструктурой.
- При разделении кластера на несколько частей некоторые узлы могут прекращать отвечать на запросы или даже останавливаться, завершая исполнение кода СУБД.
- При потере узлов система восстанавливает избыточность хранения, создавая новые копии потерянных шардов, а при добавлении узлов может

[1] Насколько вообще эти системы можно назвать «согласованными».

перераспределять на них существующие данные. На время перераспределения данных возможны некоторые ограничения обслуживания.

Распределённые платформы могут быть как аналитическими (Greenplum, Microsoft PDW, Netezza, Teradata), так и ориентированными на обработку транзакций (Apache Cassandra, MongoDB, CouchDB и множество других).

Аналитические платформы с самого начала обеспечивали максимально строгую согласованность, используя дорогостоящие механизмы распределённых транзакций; это оправдано, т. к. изменение данных составляет малую часть нагрузки. Распределённые аналитические платформы обладают рядом особенностей:

- в этих платформах принята реляционная модель данных и реализована полная (а иногда и расширенная) спецификация языка SQL;
- как правило, в таких платформах есть выделенный узел-координатор, принимающий запросы от пользователей, распределяющий работу между узлами-обработчиками и осуществляющий финальную подготовку данных перед отправкой пользователям.

В ранних транзакционных распределённых платформах был «ползунок», позволяющий выбирать между согласованностью и доступностью, перемещая систему из класса CP в AP и обратно. Помимо крайних значений, у этого «ползунка» были ещё и промежуточные. Мало того, разработчик, использующий платформу, мог менять настройки для каждой операции в отдельности.

В 2010 году профессор Йельского университета Даниэль Абади (Daniel J. Abadi) опубликовал статью, в которой предложил новую классификацию компромиссов, названную PACELC[1]. Это аббревиатура, которая расшифровывается так: if Partitioned then choose between Availability and Consistency Else choose Latency or Consistency. В нормальном режиме система ищет баланс между временем отклика (latency) и согласованностью, а в разделённом — между согласованностью и доступностью. В отличие от CAP, эта классификация:

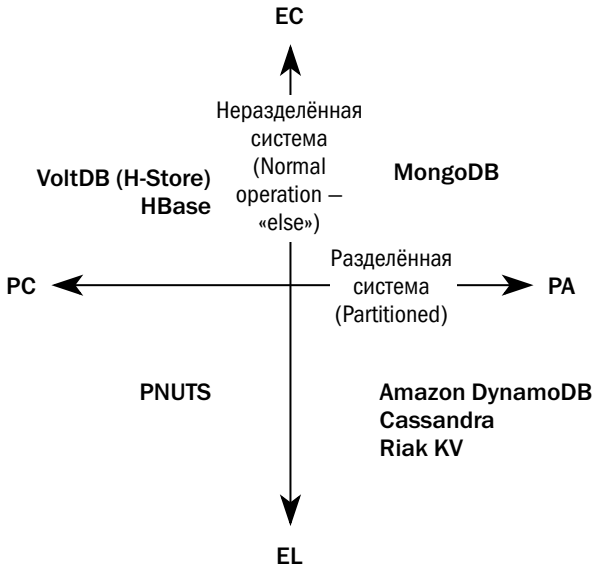
- рассматривает только «настоящие» распределённые системы, подверженные сетевому разделению;
- указывает компромиссы не только при разделении, но и в штатном режиме;
- вводит новый показатель — время отклика.

Распределённая система хранит несколько копий (реплик) данных. Чем больше реплик синхронно обновляется при записи и чем больше реплик опрашива-

[1] Читается «pass-elk», «пропусти лося».

ется при чтении, тем хуже (больше) время отклика, но больше шансов сохранить данные при аварии.

Каждая система может быть отнесена к одному из четырёх классов по классификации PACELC.

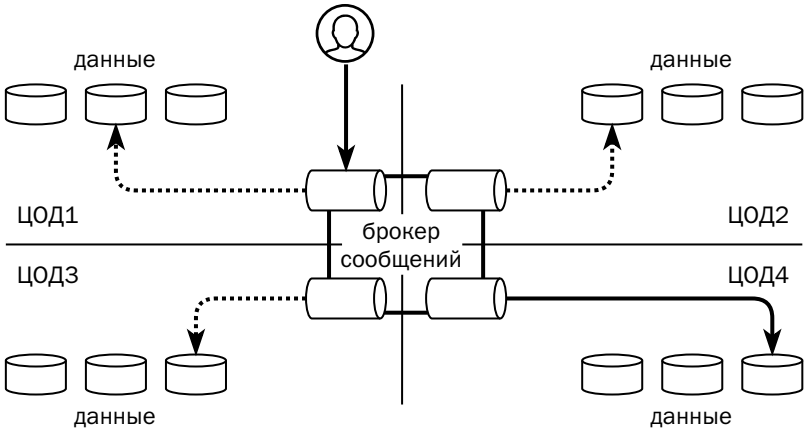


Amazon DynamoDB и его наследники Cassandra и Riak KV с настройками по умолчанию относятся к классу PA/EL, а MongoDB, опять же с настройками по умолчанию, — к классу PA/EC. Аналитическая платформа HBase и транзакционная VoltDB, обещающая ACID, подобно монолитным платформам, характеризуются как PC/EC.

Класс PC/EL на первый взгляд выглядит парадоксальным, но в качестве примера платформы такого класса приводится PNUTS, база данных для поддержки веб-приложений в Yahoo.

База данных PNUTS разделена на несколько регионов, которые могут находиться на значительном географическом удалении друг от друга. В каждом регионе хранится полная копия всех данных. Приложение читает данные из копии в ближайшем регионе, а изменения записывает в распределённый брокер сообщений. Брокер проверяет, можно ли изменить или создать запись с таким ключом, и если да, то подтверждает запись. Проверка осуществляется синхронно в одном регионе, который для этого ключа назначен основным, а само изменение данных

происходит асинхронно. Очевидно, что такая схема даёт максимально возможную производительность: трафик между регионами ограничен синхронизацией брокера, а синхронные операции включают в себя только запись в журнал (очередь) и чтение. Уровень согласованности при этом оставляет желать лучшего, что однозначно относит систему к классу EL. Кажущийся парадокс состоит в том, что при разделении эта система вдруг становится согласованной (PC). Но на самом деле платформа не повышает согласованность, а понижает доступность, отключая регионы, находящиеся в меньшинстве.



В заключение статьи профессор Абади подчёркивает, что классификация по PACELC ни в коем случае не может рассматриваться как исчерпывающая характеристика платформы, в особенности учитывая, что изменением настроек можно «перемещать» систему из одного класса в другой.

Современные транзакционные распределённые платформы обещают сильную согласованность вплоть до сериализации транзакций, то есть относятся к классам */EC. В них реализованы оптимизированные алгоритмы распределённых транзакций, которые на современном оборудовании в большинстве случаев выполняются по наилучшему сценарию (best-case).

7.2. Исторический экскурс: CA-системы

Как было сказано выше, распределённые системы подвержены сетевому разделению. Что же такое тогда CA-система?

Профессор Брюер в качестве примера приводит монолитную систему, несмотря на то что его лекция — о системах распределённых. Но есть и распределённые системы, не подверженные сетевому разделению, — это платформы с общим компонентом. Если у узла есть связь с этим компонентом, значит, у него есть и связь с остальными узлами, а если связь потеряна, то это равносильно отказу узла.

В качестве такого компонента выступает общая система хранения данных (shared disk). В подавляющем большинстве случаев это означает подключение через SAN[1], что ограничивает применение CA-платформ крупными предприятиями, способными содержать SAN-инфраструктуру.

Чтобы несколько серверов могли работать с одними и теми же данными, необходима кластерная файловая система. Такие файловые системы есть, например, в портфелях HPE (CFS), Veritas (VxCFS) и IBM (GPFS). С большой вероятностью СУБД, которой необходима распределённая файловая система, будет работать с любой такой системой. На практике производители СУБД тестируют свои платформы лишь с некоторыми распределёнными ФС, публикуют список протестированных систем в документации и гарантируют поддержку только в том случае, когда выбрана файловая система из списка.

Oracle RAC

Опция Real Application Cluster впервые появилась в 2001 году в релизе Oracle 9i. В таком кластере несколько экземпляров сервера работают с одной и той же базой данных.

Oracle может работать как с кластерной файловой системой, так и с собственным решением — ASM, Automatic Storage Management.

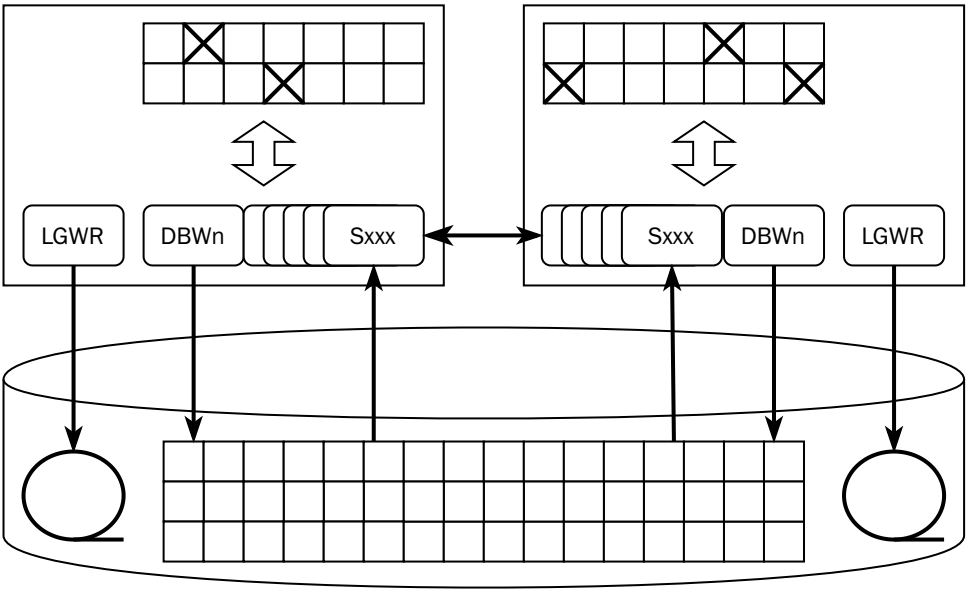
Каждый экземпляр ведёт свой журнал. Транзакция выполняется и фиксируется одним экземпляром.

Все экземпляры поддерживают собственный кеш, и одни и те же страницы (блоки) могут находиться одновременно в кешах нескольких экземпляров. Более того, если какая-то страница нужна одному экземпляру и она есть в кеше другого экземпляра, он может получить его у «соседа» при помощи механизма cache fusion, вместо того чтобы читать с диска.

Но что произойдёт, если одному из экземпляров потребуется изменить данные?

---

[1] Storage area network, сеть хранения данных. Системам хранения данных посвящён специальный раздел в главе 12 «Оборудование».



Особенность Oracle в том, что у него нет выделенного сервиса блокировок: если сервер хочет заблокировать строку, то запись о блокировке ставится прямо на той странице памяти, где находится блокируемая строка. Благодаря такому подходу Oracle — чемпион по производительности среди монолитных баз, т. к. сервис блокировок никогда не становится узким местом. Но в кластерной конфигурации такая архитектура может приводить к интенсивному сетевому обмену и взаимным блокировкам.

Как только строка блокируется, экземпляр оповещает все остальные экземпляры о том, что во всём кластере должна существовать единственная копия страницы, в которой хранится эта строка. Если другому экземпляру понадобится изменить строку в той же странице, он должен ждать, пока изменения, внесённые первым экземпляром, не будут записаны в журнал на диске (при этом транзакция может продолжаться). Может случиться и так, что страница будет изменена последовательно несколькими экземплярами, и тогда при записи страницы на диск придётся выяснять, у кого же хранится её актуальная версия.

Случайное обновление одних и тех же страниц через разные узлы RAC приводит к резкому снижению производительности базы данных — вплоть до того, что производительность кластера может быть ниже, чем производительность единственного экземпляра.

Правильное использование Oracle RAC — физическое деление данных (например, при помощи механизма секционированных таблиц) и обращение к каждому набору секций через выделенный узел. Главным назначением RAC стало не горизонтальное масштабирование, а обеспечение отказоустойчивости.

Если узел перестаёт отвечать на периодические опросы (heartbeat), то узел, обнаруживший потерю первым, запускает процедуру голосования на диске. Если и здесь потерянный узел не отметился, то один из узлов берёт на себя обязанности по восстановлению данных:

- «замораживает» все страницы, которые находились в кеше пропавшего узла;
- считывает журналы (redo) потерянного узла и повторно применяет изменения, записанные в этих журналах, попутно проверяя, нет ли у других узлов более свежих версий изменяемых страниц;
- откатывает незавершённые транзакции.

Чтобы упростить переключение между узлами, в Oracle есть понятие сервиса (service) — виртуального экземпляра. Экземпляр может обслуживать несколько сервисов, а сервис может переезжать между узлами. Экземпляр приложения, обслуживающий определённую часть базы (например, группу клиентов), работает с одним сервисом, а сервис, отвечающий за эту часть базы, при выходе узла из строя переезжает на другой узел.

IBM PureData System for Transactions

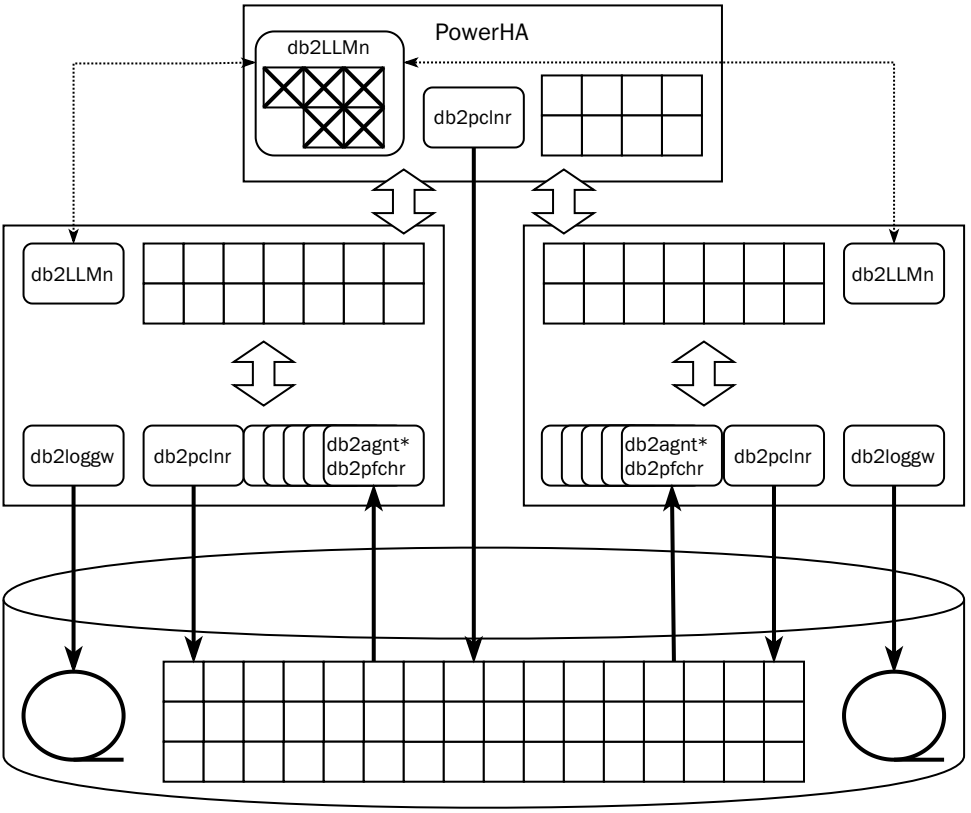
Кластерное решение Голубого Гиганта для СУБД идеологически является наследником кластера Parallel Sysplex, построенным на «обычном» оборудовании вместо мейнфреймов. В 2009 году вышел продукт Db2 pureScale, представляющий собой комплект программного обеспечения, а с 2012 года IBM предлагает программно-аппаратный комплекс (appliance) под названием PureData System for Transactions. Не следует путать его с PureData System for Analytics, которая есть не что иное, как переименованная Netezza.

Архитектура pureScale на первый взгляд похожа на Oracle RAC: точно так же несколько узлов подключены к общей системе хранения данных, и на каждом узле работает свой экземпляр СУБД со своими областями памяти и журналами транзакций. Но, в отличие от Oracle, в Db2 есть выделенный сервис блокировок, представленный набором процессов `db2LLM*`. В кластерной конфигурации этот сервис выносится на отдельный узел, который в Parallel Sysplex называется coupling facility (CF), а в PureData — PowerHA.

PowerHA предоставляет следующие сервисы:

- менеджер блокировок;
- глобальный буферный кеш;
- область межпроцессных коммуникаций.

Для передачи данных от PowerHA к узлам БД и обратно используется удалённый доступ к памяти, поэтому кластерный интерконнект должен поддерживать протокол RDMA (Remote Direct Memory Access). PureScale может использовать как Infiniband, так и RDMA over Ethernet.



Если узлу нужна страница, которой нет в его локальном кеше, то этот узел запрашивает страницу в глобальном кеше, и только если и там она отсутствует, читает её с диска. В отличие от Oracle, запрос идёт только в PowerHA, а не в соседние узлы.

Если экземпляр собирается менять строку, он блокирует её в исключительном режиме, а страницу, где находится строка, — в разделяемом режиме. Все блоки-

ровки регистрируются в глобальном менеджере блокировок. Когда транзакция завершается, узел посылает сообщение менеджеру блокировок, который копирует изменённую страницу в глобальный кеш, снимает блокировки и помечает недействительной изменённую страницу в кешах других узлов.

Если страница, в которой находится изменяемая строка, уже заблокирована, то менеджер блокировок прочитает изменённую страницу из памяти узла, сделавшего изменения, уберёт копии изменённой страницы из кешей других узлов и отдаст разделяемую блокировку страницы узлу, который её запросил.

Грязные (изменённые, но пока не сохранённые на диск) страницы могут быть записаны на диск как с обычного узла, так и с PowerHA (castout).

При отказе одного из узлов pureScale восстановление ограничено только теми транзакциями, которые в момент сбоя ещё не были завершены: страницы, изменённые этим узлом в завершившихся транзакциях, есть в глобальном кеше на PowerHA. Узел перезапускается в урезанной конфигурации на одном из серверов кластера, откатывает незавершённые транзакции и освобождает блокировки.

PowerHA работает на двух серверах, и основной сервер синхронно реплицирует своё состояние на резервный. При отказе основного узла кластер PowerHA продолжает работу с резервным узлом.

Разумеется, если обращаться к набору данных через один узел, общая производительность кластера будет выше. PureScale даже может заметить, что некоторая область данных обрабатывается одним узлом, и тогда все блокировки, относящиеся к этой области, будут обрабатываться узлом локально без коммуникаций с PowerHA. Но как только приложение попытается обратиться к этим данным через другой узел, централизованная обработка блокировок будет возобновлена.

Внутренние тесты IBM на нагрузке, состоящей из 90 % чтения и 10 % записи, что очень похоже на реальную промышленную транзакционную нагрузку, показывают почти линейное масштабирование до 128 узлов. Условия тестирования, увы, не раскрываются.

HPE NonStop

Своя высокодоступная платформа есть и в портфеле Hewlett-Packard Enterprise. Это платформа NonStop, выпущенная на рынок в 1976 году компанией Tandem Computers. В 1997 году компания была поглощена компанией Compaq, которая, в свою очередь, в 2002 году влилась в Hewlett-Packard.

NonStop используется для построения критичных приложений — например, HLR[1] или процессинга банковских карт. Платформа поставляется в виде программно-аппаратного комплекса (appliance), включающего в себя вычислительные узлы, систему хранения данных и коммуникационное оборудование. Сеть ServerNet (в современных системах — Infiniband) служит как для обмена между узлами, так и для доступа к СХД.

В ранних версиях системы использовались специализированные собственные процессоры, которые были синхронизированы друг с другом: все операции исполнялись одновременно несколькими процессорами, и как только один из процессоров ошибался, он отключался, а один из оставшихся продолжал работу. Позднее система перешла на обычные процессоры (сначала MIPS, затем Itanium и, наконец, x86), а для синхронизации стали использоваться другие механизмы:

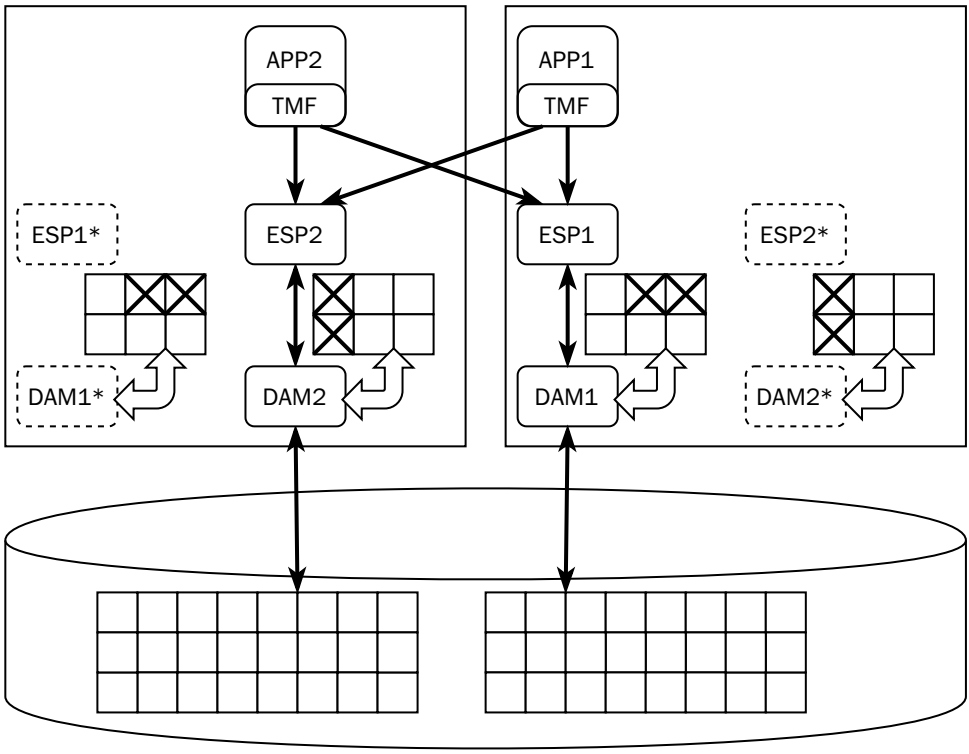
- Сообщения: у каждого системного процесса есть двойник (shadow), которому активный процесс периодически отправляет сообщения о своём состоянии. При сбое основного процесса процесс-двойник начинает работу с момента, определённого последним сообщением.
- Голосование: у системы хранения данных есть специальный аппаратный компонент, который принимает несколько одинаковых обращений и выполняет их только в том случае, если обращения совпадают. Вместо физической синхронизации процессоры работают асинхронно, а результаты их работы сравниваются только в моменты ввода-вывода.

Начиная с 1987 года на платформе NonStop работают реляционные СУБД — сначала SQL/MP, а позже SQL/MX.

Вся база данных делится на части, и за каждую часть отвечает свой процесс Data Access Manager (DAM). Он обеспечивает запись данных, кешировние и механизм блокировок. Обработкой данных занимаются процессы-исполнители (ESP, Executor Server Process), работающие на тех же узлах, что и соответствующие менеджеры данных. Планировщик SQL/MX делит задачи между исполнителями и объединяет результаты. При необходимости внести согласованные изменения используется протокол двухфазной фиксации, обеспечиваемый библиотекой TMF (Transaction Management Facility).

[1] Home Location Register — база данных сотового оператора, которая содержит оперативную информацию об абонентах: идентификатор SIM-карты, привязанные телефонные номера, текущее положение абонента, набор доступных услуг и др.

Система NonStop поддерживает процессы-двойники для DAM и ESP, как и для других процессов (на рисунке ниже двойники обозначены звёздочкой и пунктирным контуром).



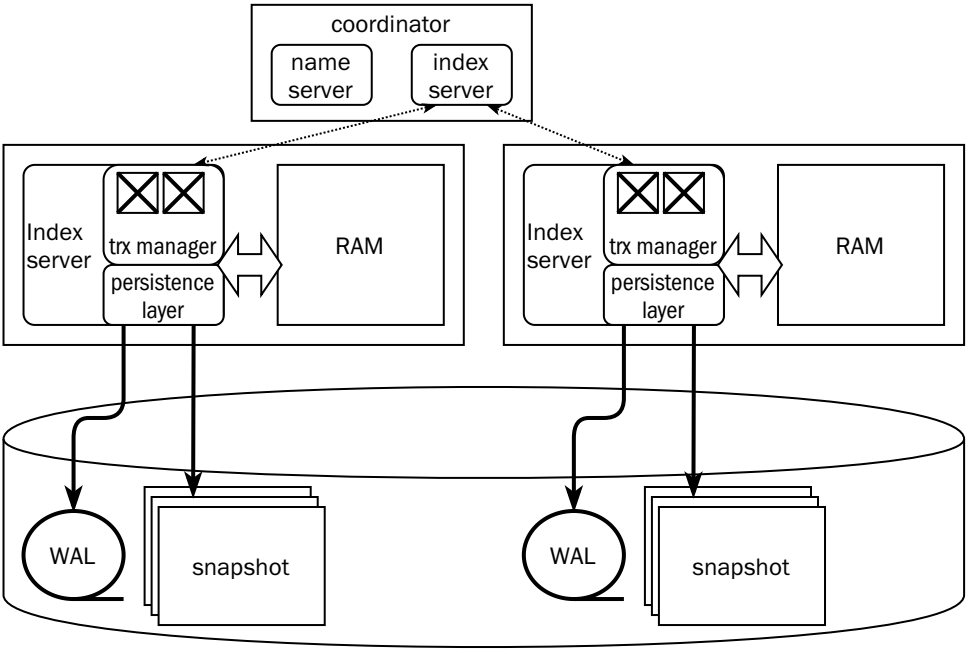
NonStop SQL умеет приоритезировать процессы так, чтобы длинные аналитические запросы не мешали исполнению транзакций. Однако её назначение — именно обработка коротких транзакций, а не аналитика. Разработчик гарантирует доступность кластера NonStop на уровне пять «девяток», то есть всего пять минут простоя в год.

SAP HANA

Первый стабильный релиз СУБД HANA (1.0) состоялся в ноябре 2010 года, а пакет SAP ERP перешёл на HANA с мая 2013 года. Платформа базируется на купленных технологиях: TREX Search Engine (поиск в колоночном хранилище), СУБД P*TIME и MAX DB.

Само слово «HANA» — акроним, означающий High performance ANalytical Appliance. Платформа содержит два движка — монолитный строчный и распределённый колоночный. Колоночный движок требует общую кластерную файловую систему для всех узлов, и некоторые программно-аппаратные комплексы, поставляемые компанией Lenovo, допускают даже эксплуатацию без SAN — роль общей СХД играет кластер GPFS на локальных дисках.

В отличие от платформ Oracle, IBM и HPE NonStop, SAP HANA хранит первичный образ данных в оперативной памяти, записывая на диск только журналы и периодические снимки — для восстановления в случае аварии.



Каждый узел кластера HANA отвечает за свою часть данных, а карта данных хранится в специальном компоненте — Name Server, расположенном на узле-координаторе. Данные между узлами не дублируются. Информация о блокировках также хранится на каждом узле, но в системе есть глобальный детектор взаимных блокировок.

Клиент HANA при соединении с кластером загружает его топологию и в дальнейшем может обращаться напрямую к любому узлу в зависимости от того, какие данные ему нужны. Если транзакция затрагивает данные единственного узла, то она может быть выполнена этим узлом локально, но если изменяются данные

нескольких узлов, то узел-инициатор обращается к узлу-координатору, который открывает и координирует распределённую транзакцию, фиксируя её при помощи оптимизированного протокола двухфазной фиксации.

Узел-координатор дублирован, поэтому в случае выхода координатора из строя в работу немедленно вступает резервный узел. А вот если выходит из строя узел с данными, то единственный способ получить доступ к его данным — перезапустить узел. Как правило, в кластерах HANA держат резервный (spare) сервер, чтобы перезапустить потерянный узел как можно быстрее.

7.3. Согласованность в распределённых системах

В разделе, посвящённом изоляции транзакций, мы рассматривали разные уровни изоляции, дающие разные гарантии согласованности данных. Уровни согласованности, рассматриваемые в этом разделе, никак не связаны с уровнями изоляции транзакций.

Линеаризация изменений

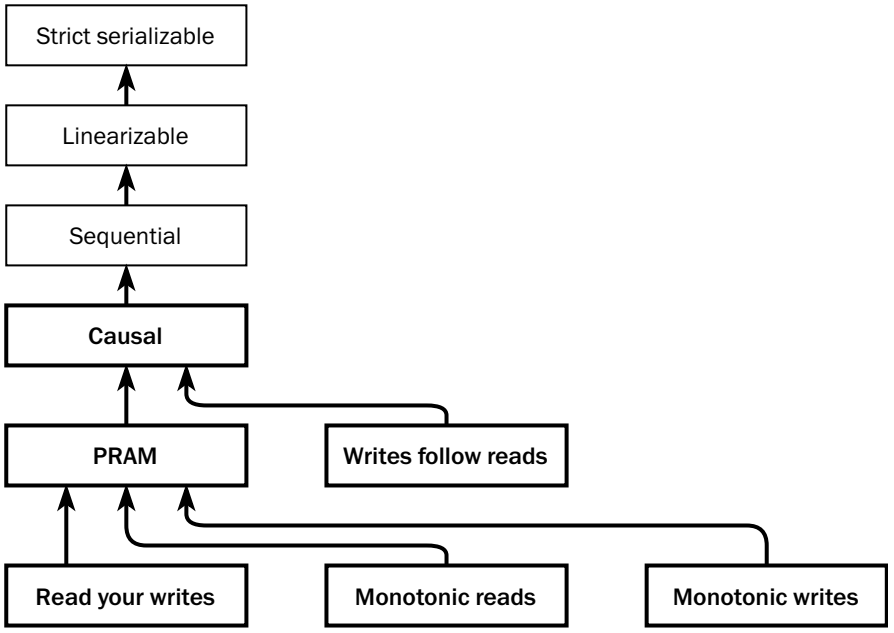
Работая с монолитной системой, приложение подразумевает линеаризацию изменений (linearizability). Это означает, что любое чтение возвращает результат последнего изменения данных. Для обеспечения линеаризации в системе необходимы часы, чтобы приписать каждой операции уникальную временну́ю метку и обеспечить сравнение меток любых двух операций — только в этом случае можно однозначно сказать, какая операция была выполнена раньше. В монолитной системе такие часы есть, а вот в распределённой системе создать единые часы, возвращающие идеально синхронизированное время для всех узлов, невозможно.

Если распространить гарантии линеаризации не на атомарные изменения, а на транзакции, то получится самая строгая из возможных моделей согласованности. Система, реализующая такую модель, называется строго согласованной (strict consistent или strict serializable). Все остальные гарантии согласованности, от линеаризации и слабее, формулируются именно в терминах изменений, а не в терминах транзакций, которые могут состоять из нескольких изменений.

Производительность системы с едиными часами растёт вместе с производительностью сервера, на котором эта система работает. При использовании

нескольких физических серверов единые часы становятся недоступны, поэтому приходится ослаблять требования к согласованности.

Эмпирическое правило гласит, что чем слабее модель согласованности, тем больше возможностей масштабирования системы, то есть тем большей производительности можно достичь. Иерархия моделей приведена на рисунке ниже.

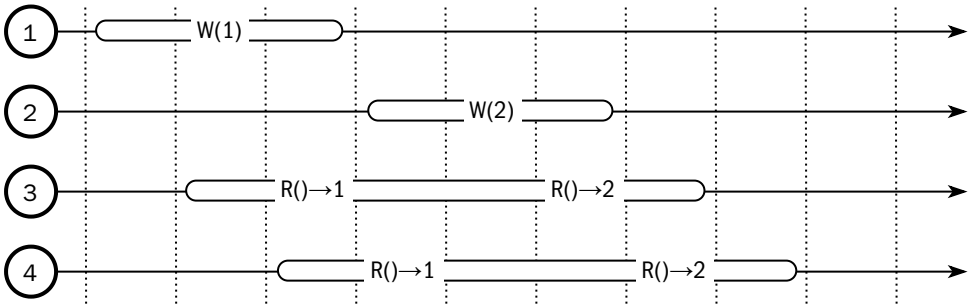


Модели согласованности в распределённых системах и уровни изоляции транзакций, о которых мы говорили ранее, никак не связаны — это ортогональные измерения. Каждую распределённую систему можно поместить в точку на плоскости, координатам которой соответствуют уровень изоляции и модель согласованности. Большинство систем при этом попадут в две точки. Ранние NoSQL-платформы обеспечивают уровень согласованности не более, чем PRAM, и уровень изоляции транзакций read committed, насколько вообще операции, обеспечиваемые этими платформами, можно назвать «транзакциями». Современные распределённые реляционные платформы обещают строгую сериализацию.

Разберёмся, что представляют собой уровни согласованности.

Рассмотрим распределённую систему, состоящую из четырёх узлов. Пусть вся база данных состоит из единственного ключа, над которым каждый узел может выполнить операцию чтения `R()` или операцию записи `W(x)`. Предположим, что

у нас есть волшебные часы, позволяющие наблюдать работу системы в реальном времени, как это показано на рисунке:

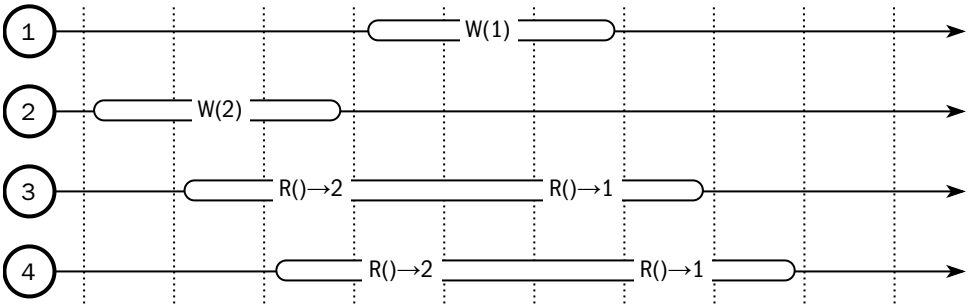


Все действия над данными происходят в некотором порядке, который называется расписанием (schedule). Линеаризация изменений гарантирует, что расписание одинаково для всех узлов, а действия упорядочены в соответствии с показаниями глобальных часов реального времени. В данном случае расписание выглядит так:

```
1:W(1) 3:R() 4:R() 2:W(2) 3:R() 4:R()
```

Если убрать требования к наличию единых системных часов, то полученная модель согласованности будет носить название последовательной (sequential). В этой модели расписание по-прежнему одинаково для всех узлов системы, но не привязано к часам реального времени.

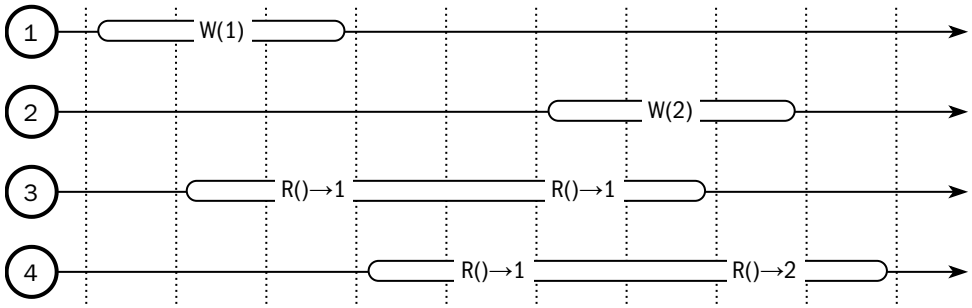
Пусть платформа не имеет доступа к волшебным часам, а временны́е метки операций генерируются из показаний часов узла, выполняющего операцию. Из-за рассинхронизации часов на узлах системы с точки зрения платформы рассматриваемая последовательность операций выглядит так:



Платформа считает, что запись на узле 2 произошла раньше записи на узле 1, но оба читающих узла, 3 и 4, видят эти изменения в одинаковом порядке. Расписание выглядит так:

```
2:W(2) 3:R() 4:R() 1:W(1) 3:R() 4:R()
```

Линеаризация операций может быть выполнена единственным образом, а вот расписаний, обеспечивающих последовательную согласованность, может быть несколько, при этом может отличаться не только порядок операций, но и их результат. На предыдущем рисунке мы видели, что часы узла 2 отстают, и метка операции записи на втором узле меньше метки записи на первом узле, несмотря на то что эта операция выполнена позже. Пусть теперь часы на узле 2, наоборот, спешат:

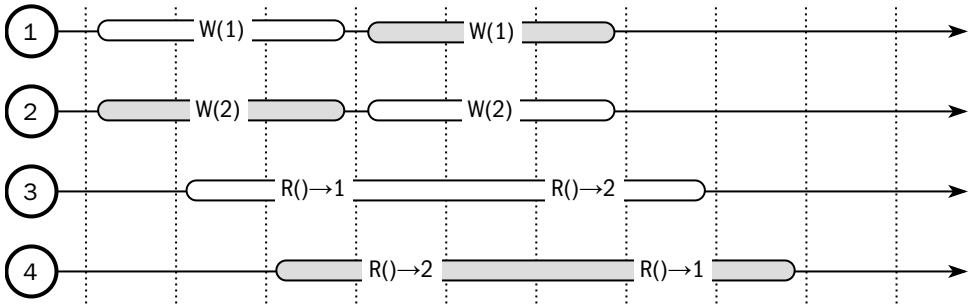


Расписание выглядит так:

```
1:W(1) 3:R() 4:R() 3:R() 2:W(2) 4:R()
```

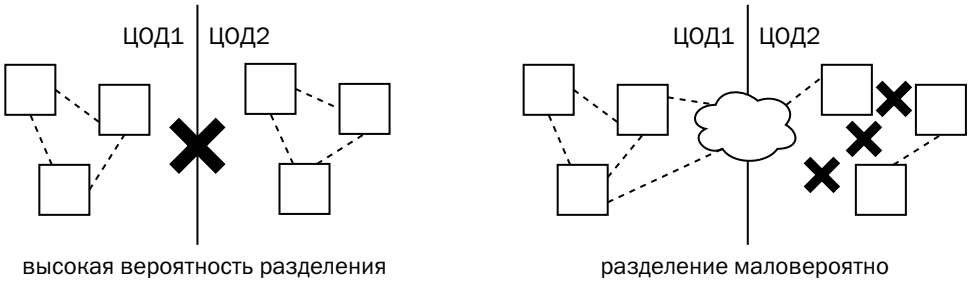
Мы видим, что чтение на узле 3 оба раза вернуло значение, записанное узлом 1. И тем не менее система, составившая такое расписание, удовлетворяет требованиям последовательной согласованности, поскольку расписание по-прежнему едино для всех узлов.

В распределённой системе, не удовлетворяющей требованиям последовательной согласованности, единого расписания не существует, и разные узлы могут видеть изменения в разном порядке. На рисунке ниже с точки зрения узла 3 изменения выполнены в порядке `1:W(1) 2:W(2)`, а с точки зрения узла 4 — в обратном порядке (на рисунке показаны серым):



Последовательная согласованность — достаточно сильная модель, и ранние распределённые системы давали более слабые гарантии, о которых мы поговорим ниже.

Ни линеаризация, ни последовательная согласованность не достижимы при разделении системы на несколько частей. Учитывая этот момент, современные распределённые СУБД создаются в расчёте на надёжные сети. В отличие от идеальной модели, реальная сеть неоднородна, то есть вероятность сетевого разделения значительна в одних местах (например, между вычислительными центрами) и пренебрежимо мала в других (внутри ЦОДа)[1]:



Современные распределённые СУБД учитывают топологию сети и распределяют данные так, чтобы при разделении по максимально вероятному сценарию каждый сегмент содержал бы копию всех данных.

Как и с уровнями изоляции транзакций, разные платформы могут использовать термины «линеаризация» и «сериализация» с некоторыми оговорками. Так, например, RethinkDB и pegasusDB гарантируют линеаризацию в рамках ключа.

[1] Разумеется, коммутатор, обеспечивающий передачу данных внутри ЦОДа, также может выйти из строя. Однако сегодня разработаны сетевые топологии, позволяющие поддерживать несколько резервных сетевых путей за счёт относительно небольшого избытка оборудования. См., например, «сеть Клоза» (Clos network).

Перед началом разработки приложений тщательно изучайте фирменную документацию!

Причинная согласованность

Уровень согласованности, который на самом деле достаточен для большинства приложений, называется причинной согласованностью (causal consistency). Это соглашение является подмножеством последовательной согласованности и гарантирует, что если между событиями (записью и чтением) есть причинно-следственная связь, то эти события упорядочены, а если нет, то порядок событий не определён.

Под причинно-следственной связью понимается ситуация, когда результат одного события зависит от результатов другого. Например, в одном сеансе приложение изменяет значение какого-то ключа, а в другом — считывает его.

Причинная согласованность включает в себя четыре требования:

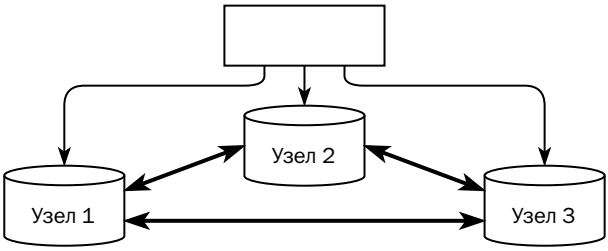
- видимость собственных изменений;
- монотонность чтения;
- монотонность записи;
- запись после чтения.

Видимость собственных изменений (read your writes, или read my writes) означает, что если в каком-то сеансе в базу данных внесены изменения, то все последующие чтения в этом сеансе должны видеть эффект этих изменений. Например, если размер ботинок клиента с ключом 100 установлен в 43, то запрос размера ботинок этого клиента должен вернуть 43. Важно, что речь тут только о том сеансе, где были сделаны изменения; другие сеансы могут получать и другие данные — например, предыдущее значение.

Пусть база данных состоит из нескольких узлов и запись в базу считается успешной, если её подтвердил хотя бы один узел. Остальные узлы получат данные позже — либо от приложения, но асинхронно, либо репликацией с других узлов. Чтение данных выполняется со случайного узла — например, с наименее загруженного.

Если приложение записывает данные на один узел, а читает с другого, причём до того, как операция записи будет выполнена на другом узле, то операция чтения вернёт предыдущую версию данных, и соглашение о видимости собственных изменений будет нарушено.

Видимость собственных изменений обеспечивается только в том случае, если чтение и запись происходят с одного и того же узла или набора узлов — в англо-



язычной литературе это называеся sticky availability. Если приложению необходимо, чтобы гарантия соблюдалась даже при сетевом разделении кластера базы данных, то оно должно записывать данные на несколько узлов и, возможно, читать с нескольких узлов, добиваясь наличия кворума.

Монотонность записи (monotonic write) подразумевает, что если изменения были внесены в базу данных каким-то сеансом в определённом порядке (например, сначала клиенту установили размер обуви в 41, а потом исправили на 43), то любой читающий сеанс увидит их в этом же порядке. При этом не накладывается никаких ограничений на изменения, внесённые разными сеансами.

Если каждое приложение всё время работает с одним узлом, а данные с одного узла на другой реплицируются последовательно, то соглашение выполняется. Однако, как правило, приложение в одном сеансе может записывать данные в несколько разных узлов, и если две операции, выполненные на двух разных узлах, дойдут до третьего в обратном порядке, то монотонность записи будет нарушена.

Монотонность чтения (monotonic read) заключается в том, что если читающий сеанс увидел какую-то версию данных, то он уже никогда не увидит предыдущую версию. В нашем примере это означает, что если сеанс, считав размер обуви клиента, получил 43, то следующая операция чтения в этом сеансе увидит или 43, или более новое значение, но никогда не увидит 41. Никаких ограничений на порядок чтения разными сеансами на этом уровне не накладывается: другие сеансы по-прежнему могут видеть размер обуви 41.

Если приложение читает данные с произвольного узла, то есть вероятность, что, получив свежую версию с одного узла, позже оно может получить предыдущую версию с другого. Таким образом, монотонность чтения нарушается.

Один из вариантов выполнения соглашений о монотонности чтения и записи заключается в том, что для каждого ключа выбирается главный узел, и все операции с данным ключом выполняются только на главном узле.

Второй вариант описан в статье сотрудников Принстонского университета Ричарда Липтона (Richard J. Lipton) и Джонатана Сандберга (Jonathan S. Sandberg) «Конвейер: масштабируемая разделяемая память» (PRAM, pipeline random access memory), опубликованной в 1988 году. Каждый процесс в описанной модели работает с данными в локальной памяти и рассылает сообщения об изменениях остальным процессам. Описанная схема в точности соответствует одновременному выполнению требований видимости собственных изменений, монотонности записи и монотонности чтения.

В современных распределённых платформах применяются и другие подходы к обеспечению согласованности, о которых мы поговорим в следующей главе.

Наконец, четвёртое требование называется «запись после чтения» (writes follow reads) или «соблюдение причинно-следственных связей в рамках сеанса» (session causality). Это требование означает, что если какой-то сеанс сделал изменение w_1, воспользовавшись результатами изменения другого сеанса w_0, то любой третий сеанс увидит изменения именно в последовательности (w_0, w_1), но не в обратной. Пусть, например, пользователь интернет-магазина изменил в своём профиле размер обуви с 41 на 43, а затем робот, рассылающий новую модель кроссовок по предварительным заказам, считал размер из профиля клиента и сформировал накладную на кроссовки 43 размера. Любой сеанс может увидеть одну из трёх ситуаций:

- у клиента 41 размер, накладная не сформирована (не внесено ни одно изменение);
- у клиента 43 размер, накладная не сформирована (внесено только первое изменение);
- у клиента 43 размер, накладная сформирована (внесены и первое, и второе изменения).

При этом никто не может увидеть ситуацию «у клиента 41 размер, накладная сформирована», в которой более позднее изменение сделано, а более раннее — нет. Но никакого ограничения на несвязанные изменения не накладывается: если, к примеру, другой клиент зарегистрировался в магазине, то третий сеанс может увидеть ситуацию, когда накладная уже сформирована, а профиля другого клиента ещё не существует, а четвёртый сеанс — обратную.

Из выполнения требования «запись после чтения» в общем случае не следует выполнение остальных требований (видимость собственных изменений, монотонная запись, монотонное чтение). Для причинной согласованности должны быть выполнены все четыре требования.

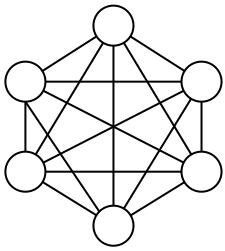
В литературе, посвящённой распределённым базам данных, употребляется термин «согласованность в конечном счёте» (eventual consistency). Такая согласованность означает, что данные постоянно реплицируются между узлами, и если остановить приложение, вносящее изменения в базу данных, то рано или поздно данные на всех узлах станут одинаковыми. Термин «согласованность в конечном счёте» не подразумевает какую-то модель согласованности, но и причинная согласованность не включает в себя требования непрерывной репликации: если репликация данных на узел прекратится и приложение будет продолжать работать с этим узлом, то причинная согласованность не будет нарушена. Требования здравого смысла подсказывают, что непрерывная репликация всё же важна, и чтобы подчеркнуть это в теоретических выкладках, в литературе используется термин «causal+»[1], включающий в себя и причинную согласованность, и согласованность в конечном счёте.

7.4. Топология кластера

Статическая топология

Распределённые системы решают проблему масштабирования за счёт распределения нагрузки между узлами, порождая при этом ряд новых проблем. В частности, каждому участнику распределённой системы необходимо знать состояние остальных узлов. Граф, по которому узлы обмениваются информацией о состоянии, называется топологией кластера (overlay network).

Самая простая и очевидная идея состоит в том, чтобы каждый узел периодически посылал всем остальным узлам короткое сообщение о своём состоянии (heartbeat). Такая топология называется полносвязным графом (full mesh):

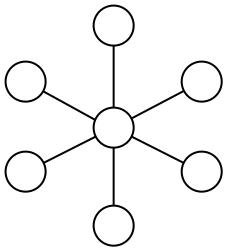


[1] На русский язык этот термин можно было бы перевести как «расширенная причинная согласованность».

Поскольку каждый узел непосредственно связан с каждым соседом, он узнаёт информацию о состоянии каждого узла немедленно и из первых рук. Проблема в том, что чем больше узлов в кластере, тем больше сообщений требуется передавать: количество связей каждого узла растёт линейно в зависимости от количества узлов в кластере, а общее количество связей — пропорционально квадрату количества узлов.

Современные сети настолько производительны, что даже на больших кластерах, состоящих из нескольких десятков узлов, обмен «каждый с каждым» не ведёт к деградации производительности. Полносвязный граф применяется, например, в MongoDB или Hazelcast.

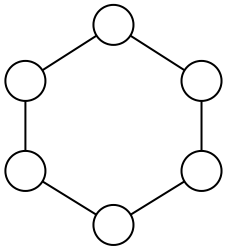
Существуют и альтернативные модели кластерной топологии, разработанные до эпохи повсеместного внедрения многоярусных сетей, построенных на базе 10-гигабитных или даже 100-гигабитных коммутаторов. Одной из таких альтернатив может служить топология «звезда» (star) — каждый узел обменивается состоянием с узлом-координатором, который информирует узлы о состоянии друг друга:



Расстояние между узлами в таком кластере не превышает двух, а количество связей растёт линейно с ростом количества узлов. Как правило, центральный узел выполняет только роль координатора, не обрабатывая данные, поэтому расстояние между рабочими узлами в точности равно двум.

Существенным недостатком такой топологии является то, что отказ узла-координатора ведёт к остановке всего кластера и выборам нового координатора. В связи с этим «звезда», как и полносвязный граф, используется в кластерах с небольшим количеством узлов — например, в Zookeeper и etcd. Также топология «звезда» используется в кластерах высокой доступности для монолитных платформ — Oracle, PostgreSQL и др. и в достаточно крупных кластерах — например, Hadoop, Oracle TimesTen Velocity Scale или брокере сообщений Kafka. В центре такой звезды находится не отдельный узел, а небольшой (обычно состоящий из трёх–пяти узлов) отказоустойчивый кластер — например, Zookeeper или etcd.

Ещё одна попытка оптимизировать топологию кластера — объединение узлов в кольцо (ring):



Здесь, как и в «звезде», количество связей растёт линейно с ростом количества узлов, а количество связей одного узла постоянно и равно двум. К преимуществам такой топологии относится отсутствие выделенного узла-координатора, а к недостаткам — большое время прохождения сообщений между узлами. Единственная известная автору платформа, использующая топологию «кольцо», — Apache Ignite/GridGain. Топология выглядит линейно масштабируемой, однако практический предел размера кластера — около 30 узлов.

Сравнение рассмотренных топологий приведено в таблице ниже:

	Полносвязный граф	**Звезда**	**Кольцо**
Количество связей узла	$N-1$	1	2
Общее количество связей	$N(N-1)/2$	$N-1$	N
Максимальное расстояние между узлами	1	2	$N-1$
Минимальное количество отказов сети для разделения кластера	$N-1$	—	2
Восстановление при отказе узла	Не требуется	При отказе координатора	Требуется

В изотропных топологиях, таких как полносвязный граф или кольцо, так же, как и в «звезде», может выделяться узел-координатор. В задачи этого узла входит принятие решений о запуске глобальных процессов — например, балансировки данных в связи с изменением количества узлов. Эта функциональность не требует дополнительных вычислительных ресурсов и не нарушает изотропию кластера. При потере узла-координатора новым координатором может быть выбран

любой из оставшихся узлов — в зависимости от принятого в платформе алгоритма выборов.

Важная задача для распределённой системы — обнаружение потерянных узлов (failure detection). Простейший метод заключается в том, чтобы считать потерянным узел, от которого не поступает сообщений в течение заданного времени (deadline failure detection). Проблема в том, что при задании большого тайм-аута кластер заметит потерю узла слишком поздно, а при малом тайм-ауте возможно исключение из кластера исправных узлов, которые не успели отправить информацию из-за кратковременных сбоев сети или задержек в самом программном обеспечении — например, из-за паузы, вызванной сборкой мусора в памяти.

Для увеличения точности обнаружения сбойных узлов в условиях ограниченной производительности сети был разработан адаптивный алгоритм (φ-accrual failure detection). На основании информации о некотором количестве последних сообщений (по принципу скользящего окна) для каждого узла рассчитывается среднее время между сообщениями и стандартное отклонение, а затем сравнением времени следующего сообщения с расчётным значением получается φ — уровень подозрительности. Это значение обладает следующими свойствами:

- φ работающего узла ограничен сверху и с течением времени стремится к 0 (но не обязательно всё время равен нулю);
- φ отказавшего узла монотонно возрастает и стремится к бесконечности.

В отличие от бинарных алгоритмов, определяющих узел только как «доступный» или «недоступный», адаптивный выдаёт значения в широком диапазоне, и система может принимать на основании этих значений разные решения.

Адаптивный алгоритм обнаружения сбойных узлов применяется в таких платформах, как Apache Cassandra или Hazelcast, но широкого распространения этот алгоритм не получил. Главный его недостаток — неустойчивость к сетевым штормам, а его достоинства в условиях быстрых и дешёвых сетей уже не так актуальны.

Наконец, в дополнение к прочим методам обнаружения сбоев может быть включен опрос сбойного узла (ping failure detection). Обнаружив, что какой-то узел не отвечает, другие узлы могут отправить ему явный запрос вида «жив ли ты?», и если ответ на запрос будет получен, это будет означать, что узел на самом деле функционирует.

Протоколы сплетен

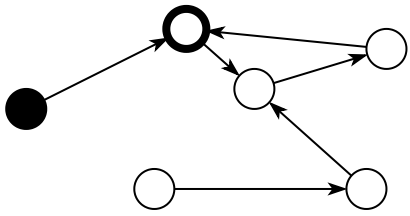
В больших кластерах информация (включая, но не ограничиваясь служебной информацией о составе кластера) может распространяться по протоколу сплетен

(gossip protocol). Суть технологии заключается в том, что каждый узел общается с узлами из своего окружения, передавая им информацию о себе и о тех узлах, о которых ему стало известно от других узлов. Выбор окружения зависит от конкретной реализации алгоритма.

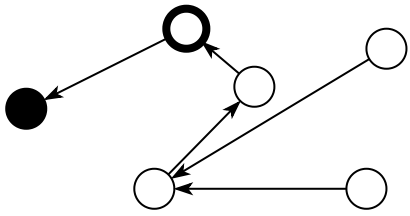
В отличие от людей, распространяющих сплетни, у узлов кластера нет цели скрыть какую-то информацию от других узлов, наоборот — они охотно делятся знаниями со всеми, стремясь дать каждому узлу полную картину. Таким образом, протокол больше похож на распространение инфекций. Более того, при описании шагов протокола узел, получивший информацию, называется «инфицированным» (infected), а не «информированным». Тем не менее общеупотребительным названием для этого семейства алгоритмов является именно «распространение сплетен».

Рассмотрим пример реализации алгоритма.

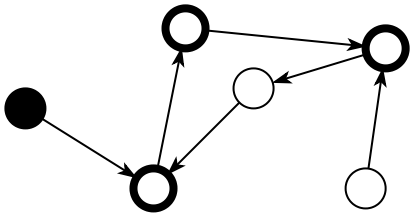
Пусть к существующему кластеру добавился узел. Новый узел знает адрес одного из существующих узлов и в момент t сообщает ему о своём появлении:



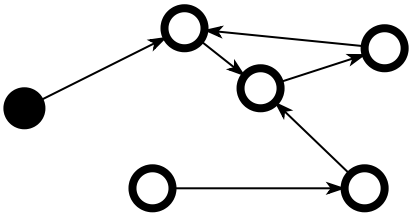
На рисунке видно, что теперь один из старых узлов знает о появлении нового узла. В следующий момент, $t+1$, узлы снова обмениваются информацией о топологии:



На этом шаге количество «инфицированных» узлов не изменилось, т. к. единственный «инфицированный» узел передал информацию новому узлу, чтобы тот узнал о других узлах кластера. В момент $t+2$ происходит новый обмен информацией:



На этом шаге «инфицированных» узлов стало три. И наконец, на следующем шаге, в момент $t+3$, о новом узле узнают все узлы:



У протокола распространения сплетен есть множество реализаций, отличающихся друг от друга разными параметрами:

- способом выбора узлов, с которыми происходит общение: в приведённом примере узлы выбираются случайно;
- количеством узлов, которым отправляет сообщение каждый узел: в нашем примере — один;
- алгоритмом чередования отправки и ожидания сообщений: в нашем примере в каждый момент времени каждый узел отправляет одно сообщение;
- в направлении передачи сообщений: в нашем примере передача однонаправленная от инициатора к выбранному узлу (push), но возможны также опрос выбранного узла (pull) и двунаправленная передача (push+pull).

Обобщённая реализация протокола сплетен представлена в 2009 году профессором Озалпом Бабаоглу (Özalp Babaoğlu) под названием T-MAN. Этот алгоритм предписывает каждому узлу запустить два потока, один из которых в цикле рассылает оповещения, а второй — принимает сообщения. Алгоритм рассылающего (активного) потока выглядит так:

```
wait(delta);
peer = select(N,sort(nodeset,myDescriptor,rank))
buffer = merge(nodeset,myDescriptor)
buffer = sort(buffer,peer,rank)
send(p,first(M,buffer))
peer_buffer = receive(p)
nodeset = merge(nodeset,peer_buffer)
```

Активный поток в цикле ожидает какое-то время, а затем сортирует все известные узлы (nodeset), используя функцию ранжирования узлов `rank()`, и выбирает случайный узел среди первых `N` узлов. После этого он записывает в буфер информацию об известных узлах и о себе, сортирует эту информацию так, как её отсортировал бы выбранный узел (peer), и отправляет выбранному узлу информацию о первых `M` узлах из сортированного буфера. Затем дожидается ответного сообщения и обновляет информацию об известных узлах информацией из полученного сообщения.

Слушающий (пассивный) поток выполняет следующую программу:

```
(peer_buffer, peer) = receive()
buffer = merge(nodeset,myDescriptor)
buffer = sort(buffer,peer,rank)
send(p,first(M,buffer))
nodeset = merge(nodeset,peer_buffer)
```

Получив сообщение от какого-либо узла, слушающий поток конструирует и отправляет ответное сообщение, а затем обновляет информацию об известных узлах информацией из полученного сообщения.

Параметрами этого алгоритма являются:

- `rank()` — функция определения ранга узлов, в соответствии с которым осуществляется сортировка;
- `N` — количество узлов, с которыми общается каждый узел;
- `M` — максимальный размер сообщения;
- `delta` — время ожидания между отправкой сообщений.

Изменяя значения этих параметров, можно регулировать накладные расходы на поддержание целостности кластера и управлять скоростью распространения информации.

Любопытно, что существуют наборы параметров, при которых алгоритм T-MAN реализует фиксированные топологии, описанные в предыдущем подразделе. В этих формулах `myID` обозначает идентификатор текущего узла, являющийся частью его полного описания (`myDescriptor`).

Кольцо:

- `M` не ограничено;
- `N` = 1;
- `rank()` = `(id+[nodeset]-myId) mod [nodeset]`, где `[nodeset]` — количество известных узлов.

Звезда:

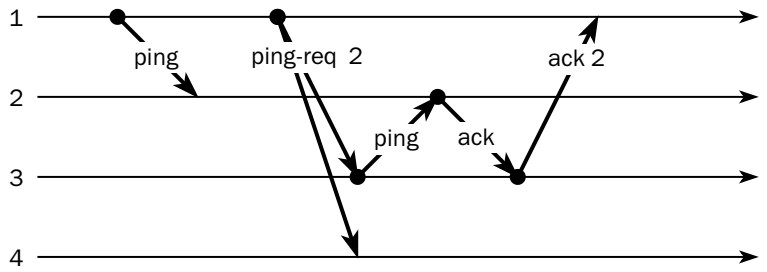
- `M` не ограничено;
- `N` = { `myId==min(id) ? [nodeset]-1 : 1` };
- `rank()` = { `myId==min(id) || id==min(id) ? 0 : 42` }.

Если `N` равен количеству узлов в кластере, а размер сообщения не ограничен, то T-MAN реализует классический «антиэнтропийный» протокол, где информация распространяется по кластеру в среднем за время $O(\log N)$. Эксперименты показывают, что такой же сложности можно добиться и с ограниченным размером сообщения, в особенности если оптимизировать алгоритм, запретив рассылающему потоку отправлять сообщения узлам, с которыми он контактировал в предыдущие `K` сеансов (tabu-list).

Один из популярных протоколов семейства gossip, используемый, например, в СУБД Tarantool, называется SWIM (Scalable Weakly-consistent Infection-style process group Membership). Важнейшие свойства этого протокола:

- при отказе узла один из оставшихся узлов узнает об этом за константное время, а все узлы — за время $O(\log N)$;
- узлы не синхронизированы и могут видеть состояние кластера по-разному, но рано или поздно информация дойдёт до каждого узла.

Протокол SWIM состоит из двух частей — обнаружения потерянных узлов и оповещения о смене топологии. Каждый узел периодически посылает запрос случайному узлу из кластера. Если запрашиваемый узел отвечает, то отправитель в своей таблице обновляет время последнего ответа от этого узла, а если нет, то отправляет случайному набору узлов просьбу проверить доступность подозрительного узла. Только если в течение некоторого периода времени никто из запрошенного набора не пришлёт информацию о доступности, то подозрительный узел помечается как потерянный:



Узел, обнаруживший потерю другого узла, рассылает сообщение об изменении топологии всем оставшимся узлам кластера.

Различные реализации протокола SWIM предусматривают некоторые усовершенствования:

- Информация о составе кластера передаётся не отдельными сообщениями, а вместе с периодическими запросами (ping, heartbeat) либо вместе с передачей данных между узлами. Служебная информация как бы «падает на хвост» другой информации, и в англоязычной литературе такой приём называется «piggybacking».
- Если узел не отвечает, то он помечается не как потерянный, а как подозреваемый в сбое. Информация о подозреваемом узле распространяется по кластеру, но если в течение некоторого времени узел ответит на запрос, он возвращается в список доступных. Если один узел считает узел X доступным, а другой — потерянным, то безопаснее верить худшей информации. Если же сам узел X заметил, что его считают потерянным, он увеличивает собственный счётчик «инкарнаций» и отправляет сообщение уже с новой «инкарнацией», которое считается более достоверным, чем все сообщения с меньшим значением счётчика.
- Алгоритм выбора узлов, которым отправляется запрос, может меняться. Так, например, вместо случайного выбора узел может перебирать несколько «соседних» узлов по кругу (round-robin).
- Для ускорения ввода узла в кластер новый узел добавляет в сообщение информацию о параметрах подключения (например, IP-порт), а в ответ получает сразу всю информацию о составе кластера.

Многие современные системы уходят от реализации протоколов семейства gossip из-за их сложности, переходя к более простым топологиям — звезде или полносвязному графу.

Литература

- Eric A. Brewer. Towards Robust Distributed Systems.
  people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf
- Seth Gilbert, Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services.
  citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.67.6951&rep=rep1&type=pdf

- Richard J. Lipton, Jonathan S. Sandberg. PRAM: A Scalable Shared Memory.
  ftp.cs.princeton.edu/techreports/1988/180.pdf
- MongoDB: Causal Consistency and Read and Write Concerns.
  www.mongodb.com/docs/manual/core/causal-consistency-read-write-concerns/
- Dr. Jack Lange. Consistency and replication consistency models.
  people.cs.pitt.edu/~jacklange/teaching/cs2510-f15/obsolete_lectures/07.2-Consistency Models.pdf
- Nicolas Liochon. CAP, Availability, High-Availability and Big Data databases in a world of partitions.
  blog.thislongrun.com/2015/04/cap-availability-high-availability-and_16.html
- Мартин Клеппман. Забудьте CAP-теорему как более не актуальную / пер. Андрея Гордиенкова.
  habr.com/ru/post/258145/
- Daniel J. Abadi. Consistency Tradeoffs in Modern Distributed Database System Design.
  www.cs.umd.edu/~abadi/papers/abadi-pacelc.pdf
- Zeyuan Hu. PNUTS: Yahoo!'s Hosted Data Serving Platform.
  zhu45.org/posts/2018/Mar/08/pnuts-yahoos-hosted-data-serving-platform/
- Виталий Исаев. Как мы создаём почтовую систему нового поколения Mailion. Принципы проектирования масштабируемых хранилищ данных.
  habr.com/ru/company/ncloudtech/blog/551484/
- Hayashibara Naohiro, Defago Xavier, Yared Rami, Katayama Takuya. The φ accrual failure detector.
  hdl.handle.net/10119/4784
- Márk Jelasity, Alberto Montresor, Ozalp Babaoglu. T-MAN: Gossip-based fast overlay topology construction.
  disi.unitn.it/~montreso/pubs/papers/comnet09.pdf
- Владислав Шпилевой. Tarantool+ 2019: SWIM — протокол построения кластера.
  slides.com/gerold103/tplus2019-swim

Глава 8
Изменение данных в распределённых системах

> Совместная работа — цель тиранов и деспотов. Свободные люди тянут в разные стороны.
>
> *Терри Пратчетт, «Правда»*

В предыдущей главе мы убедились, что обеспечение согласованности данных в распределённой системе — непростая задача. В этой главе рассмотрим алгоритмы, разработанные для решения этой задачи.

При переходе от монолитных систем к кластерам выяснилось, что при большом количестве узлов отказы становятся обычным явлением, несмотря на высокую надёжность каждого узла. Появление алгоритмов, описанных в этой главе, позволило обрабатывать различные аварии как штатные ситуации и естественным образом привело к отказу от высоконадёжных серверов.

8.1. Основные понятия

Для решения проблемы распределённого хранения данных используется несколько классов решений:

1. **Кластеризация монолитной БД**: источником изменений является единственный узел. Сменить лидера (источник изменений) можно только извне — либо вмешательством администратора, либо сигналом от специализированного ПО.
2. **Распределённый консенсус**: источником изменений является единственный узел, но алгоритм содержит в себе процедуру смены источника. Смена происходит без вмешательства извне — равноправные узлы договариваются о смене лидера между собой.
3. **Распределённые транзакции**: каждый узел выполняет свои действия над данными, и цель алгоритма — выполнить изменения атомарно, то есть либо подтвердить изменения на всех узлах, либо отменить их — также на всех узлах.

4. **Компенсирующие механизмы**: дизайн системы допускает некоторую несогласованность данных, но система принимает все меры к тому, чтобы при первой возможности привести данные на узлах к согласованному состоянию.
5. **Безопасные типы данных**: структуры данных, рассчитанные на то, что их содержимое изменяется несколькими узлами независимо друг от друга, а возможность синхронизации изменений есть не всегда.

Не менее интересны и практические применения описываемых алгоритмов. В конце главы приведены различные архитектурные решения, реализованные в известных распределённых системах.

Прежде чем перейти к подробному описанию перечисленных подходов, введём важное понятие: **кворум (quorum) — это минимальное количество узлов, необходимых для выполнения операции.** Если в операции участвует n узлов, а кворум для данной операции составляют k, то после подтверждения от k узлов операция считается выполненной.

Важно понимать, что кворум — отнюдь не синоним слова «большинство». Для примера рассмотрим трёхузловую систему, выполняющую две операции — чтение и запись объекта. Для каждой из этих операций значение кворума своё, причём оно зависит от требований к согласованности данных и от характера нагрузки на систему.

- Если важно, чтобы система возвращала одинаковые результаты вне зависимости от того, к какому узлу обратился клиент (именно такое определение согласованности подразумевает CAP-теорема), то сумма кворумов для чтения и записи должна быть больше, чем количество узлов ($Q_R + Q_W > n$). Например, при записи должны ответить все узлы ($Q_W = 3$), а читать можно с любого узла ($Q_R = 1$). Как правило, в реальных системах именно так и делают, потому что количество операций чтения значительно больше, чем количество операций записи. Однако если нагрузка по записи велика, а требования по согласованности сохраняются, то достаточно записи на два узла, но и читать данные надо с двух узлов ($Q_W = 2$, $Q_R = 2$).
- Если доступность важнее согласованности, то кворум для записи следует выбирать исходя из надёжности узлов и требований к производительности. Оба значения $Q_W = 2$ и $Q_W = 1$ будут корректными.

Типичная задача, для решения которой используется кворум, — определение рабочего набора узлов при разделении кластера (split brain). Обнаружив, что часть соседних узлов потеряна, узел должен принять решение: продолжать пол-

ноценную работу или перейти в аварийный режим — в зависимости от политики системы либо отключиться, либо выполнять только операции чтения. Решение узла зависит от того, собран ли в доступной части кластера кворум — то есть в данном случае именно большинство узлов. Для решения данной задачи количество узлов делают нечётным, причём часть узлов даже не обязана хранить данные — они используются исключительно при голосовании (пример такого подхода — узлы-арбитры в MongoDB).

8.2. Отказоустойчивый кластер с репликацией

Кластер с репликацией широко используется для обеспечения отказоустойчивости монолитных баз данных. Если по какой-то причине репликация остановилась, то сама реплика не способна определить причину остановки, которая могла произойти как из-за выхода из строя основного узла, так и из-за обрыва связи. В первом случае реплика должна стать основной базой и принять на себя нагрузку, а во втором — дожидаться восстановления связи.

Для определения причины сбоя и управления переключением роли узлов используется специальное кластерное программное обеспечение (clusterware). Примеры такого ПО приведены в таблице ниже.

ПО	Поставщик	Примечание
SunCluster	Sun Microsystems / Oracle	Компонент ОС Solaris. Чаще всего используется для кластеризации на уровне ОС
SystemMirror (ex. PowerHA, HACMP)	IBM	Кластерное ПО для AIX и Linux. Используется для кластеризации узлов и экземпляров БД Db2
ServiceGuard	Hewlett-Packard Enterprise	Компонент ОС HP-UX. Используется для кластеризации на уровне ОС
Veritas Infoscale Availability (ex. Veritas Cluster)	Veritas Technologies	Отдельно поставляемое кластерное ПО для Windows, Linux и коммерческих версий Unix. «Из коробки» имеет множество модулей для работы с самыми разными объектами — СУБД, серверами, файловыми ресурсами и т. д.
Corosync / Pacemaker	сообщество Corosync	ПО с открытым исходным кодом для всех Unix-систем, включая Linux, MacOS и коммерческие версии Unix. Чаще всего используется для кластерзации Linux-серверов и экземпляров PostgreSQL
Windows Server Failover Cluster (WSFC)	Microsoft	Компонент ОС Windows. Используется для кластеризации всех платформ Microsoft, включая Windows Server и SQL Server
Data Guard Broker	Oracle	Компонент СУБД Oracle. Используется для кластеризации экземпляров этой БД
Patroni	Zalando	Набор скриптов для управления СУБД PostgreSQL. Для определения состояния кластера требует надёжное распределённое хранилище — etcd, Consul, Zookeeper и т. п.

Задача кластерного ПО состоит в определении состояния кластера и переключении ролей узлов при необходимости. Под «узлом» кластер может понимать разные объекты — например, базу данных или сервер с заданным IP-адресом. Как бы это ни казалось странным на первый взгляд, это не синонимы: при переключении кластеризованной базы данных на резервный узел IP-адрес базы может измениться. Это лучше, чем обеспечение отказоустойчивости сервера, т. к. при подмене сервера резервный сервер должен находиться в одной подсети с основным (L2-связность), а при работе на уровне БД достаточно, чтобы оба узла, основной и резервный, были доступны по IP (L3-связность).

Для корректной работы кластерному ПО необходимы модули, работающие с объектами обеспечения отказоустойчивости.

Ряд программных продуктов разработан для управления кластерами из экземпляров конкретной СУБД — например, Oracle Data Guard Broker для Oracle или Patroni для PostgreSQL. Другие продукты (например, Corosync/Pacemaker, ServiceGuard) декларируют универсальность, но практика показывает, что они хороши только для кластеризации серверов, но не баз данных.

При расположении основного и резервного экземпляров БД в разных ЦОДах хорошим тоном является расположение дополнительного (кворумного) узла в третьем ЦОДе — это позволяет корректно отличать отказ узла от сетевого сбоя. Кворумный узел кластера требуется для надёжного хранения информации о конфигурации, и полноценный сервер БД там не нужен. В некоторых продуктах (например, в Windows Server Failover Cluster) в качестве третьего узла может использоваться сетевой диск.

Как же работает кластерное ПО?

- Экземпляр кластерного ПО периодически проверяет доступность узлов кластера.
- Если узел недоступен, то экземпляр, обнаруживший недоступность, принимает решение о том, что нужно сделать для восстановления работоспособности кластера, и сообщает о своём решении остальным экземплярам. Это сообщение может выглядеть как запуск выбора лидера в кластере с целью собрать большинство голосов или как попытка записи во внешнее распределённое хранилище.
- Если голосование (или запись) прошло успешно, то кластер выполняет программу переключения.

Рассмотрим ситуацию, когда экземпляр кластерного ПО, расположенный рядом с репликой, теряет доступ к основному экземпляру БД. Он тут же сообщает о своём намерении перевести реплику в режим основного сервера.

Если потерян основной сервер БД, но кластерное ПО продолжает функционировать, то экземпляр из основного ЦОДа согласится с переездом БД, и решение будет принято. Если экземпляр кластерного ПО из основного ЦОДа недоступен, то сообщение получит только кворумный узел. Его нахождение в третьем ЦОДе даёт возможность получать сообщения, даже если нарушена сетевая связность между основным и резервным ЦОДами — в таком случае именно кворумный узел принимает решение о дальнейшей судьбе кластера. Если кворумный узел получил запрос только от одного экземпляра, то он подтверждает запрос, а если от двух, то, вероятнее всего, он поддержит сервер, являющийся основным в данный момент.

Для корректной работы кластера необходимо, чтобы связь экземпляра кластерного ПО с экземпляром СУБД, за который он отвечает, была надёжна, а это возможно только в том случае, если кластерное ПО работает на том же сервере, что и экземпляр базы данных. Упрощению развёртывания отказоустойчивого кластера и повышению надёжности в облачных средах способствовало бы встраивание кластерного ПО непосредственно в код базы данных. Первым решением такого класса стал встроенный кластер (BiHA, Built-in High Availability) в СУБД Postgres Pro.

8.3. Распределённый консенсус

Один из важнейших алгоритмов — достижение **распределённого консенсуса**, то есть **согласия всех узлов кластера относительно решения по какому-либо вопросу**.

В процессе работы распределённых систем постоянно возникают задачи, требующие наличия одинаковой информации на всех узлах. Например, в разделе, посвящённом шардированию, мы говорили о «честном распределении» фрагментов между узлами системы, которое не может быть одинаково рассчитано несколькими узлами независимо друг от друга. Другим примером задачи, требующей синхронизации, может быть изменение схемы данных в реляционной базе.

Очевидно, что должен быть какой-то авторитетный источник, которому доверяют все узлы, но проблема в том, что этот источник становится единой точкой отказа. Алгоритм распределённого консенсуса позволяет равноправным узлам кластера договариваться в отсутствие внешнего арбитра, который мог бы взглянуть на кластер со стороны.

Первым алгоритмом распределённого консенсуса стал Paxos, названный так в честь греческого острова, где якобы функционировал парламент, использовавший описанную в алгоритме систему голосования. Работа над этим алгоритмом началась ещё в 80-е годы XX века, в 90-е годы он уже начал использоваться, а опубликован был только в 1998 году (Leslie Lamport, «The part-time parliament»). Однако наибольшую известность получила статья «Paxos простыми словами» («Paxos Made Simple»), опубликованная в 2001 году.

Распределённый консенсус определяется так:

1) может быть выбрано только то решение, которое кем-то предложено;
2) должно быть выбрано только одно решение;
3) окружающие узнают о принятом решении только тогда, когда оно действительно принято.

В этой же статье сформулированы предположения о среде, в которой должен быть достигнут консенсус:

- каждый узел может отправить сообщение любому другому узлу;
- узлы работают с разной скоростью, глобальных часов не существует; имея два независимых события, мы не можем сказать, какое из них произошло раньше;
- время доставки сообщений между узлами не определено — сообщение может доставляться сколь угодно долго;
- в системе отсутствуют «византийские отказы», то есть сообщения могут теряться или дублироваться, но не могут быть искажены;
- любой узел может быть потерян;
- потерянный узел может быть перезапущен (а может и не быть);
- узел запоминает своё состояние между перезапусками.

И вот теперь настало время перейти к описанию алгоритма.

Paxos

Любой из узлов кластера может стать инициатором решения. Инициатор пытается заручиться поддержкой большинства голосующих узлов, то есть сформировать кворум. Если кворум собран, то решение ставится на голосование. После того как большинство участников проголосовали за это решение, оно считается принятым.

В процессе принятия решения узлы играют следующие роли:

- инициатор (proposer) — узел, предлагающий решение;
- избиратель (acceptor) — узел, участвующий в голосовании;
- слушатель (learner) — узел, не участвующий в голосовании, но сохраняющий у себя результат голосования.

По каждому вопросу может быть предложено несколько решений, поэтому узел может играть одновременно несколько ролей даже в голосовании по одному вопросу.

Чтобы не путать предложения, реализация должна обеспечить их нумерацию, причём номер должен постоянно возрастать, чтобы упорядочить предложения (напомню, временны́е метки не годятся для упорядочивания, потому что по условиям задачи общие часы в кластере отсутствуют).

Каждый узел хранит следующие значения:

- (n_a, v_a) — максимальный номер предложения, которое этот узел принял, и принятое решение;
- n_h — максимальный номер предложения, которое поступало этому узлу.

Чтобы предложить решение, узел-инициатор выбирает номер $n > n_h$ и рассылает приглашение (prepare) узлам-избирателям.

Получив приглашение, узел-избиратель сравнивает его с хранящимися у него значениями. Если он уже получал предложение с бо́льшим номером, то есть $n_h > n$, то он может не отвечать, но обычно в целях оптимизации посылает инициатору отказ. Если же полученное предложение новее всех, полученных ранее, то узел-избиратель обещает (promise) поддержать инициатора и не принимать предложений с номером меньше, чем n (то есть отныне полагает $n_h = n$). Кроме того, в ответ на приглашение узел-избиратель отправляет номер принятого предложения и значение, которое было принято, — (n_a, v_a).

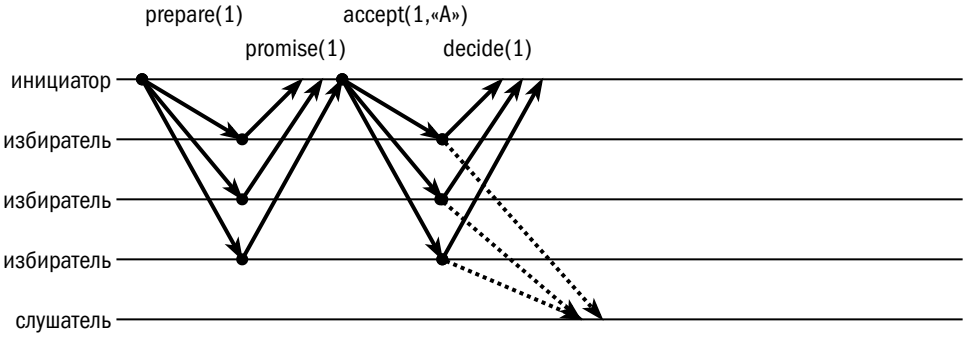
Если узлу-инициатору удалось заручиться поддержкой большинства избирателей, то он отсылает принятое решение на утверждение (accept).

Если кто-то из избирателей до этого уже принял решение, то инициатор должен выбрать максимальное из полученных n_a и отослать на утверждение именно то решение v_a, которое было принято в этом раунде. В противном случае инициатор может выбирать решение произвольно.

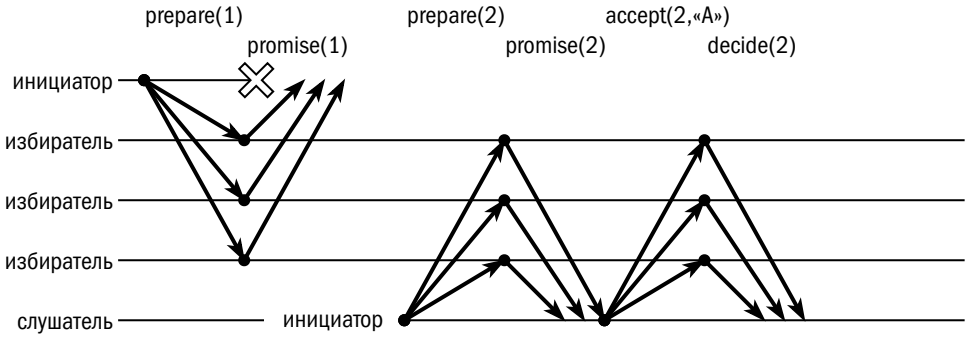
Получив запрос, узел-избиратель обязан его подтвердить (decide), если только за время между предложением и запросом подтверждения не пришло новое предложение с новым номером.

После того как решение принято, оно должно быть донесено до слушателей (learners). Классический алгоритм предполагает, что одновременно с отправкой подтверждения инициатору избиратель отправляет уведомление всем слушателям, и когда слушатель получит уведомления от большинства избирателей, он понимает, что решение принято. Однако в такой схеме количество отправляемых уведомлений равно произведению количества избирателей на количество слушателей, поэтому обычно применяют всевозможные оптимизации — например, избиратели отправляют уведомления только одному слушателю, а тот уже информирует остальных. Очевидно, что при этом надо предусмотреть отказ слушателя.

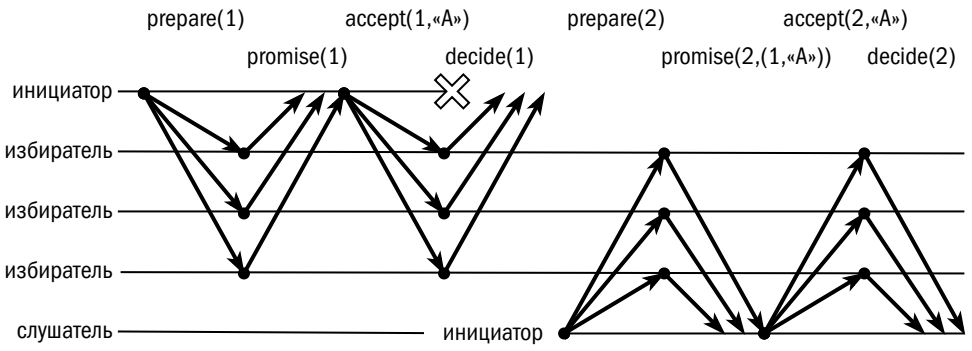
Рассмотрим работу алгоритма Paxos на примерах. Самый простой пример — когда узлы надёжны и все коммуникации проходят успешно. Здесь показаны все три типа узлов — инициатор, избиратели и слушатель. На дальнейших иллюстрациях слушатели не показаны:



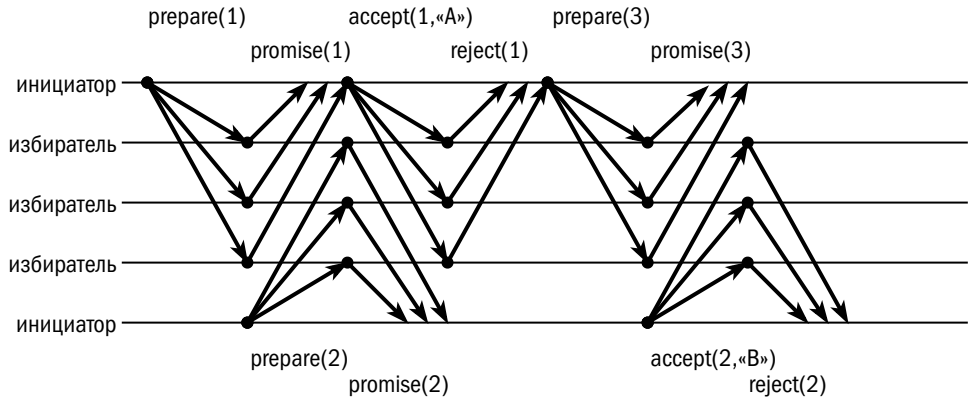
Если в процессе переговоров будет потерян голосующий узел, то это не повлияет на результаты голосования, т. к. для принятия решения достаточно большинства. Если же будет потерян узел-инициатор, то инициатором может стать любой другой узел:



Может случиться и такое, что узел-инициатор будет потерян после отправки решения на голосование. Если хотя бы один узел из тех, что участвовали в первом раунде голосования, примет участие во втором, то будет принято то же самое решение, которое было предложено в первый раз. Узел-избиратель сообщит новому инициатору, что голосование по этому вопросу уже было, и передаст принятое решение:



Теоретически возможна ситуация, когда несколько инициаторов работают одновременно и решение не принимается:



На практике проблема «гонок» (races) решается по-разному разными реализациями Paxos, которых насчитывается не один десяток. Простейшее (но далеко не самое эффективное) решение заключается во введении случайных задержек перед отправкой предложения.

Multi-paxos

Paxos-группа, то есть набор узлов, достигающих консенсуса при помощи алгоритма Paxos, представляет собой ячейку, в которую можно единственный раз записать неизменяемое значение. Реальные базы данных всё же предназначены для хранения изменяемых объектов. Чтобы объекты в кластере можно было изменять, результаты консенсуса рассматриваются как операции или элементы журнала.

Выполнение всех шагов алгоритма Paxos для каждой операции было бы весьма дорого, поэтому в целях оптимизации применяется вариант Paxos с выделенным узлом-инициатором. На первом шаге голосующие узлы вместе с обещанием принять предложение инициатора выдают ему мандат (lease) на последующие предложения в течение некоторого времени. Если узел-инициатор хочет записать следующее значение, то он может сразу приступать к стадии рассылки значения (accept). Таким образом количество сообщений между узлами сокращается.

Мандат может продлеваться до тех пор, пока узел-инициатор функционирует. Если узел-инициатор покидает кластер или не предпринимает попыток записи в течение действия мандата, следующая запись вновь начинается с фазы подготовки (prepare).

Raft

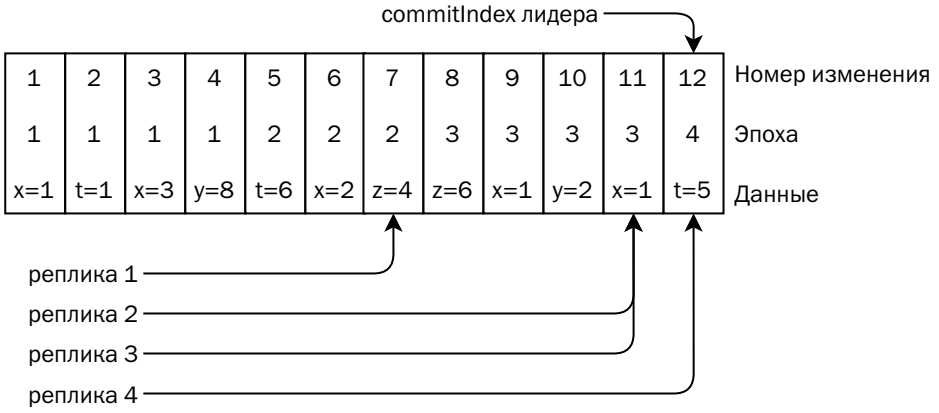
Долгое время слова «распределённый консенсус» и «Paxos» были практически синонимами. Однако в 2013 году сотрудники Стендфордского университета Диего Онгаро (Diego Ongaro) и Джон Оустерхаут (John Ousterhout) опубликовали статью «В поисках понятного алгоритма консенсуса» («In Search of an Understandable Consensus Algorithm»). Учёные отмечают два недостатка алгоритма Paxos:

- алгоритм весьма сложен для понимания;
- общепризнанный алгоритм позволяет сохранить единственное значение — не существует единого соглашения, что такое «Multi-paxos».

В качестве альтернативы авторы предлагают собственный алгоритм под названием «Raft». Алгоритм обеспечивает репликацию журнала с узла, являющегося лидером, на остальные узлы кластера, а также предлагает надёжный алгоритм смены лидера.

Работа кластера Raft делится на эпохи (terms), а каждая эпоха состоит из двух фаз — выбора лидера и репликации.

В каждой эпохе один из узлов является лидером — источником изменений. Лидер ведёт журнал изменений и передаёт эти изменения репликам (followers). У каждого изменения есть монотонно возрастающий идентификатор (порядковый номер). Изменение считается подтверждённым (committed) после положительного ответа от большинства реплик.



Каждое сообщение, отправляемое лидером, содержит следующую информацию:

- номер текущей эпохи;
- номер изменения (logIndex);
- идентификатор узла-лидера (leaderId);
- новые данные (entries[]); в целях оптимизации может передаваться несколько записей за раз, но если новых записей нет, то лидер передаёт пустое сообщение (heartbeat);
- номер эпохи и идентификатор записи, предшествующие тем, что передаются в сообщении (prevLogTerm, prevLogIndex);
- идентификатор последней подтверждённой записи (commitIndex).

Получив такое сообщение, реплика проверяет, что её текущая эпоха не новее, чем эпоха лидера, и что у неё уже есть предыдущие изменения. Если какое-то из этих условий не выполнено, то реплика отвечает отказом. Если условия репликации выполнены, то реплика сохраняет у себя новые изменения. Если у неё уже были изменения с теми же индексами, то они перезаписываются данными, полученными от лидера. Кроме того, реплика обновляет значение commitIndex и номер текущей эпохи, если это необходимо. Вместе с результатом репликации реплика возвращает номер своей эпохи.

Если репликация завершилась успешно, то лидер сохраняет у себя идентификатор последнего записанного изменения для данной реплики.

Если реплика ответила отказом из-за того, что у неё ещё нет предыдущих записей (prevLogIndex), но эпоха реплики совпадает с эпохой лидера, то лидер уменьшает prevLogIndex и повторяет попытку репликации.

Возможна ситуация, что узел-лидер потерял часть сообщений (например, вследствие разделения сети), и на остальных узлах уже началась новая эпоха с новым лидером. В этом случае реплика отвечает, что она уже живёт в более новой эпохе, и старый лидер сам становится репликой, прекращая передачу изменений.

Поток сообщений репликации не только передаёт информацию, но и служит сигналом, что с лидером всё в порядке, и он продолжает работу. Таким образом, отдельный поток служебных сообщений с информацией о составе кластера становится не нужен.

Если реплика в течение продолжительного времени не получает сообщений от лидера, она может инициировать процедуру выборов нового лидера. Для запуска голосования она увеличивает на единицу счётчик эпох и отправляет остальным узлам сообщение, в котором содержатся:

- номер новой эпохи;
- идентификатор узла-реплики, который желает сталь новым лидером (candidateId);
- номер эпохи и идентификатор последней журнальной записи, которая у неё есть (lastLogTerm, lastLogIndex).

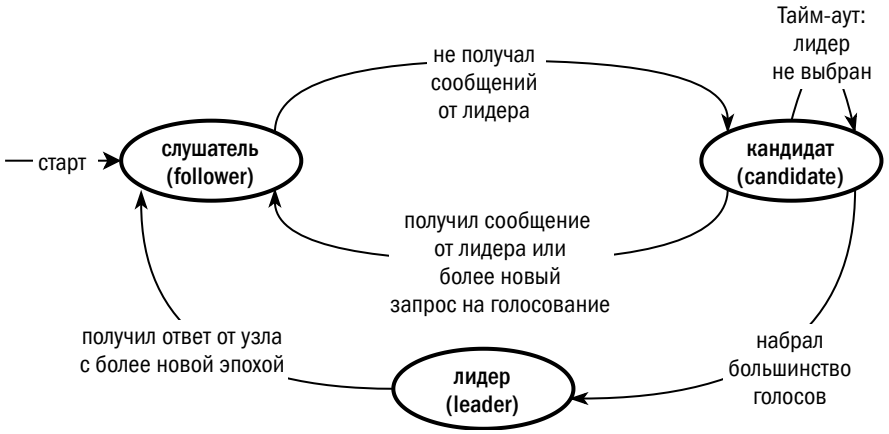
Получив запрос на голосование, узел действует следующим образом:

- если эпоха из сообщения меньше, чем эпоха узла, он голосует против;
- если в этой эпохе узел уже голосовал за другого кандидата, он голосует против;
- если в этой эпохе узел уже голосовал за этого кандидата, он голосует за;
- если в этой эпохе узел ещё не голосовал, то он проверяет, что lastLogIndex больше или равен номеру последнего изменения, полученного этим узлом; если условие выполняется, то узел голосует за и запоминает свой выбор, иначе голосует против.

Узел-кандидат, набравший большинство голосов, становится новым лидером и начинает рассылать сообщения для репликации.

Если ни один из кандидатов не смог набрать большинство голосов, то процесс голосования перезапускается, причём каждый узел перед посылкой нового запроса выдерживает паузу случайной продолжительности. Рисунок из оригинального документа («In Search of an Understandable Consensus Algorithm») на следующей странице показывает жизненный цикл узла — участника Raft-группы.

Raft широко используется в распределённых системах. Например, этот алгоритм используется в хранилищах критичной информации о конфигурации кластера, таких как Consul (consil.io) или etcd (etcd.io). Raft реализован также



в платформах ScyllaDB, Tarantool, Picodata. Брокер Kafka отказался от внешнего хранилища конфигурации в Apache Zookeeper в пользу KRaft — собственной реализации протокола Raft.

Широкое использование алгоритма позволило обнаружить и его слабые стороны. Например, при частичной потере связности могут возникать совершенно ненужные раунды выборов, и лидер в кластере будет «мерцать».

Если кластер разделился, а потом в большей из получившихся частей отказали несколько узлов, то ни одна из частей кластера не сможет собрать кворум и продолжить работу. Администраторы по ошибке могут запустить вручную две части (или более), и при восстановлении связи между частями журналы на узлах, находившихся в разных частях, будут отличаться.

Популярная статья инженера компании MongoDB Хенрика Инго (Henrik Ingo) «Четыре доработки алгоритма Raft» («Four modifications for the Raft consensus algorithm») предлагает простые и эффективные решения данных проблем:

1. Помимо прочих служебных полей, в журнале появляется уникальный идентификатор базы данных.
2. Начиная работу, узел либо присоединяется к кластеру, либо явно инициализирует новый кластер, генерируя при этом идентификатор базы данных. Если при разделении кластера на две части, ни в одной из которых нет кворума, администраторы запустят несколько кластеров вручную, они вынуждены будут сгенерировать новые идентификаторы для каждого нового кластера, и при попытке объединить их модифицированный Raft вернёт ошибку.

3. Если кто-то пытается инициировать процедуру выборов, то узел, регулярно получающий сообщения от лидера, голосует против. Тем самым не алгоритм избегает «мерцания» лидера при частичном нарушении связности кластера.
4. Перед тем как предлагать себя в качестве лидера кластера, узел проводит предварительное голосование. Если позиция журнала на узле, получившем такой запрос, ушла дальше, чем у «предварительного» кандидата, он голосует против, возвращая номер текущей эпохи. Тем самым решается проблема «инфляции эпохи», когда узел с периодически отказывающим сетевым соединением пропадает из кластера, а вернувшись, каждый раз начинает новый раунд выборов с увеличением номера эпохи.

Zookeeper Atomic Broadcast

Apache Zookeeper — распределённое хранилище в памяти для критичной информации о конфигурации кластера, конкурент etcd и Consul. Для достижения консенсуса между узлами Zookeeper используется собственный алгоритм — Zookeeper Atomic Broadcast. Он похож на Raft, но его разработка была начата в 2007 году — за шесть лет до публикации Raft. Эксперты по распределённым системам, создатели инструментария Jepsen рекомендуют: «Используйте Zookeeper. Это зрелый и хорошо спроектированный продукт, проверенный временем».

Работа Zookeeper, как и работа Raft-кластера, делится на эпохи. Протокол ZAB (Zookeeper Atomic Broadcast), в свою очередь, состоит из трёх фаз: выбора лидера, синхронизации и репликации.

Точный алгоритм выбора потенциального лидера в документации не описан, сказано лишь, что «узлы прикладывают все усилия, чтобы выбрать лидера с максимально длинной историей». После того как потенциальный лидер найден, он ожидает, что все реплики пришлют информацию о себе, включая номер эпохи, которую считают текущей. Набрав кворум, лидер объявляет начало новой эпохи и рассылает её номер репликам. Если номер новой эпохи больше, чем номер эпохи реплики, реплика отсылает подтверждение. Когда большинство реплик подтверждают начало новой эпохи, лидер проверяет, что не существует реплик (включая тех, что не прислали подтверждения), у которых уже начата та же эпоха и есть более новые изменения, чем у лидера. Если это так, то лидер новой эпохи считается выбранным, иначе процедура выборов повторяется.

После того как лидер выбран, начинается синхронизация. До окончания синхронизации лидер буферизирует все новые изменения, не рассылая их репликам. Для каждой реплики выполняется одна из операций: передача снимка (`SNAP`), если отставание слишком велико; обрезка истории (`TRUNC`), если реплика убежала вперёд, но лидер решил не записывать какие-то изменения; досылка последних изменений (`DIFF`), если отставание невелико и выгоднее прислать изменения, чем полный снимок состояния. По окончании синхронизации реплика переключает номер эпохи и отсылает подтверждение лидеру. Когда большинство реплик подтвердили окончание синхронизации, лидер считает новую эпоху начатой и приступает к рассылке буферизированных изменений.

Так же, как и в Raft, лидер рассылает изменения репликам и, когда большинство подтверждает приём, считает изменение подтверждённым (committed). Если изменений в течение продолжительного времени нет, лидер шлёт пустое сообщение (ping). До тех пор, пока реплики получают сообщения от лидера, они подтверждают все его изменения и не пытаются выбрать нового лидера.

Другие алгоритмы распределённого консенсуса

Во многих распределённых системах реализованы собственные алгоритмы распределённого консенсуса. В отличие от Paxos и Raft, сходимость которых логически доказана, критерием их верности, как и для ZAB, является практика.

Для тестирования распределённых систем широко применяется инструментарий Jepsen (jepsen.io) совместно со средствами деструктивного тестирования, моделирующими разделение сетей, потерю пакетов, рассинхронизацию часов и прочие неполадки, которые могут стать причиной некорректного поведения распределённых систем. Jepsen моделирует различные операции записи и чтения, направляя их на произвольные узлы кластера, и проверяет, выполняются ли в результате этих операций гарантии, заявленные разработчиками тестируемых систем.

MongoDB — распределённая документоориентированная СУБД. Для репликации между узлами используется собственный протокол. К особенностям протокола репликации MongoDB можно отнести:

- изменения сначала применяются к состоянию узла-лидера, а потом реплицируются; в Raft изменения применяются только после того, как они подтверждены большинством слушателей;

- при записи или чтении можно указывать, сколько узлов должно подтвердить операцию;
- возможна каскадная репликация, то есть некоторые реплики могут получать информацию не от лидера, а от других реплик.

Тестирование MongoDB при помощи Jepsen выявило множество ситуаций, когда данные в кластере оказывались несогласованными и база нарушала гарантии целостности. Анализ показал, что основные ошибки, которые ведут к таким нарушениям, таковы:

- разделение репликации и heartbeat-сообщений на два разных потока;
- использование временны́х меток вместо абстрактных «эпох» — из-за этого реплики принимали изменения от узлов, которые считали себя лидерами, но лидерами уже не являлись;
- чрезмерно сложная процедура выбора лидера.

Изначальный протокол репликации был заменён другим, похожим на Raft, в релизе 3.2, а к релизу 3.4 он был стабилизирован и позволил достичь высокой надёжности при обработке однодокументных транзакций.

Cassandra, «распределённое хранилище колонок», позиционируется как доступная (AP) система, поэтому запись должна быть выполнена всегда. В отличие от систем, где есть выделенный лидер, в Cassandra запрос на запись может обработать любой узел, включая узлы, вообще не хранящие копию изменяемых данных. Узел, поддерживающий клиентский сеанс, отправляет запрос на запись всем узлам, хранящим нужные данные, и возвращает подтверждение клиенту, когда получит требуемое количество подтверждений. «Требуемое количество» — один из параметров каждого запроса, который может принимать различные значения — от `ANY` (запись хотя бы на один произвольный узел) до `ALL` (запись на все узлы, где должны храниться записываемые данные).

В результате такого подхода в данных, хранящихся в БД Cassandra, возникают самые разные аномалии, включая ситуации, когда в объекте фиксируется часть изменений из одной транзакции и часть — из другой. В качестве компенсирующей меры Cassandra применяет механизмы восстановления целостности, о которых рассказано в разделе 8.5 «Компенсация несогласованности».

В Cassandra 2.0 появился механизм «легковесных транзакций» (lightweight transactions), который гарантирует целостность операций compare-and-set (хотя Jepsen-тесты показывают, что эти гарантии не всегда выполняются). Для поддержания согласованности данных используется протокол, основанный на Paxos, но требующий четыре коммуникации между узлом-инициатором транзакции

и узлами, хранящими данные: подготовка (prepare/promise), чтение (read/result), запись (propose/accept) и подтверждение (commit/ack).

Разработчикам, использующим Cassandra, рекомендуется:

- по возможности избегать конкурентных обновлений, используя платформу для приложений с высокой интенсивностью записи, например, для хранения журналов операций;
- использовать безопасные типы данных (CRDT, Conflict-free Replicated Data Types).

8.4. Распределённые транзакции

Понятие «распределённой транзакции» появилось едва ли не раньше, чем понятие «распределённого консенсуса». В 1986 году инженеры компании IBM опубликовали описание распределённой системы R*, где уже называли протокол двухфазной фиксации (two-phase commit, 2PC) «хорошо известным», а в 1991 году консорциум X/Open опубликовал отраслевой стандарт обработки распределённых транзакций «Distributed Transaction Processing: The XA Specification».

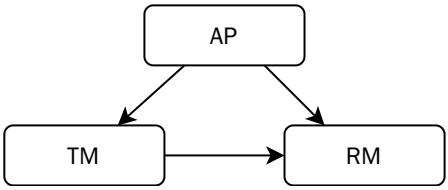
Протокол двухфазной фиксации долгое время был фактически синонимом понятия «распределённая транзакция», да и до сих пор он не утратил своих позиций.

Протокол двухфазной фиксации

Под транзакцией в документе X/Open понимается какое-либо изменение ресурса. Ключевыми признаками транзакции являются возможность отката и перевод ресурса из одного согласованного состояния в другое, то есть атомарность и согласованность, A и C из знаменитой аббревиатуры ACID.

Участники распределённой системы делятся на три типа:

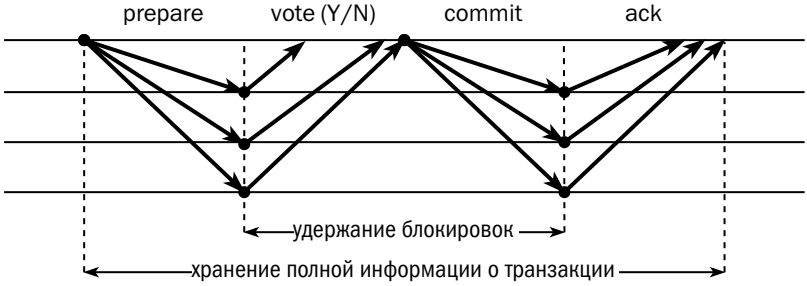
- приложение (application program) — соответствует интуитивному понятию «приложения», то есть нечто, инициирующее процессы, приводящие к изменению данных;
- владелец ресурса (resource manager) — участник, предоставляющий интерфейс доступа; владельцем ресурса может быть, например, база данных, очередь или файловая система, а ресурсом — таблица, топик, файл;
- координатор (transaction manager) — участник, управляющий транзакциями с использованием нескольких ресурсов, то есть принимающий решение, подтвердить (commit) или откатить (rollback) распределённую транзакцию.



Приложение может обращаться напрямую к любому ресурсу через интерфейс его владельца, но если необходимо выполнить транзакцию, затрагивающую несколько ресурсов, управляемых разными владельцами, то эту работу приложение делегирует координатору. Координатор должен генерировать идентификаторы распределённых транзакций, обращаться к ресурсам в рамках распределённой транзакции и завершать распределённые транзакции. Кроме того, в координаторе могут быть реализованы некоторые оптимизации протокола двухфазной фиксации — например, однофазная фиксация, если в транзакции были изменены ресурсы единственного владельца.

Процесс фиксации транзакции состоит из следующих шагов:

- Координатор отправляет всем участникам команду «приготовиться» (prepare).
- Участник, получив такую команду, должен убедиться, что все данные доступны, блокировки получены и т. д. Если участник может гарантировать корректное завершение транзакции, то он отправляет в ответ подтверждение, в противном случае — отказ. Отправив отказ, участник может сразу же откатить свою транзакцию.
- Если все участники подтвердили готовность, то координатор командует всем участникам подтвердить (commit) транзакцию; если же хотя бы один участник ответил отказом, координатор командует откатить (rollback) транзакцию.
- Получив команду координатора, каждый участник подтверждает либо откатывает транзакцию.



Подтверждение распределённой транзакции можно рассматривать как задачу достижения консенсуса по вопросу о том, подтверждать или откатывать транзакцию. При этом координатору не важны ни содержимое транзакции, ни способ её подтверждения.

К достоинствам протокола двухфазной фиксации можно отнести его простоту, к недостаткам — длительные блокировки и неустойчивость к сбоям.

На рисунке выше показано время, в течение которого участники вынуждены удерживать блокировки (locks), — от начала стадии подготовки до подтверждения. Оно состоит из времени обработки транзакции самым медленным участником и времени на отправку и получение сообщения.

Описываемый стандарт предполагает, что каждый участник имеет доступ к надёжной системе хранения данных, где сохраняет своё состояние. После любого сбоя участник восстанавливается, читая состояние с диска. Сбои обрабатываются следующим образом:

- Если ресурс отказал, не успев ответить на подготовительный запрос, координатор, не дождавшись подтверждения, откатывает транзакцию по тайм-ауту.
- Если ресурс подтвердил готовность, но позже отказал, то координатор подтверждает транзакцию. Данные отказавшего ресурса остаются некорректными, и владелец этого ресурса несёт ответственность за то, чтобы при восстановлении привести их в согласованное состояние.

В любом случае координатор обязан хранить исход каждой транзакции (trx info) до тех пор, пока не получит подтверждение от всех её участников.

Реакция на отказ координатора в стандарте не описана, поэтому в каждом продукте, реализующем двухфазный протокол, обработка отказов координатора реализована по-своему.

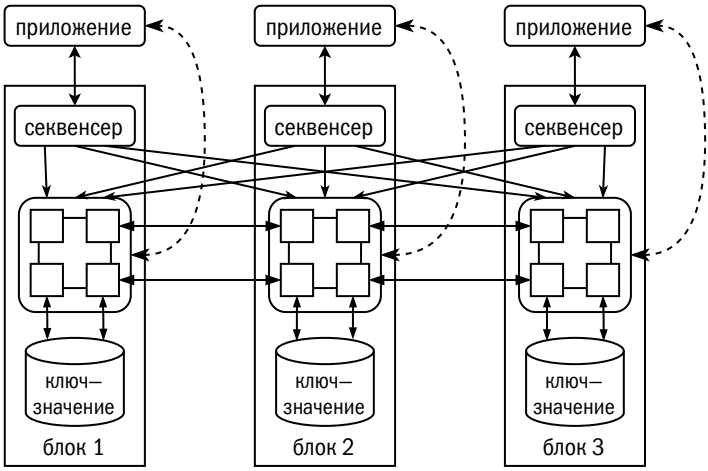
Детерминированные транзакции (Calvin transactions)

В 2012 году учёные Йельского университета опубликовали новый подход к реализации распределённых транзакций. В его основу положена идея формирования журнала транзакций и подачи его на вход исполнителю, вместо того чтобы позволить ему самому формировать журнал. Подход был назван в честь французского богослова Жана Кальвина (Jean Calvin), считавшего, что все события человеческой жизни предопределены Богом — ведь если журнал сформирован заранее, то и результат исполнения транзакций из этого журнала предопределён.

Из этой идеи следуют интересные выводы:

- один и тот же журнал можно подавать на вход любому количеству исполнителей, и данные в базах, изменяемых этими исполнителями, будут одинаковы;
- если экземпляр обычной БД записал транзакцию в журнал, значит, эта транзакция выполнена; но если исполнитель получает журнал на вход, то возможна ситуация, когда он не сможет выполнить транзакцию.

На рисунке ниже изображена машина детерминированных транзакций:



Машина состоит из секвенсера (sequencer), который упорядочивает транзакции, исполнителя (executor) и хранилища, поддерживающего интерфейс «ключ—значение». Каждый из этих элементов масштабируется горизонтально и независимо друг от друга, но в референсной реализации база разбивается на блоки, содержащие все три компонента.

Обработка транзакций происходит следующим образом:

1. Клиентское приложение преобразует описание транзакции на языке SQL или процедурном языке в последовательность обращений (чтения или записи) к конкретным объектам. Для этого запрос передаётся исполнителям (на рисунке взаимодействие показано пунктиром). Исполнители возвращают приложению последовательность обращений и номера версий объектов, которые были прочитаны и должны быть изменены. Если БД предоставляет приложению интерфейс «ключ—значение», то этот этап пропускается — приложение уже знает, с какими объектами в какой последовательности оно будет работать.

2. Приложения передают полученные последовательности секвенсерам. Секвенсеры делят время на эпохи, в течение которых транзакции накапливаются, а по окончании накопленные пакеты передаются исполнителям. Продолжительность эпохи составляет около 10 мс, что позволяет нескольким секвенсерам отправлять пакеты транзакций одновременно и не зависеть от точности синхронизации часов. Задача секвенсера — упорядочивание транзакций: сначала по номерам эпох, а внутри эпохи, — например, по идентификатору клиентского подключения.
3. Исполнители начинают обрабатывать транзакции по порядку. Сначала каждый исполнитель считывает все объекты, которые должны быть прочитаны в транзакции, и сравнивает их версии с теми, что записаны в журнале. О результатах сравнения исполнитель информирует тех исполнителей, которые должны изменить данные в этой транзакции.
4. Если пишущие исполнители видят, что версии объектов не изменились со времени подготовки транзакции, то они выполняют требуемые изменения, иначе — нет.

Рассмотрим обработку транзакций на примере.

Пусть выполняются два экземпляра приложения, предназначенных для перевода денег со счёта на счёт. Алгоритм описан так: «если на счёте-источнике достаточно денег, то уменьши остаток на счёте-источнике и увеличь остаток на счёте-приёмнике». В одной транзакции Василий переводит 1000 рублей Марии, а в другой — 1500 рублей Анне. Все три счёта находятся в разных блоках. На момент начала исполнения транзакций данные выглядят так:

Мария

id	amount	version
42	5000	91

Василий

id	amount	version
14	10000	89

Анна

id	amount	version
120	3000	121

Приложение читает данные и преобразует транзакции в следующую последовательность действий:

- перевод от Василия Анне — `R(14,89)`, `R(120,121)`, `W(14,8500)`, `W(120,4500)`;
- перевод от Василия Марии — `R(14,89)`, `R(42,91)`, `W(14,9000)`, `W(42,6000)`.

Здесь псевдофункция `R()` считывает объект и возвращает истину, если объект найден и его версия совпадает с ожидаемой, и ложь — в противном случае. Псевдофункция `W()` записывает данные с требуемым ключом, и если объект с таким ключом уже был, то увеличивает его версию на 1. Никакие блокировки, в том числе оптимистические, не требуются, поскольку чтение и запись выполняются последовательно в одном потоке и между ними гарантированно не будет никаких других операций.

Предположим, что транзакции переданы исполнителям в таком порядке, как показано в списке. Тогда оба чтения первой транзакции будут выполнены успешно, о чём блок «Василий» сообщит блоку «Анна», а блок «Анна» — блоку «Василий». После этого каждый блок внесёт изменения в свои данные, и они примут следующий вид:

Мария

id	amount	version
42	5000	91

Василий

id	amount	version
14	8500	90

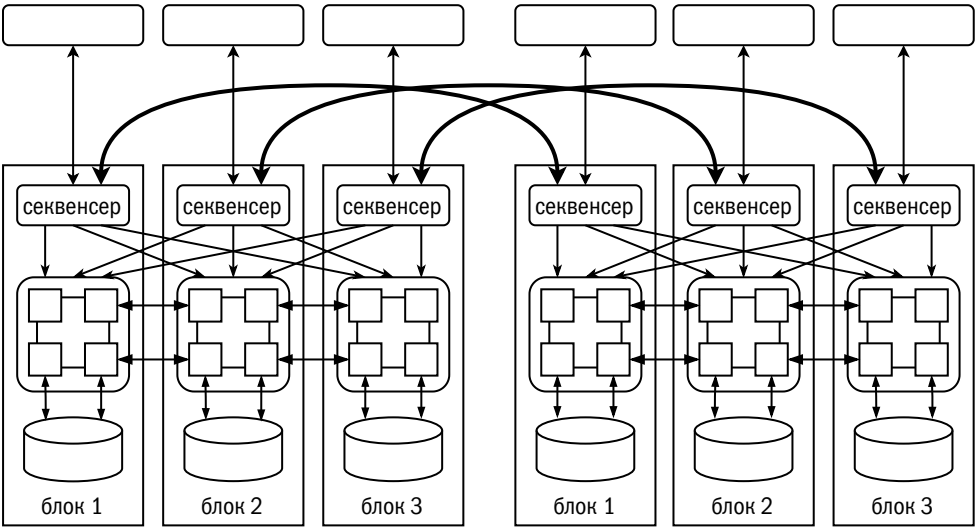
Анна

id	amount	version
120	4500	122

При выполнении второй транзакции чтение в блоке «Василий» закончится ошибкой, поскольку версия данных новее, чем та, что указана в журнале. Блок «Василий» сообщит об ошибке блоку «Мария» и закончит обработку транзакции, не изменяя данные. Блок «Мария», получив сообщение об ошибке, также не будет менять свои данные. Транзакция будет возвращена на повторную обработку, после которой примет вид: `R(14,90)`, `R(42,91)`, `W(14,7500)`, `W(42,6000)`.

Несомненным достоинством механизма детерминированных транзакций является практически неограниченная возможность горизонтального масштабирования с сохранением ACID-гарантий. Правда, система хорошо масштабируется только в том случае, если разные приложения работают с разными данными. При наличии «горячих» записей, за которые идёт высокая конкуренция, возможен резкий рост накладных расходов на повторное планирование транзакций.

При необходимости растянуть кластер Calvin на несколько ЦОДов достаточно синхронизировать секвенсеры:



Поскольку секвенсеры оперируют абстрактными «эпохами» вместо временных меток, точная синхронизация времени не требуется.

В то же время у алгоритма детерминированных транзакций есть и очевидные недостатки:

- Ни одна транзакция не может длиться меньше, чем продолжительность эпохи. В референсной реализации длина эпохи составляет 10 мс, а добавление к секвенсеру синхронной Paxos-реплики увеличивает задержку до 100 мс, если реплика в том же ЦОДе, и до 500 мс, если реплика в другом ЦОДе.
- Все исполнители должны работать с одинаковой скоростью, и нагрузка на них должна быть примерно равной. В обычной базе данных транзакция может попытаться обогнать другую транзакцию, ожидающую блокировки, здесь же порядок предопределён, и транзакция обязана ждать, пока не завершатся все предыдущие транзакции.

Детерминированные транзакции реализованы в таких платформах, как FaunaDB и YDB.

Интересное развитие механизм детерменированных транзакций получил в алгоритме **Accord**, реализованном в пятом релизе[1] СУБД Apache Cassandra.

[1] На момент написания книги Apache Cassandra 5.0 находится в стадии prerelease.

Вместо номеров эпох используются временны́е метки. Метка представляет собой кортеж, состоящий из показаний часов реального времени, логического времени и идентификатора процесса, сгенерировавшего метку. Порядок исполнения транзакций однозначно определён, притом что выделенных секвенсеров нет. Любой узел может стать координатором транзакции, и он же назначает этой транзакции временну́ю метку.

Вместе с журналом транзакции, которую надо завершить, координатор рассылает участникам назначенную метку. Если все участники подтвердили, что транзакция может быть выполнена в требуемый момент, координатор посылает всем участникам команду завершения и, после того как операции чтения текущей транзакции завершены, возвращает результат клиенту. Изменения данных выполняются асинхронно.

Кластер определяет разбег часов реального времени на узлах кластера и время передачи сетевых сообщений между узлами. Прежде чем ответить координатору, что транзакция может быть успешно завершена, узел выдерживает паузу, равную сумме разбега часов и сетевой задержки — это гарантирует, что не будет новых транзакций с временно́й меткой меньше, чем у текущей транзакции.

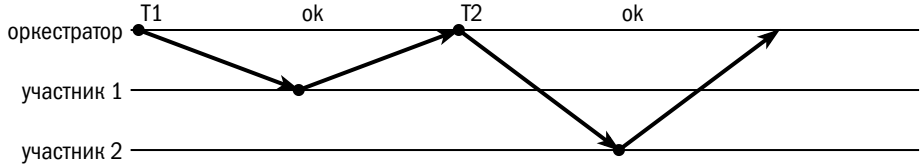
Если узел обнаружил, что существует конфликтующая транзакция (то есть транзакция, обращающаяся к тем же ключам, что и текущая), он возвращает координатору метку конфликтующей транзакции. В этом случае для подтверждения требуется ещё один раунд, во время которого координатор сообщит участникам новую временну́ю метку, которая должна быть больше любой из полученных в первом раунде.

Сага

Сагами (saga) называют древние скандинавские эпосы, описывающие жизнь множества героев в течение многих поколений. Этим же словом профессоры Принстонского университета Гектор Гарсия-Молина (Hector Garcia-Molina) и Кеннет Сэлем (Kenneth Salem) назвали в своей статье, опубликованной в 1987 году, длинные транзакции, затрагивающие множество объектов в разных базах данных. Их предложение заключалось в том, чтобы разбить длинную транзакцию на шаги и выполнять их последовательно или параллельно, но независимо друг от друга.

Каждый шаг саги представляет локальную транзакцию, выполняемую одним из участников. В результате производительность такой системы максимальна,

т. к. в ней нет ни лишних сетевых взаимодействий, ни удержания блокировок дольше, чем это необходимо.



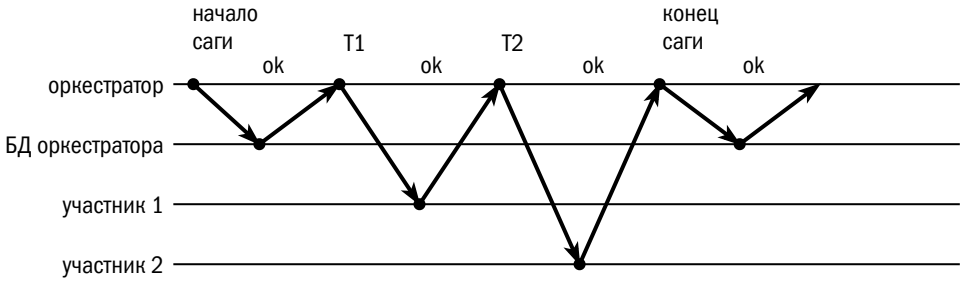
Необходимым условием корректного выполнения саг в публикации названо наличие компенсирующей транзакции T′ для каждой транзакции T. В связи с широким распространением микросервисной архитектуры концепция саг была существенно дополнена и расширена.

Итак, вот основные требования к системе, исполняющей саги:

- оркестратор (координатор саги) должен иметь собственное надёжное хранилище;
- транзакции должны быть идемпотентны, то есть многократное исполнение транзакции должно приводить к тому же эффекту, что и однократное;
- для каждой транзакции T, изменяющей данные, должна быть определена компенсирующая транзакция T′, выполняющая обратные изменения.

Выполнение саги выглядит следующим образом:

1) оркестратор присваивает саге идентификатор и фиксирует в своей внутренней базе её начало;
2) последовательно (возможно, параллельно) выполняет шаги саги;
3) фиксирует в своей внутренней базе окончание саги.



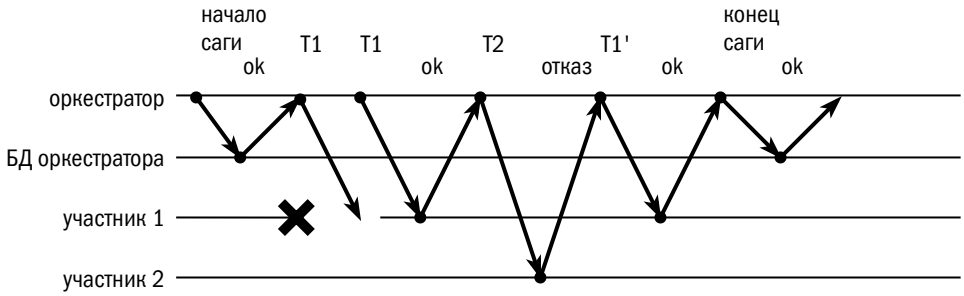
Если какой-то из участников не отвечает, оркестратор может повторить запрос, не оказывая влияния на других участников.

В конечном счёте при исполнении саги возможны всего два типа отказов:

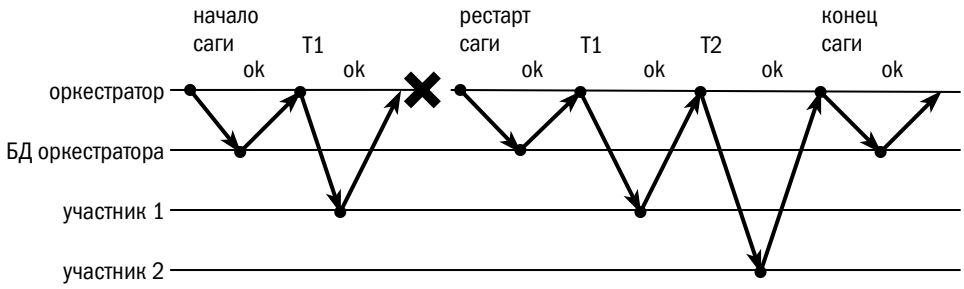
- окончательный отказ одного из участников;
- отказ оркестратора.

При отказе одного из участников оркестратор должен выполнить компенсирующие транзакции для всех транзакций, которые уже завершены.

На рисунке ниже после отказа участника 1 оркестратор предпринимает повторную попытку выполнения транзакции T_1, и новая попытка оказывается успешной. Участник 2 отказывается выполнять транзакцию T_2, когда транзакция T_1 уже зафиксирована участником 1, поэтому оркестратор выполняет компенсирующую транзакцию T_1':



При отказе оркестратора запускается новый экземпляр оркестратора, который считывает из собственной БД информацию о наличии незавершённой саги и выполняет её повторно:



За счёт требования идемпотентности транзакций повторное исполнение транзакции T_1 приведёт к тому же результату, что и однократное её выполнение в ходе нормального исполнения саги.

Помимо фиксации старта и окончания саги, оркестратор может фиксировать в своей БД контрольные точки по результатам выполнения некоторых шагов, чтобы быстрее проводить восстановление при сбоях. Необходимость контрольных точек определяется соотношением стоимости повторного исполнения шага и дополнительного обращения к собственной базе данных оркестратора.

За время использования саг в распределённых системах с медленными и нестабильными каналами был сформулирован ещё ряд требований к саге:

- компенсирующая транзакция также должна быть идемпотентна;
- должна быть возможность прервать основную транзакцию, но не должно быть возможности прервать компенсирующую транзакцию;
- компенсирующая транзакция может быть выполнена не только после основной, но и перед ней.

При всех достоинствах саги необходимо помнить, что она обеспечивает атомарность (с точки зрения клиента, но не с точки зрения оркестратора), согласованность и надёжность, но не обеспечивает изоляцию: параллельные транзакции могут видеть данные в промежуточных состояниях. Если есть возможность обращения параллельно исполняемых саг к одним и тем же данным, необходимо предусмотреть прикладные механизмы изоляции.

8.5. Компенсация несогласованности

Ряд систем — например, Apache Cassandra, Riak KV, Amazon DynamoDB — допускают временную несогласованность, предоставляя пользователям возможность записывать данные даже тогда, когда кворум для записи не собирается. При этом возникает ряд проблем:

- запись на малое количество узлов недостаточно надёжна;
- если кластер разделился, то в нескольких частях кластера могут возникать конфликтующие значения.

Распределённые системы содержат ряд механизмов для решения возникших проблем.

Нестрогий кворум и направленная передача

Если клиент не требует, чтобы данные были записаны на все узлы, а довольствуется кворумом или даже меньшим количеством, но при этом доступных для записи узлов всё равно не хватает, место отсутствующих узлов могут занять произвольные — например, тот узел, через который идёт запись (координатор). Такой кворум называется нестрогим (sloppy quorum).

Узлы-заместители сохраняют у себя данные и идентификаторы тех узлов, которые должны эти данные сохранить. В дальнейшем они периодически проверяют

доступность целевых узлов и при первой возможности передают им сохранённые для них данные. Этот механизм называется направленной передачей (hinted handoff).

Компенсационные механизмы не гарантируют ни надёжности записи, ни корректного чтения:

- Узлы-заместители какое-то время пытаются передать сохранённые данные целевым узлам, но если в течение заданного времени это не удалось, то они попросту стирают чужие данные. Таким образом записанные данные могут быть просто потеряны.
- Читающие сеансы ищут данные на тех узлах, где эти данные должны находиться; если данные находятся на узле-заместителе, то читающий сеанс о них просто не узнает.

Подход с нестрогим кворумом обычно применяется в двухуровневых системах обработки пользовательских запросов, где первый уровень, непосредственно работающий с клиентом, обеспечивает постоянную доступность (но с возможной потерей данных), а второй уровень гарантирует надёжность хранения (но с возможными прерываниями сервиса).

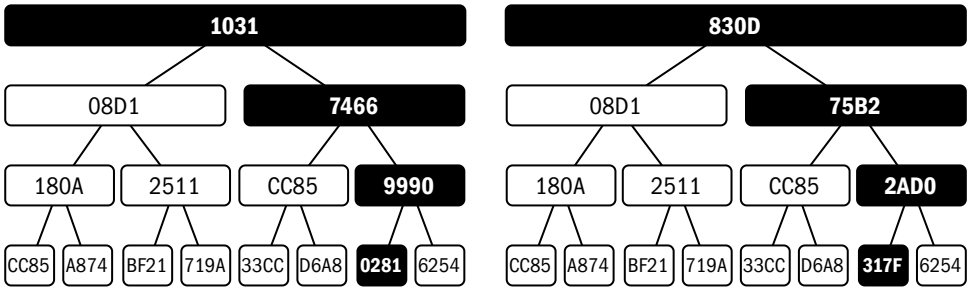
Восстановление данных и дерево Меркла

В платформах, спроектированых с учётом возможной несогласованности данных, реализованы механизмы восстановления (anti-entropy). Восстановление может быть реактивным (по требованию) или проактивным, то есть опережающим.

Реактивное восстановление происходит при чтении данных (read repair). Если читающий сеанс указывает, что для чтения необходим кворум из более чем одного узла, база считывает данные с нужного количества узлов и выбирает из них самые новые. При обнаружении на каком-то узле устаревших данных читающий сеанс перезапишет их актуальной версией.

Проактивное восстановление инициируется или фоновым процессом (Riak KV), или администратором БД (Apache Cassandra). Во время проактивного восстановления сравниваются данные на разных узлах, и если найдены различия, то устаревшая версия заменяется актуальной.

Для уменьшения трафика между узлами используется дерево Меркла (Merkle tree).



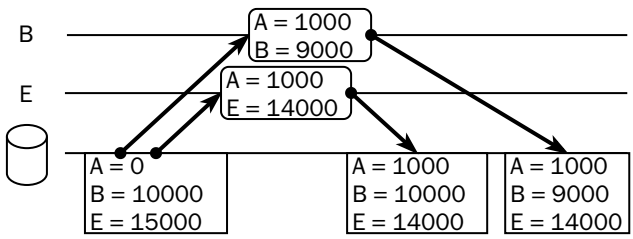
Каждый узел рассчитывает контрольные суммы для небольших наборов объектов, затем контрольные суммы для набора контрольных сумм и так далее, пока не получит одну контрольную сумму. Затем узлы обмениваются деревьями контрольных сумм, что позволяет быстро найти отличающиеся объекты.

Ещё один подход к разрешению конфликтов записи состоит в том, чтобы переложить решение на приложение, использующее базу данных. Так, например, Riak KV можно сконфигурировать, чтобы в случае конфликтующих изменений объекта он сохранял несколько объектов с одним и тем же ключом (siblings). Приложение должно быть готово к тому, чтобы при запросе по ключу получить в ответ не один объект, а несколько.

Версионирование объектов

В монолитных системах, как правило, используется правило «сохраняется последнее значение» (last write wins). Чтобы изменения данных учитывали все предыдущие изменения, приложение должно блокировать данные на всё время выполнения операции.

На рисунке ниже видно, к чему приводит использование правила «сохраняется последнее значение» без блокировки. Пусть, например, Василий и Евгений одновременно пытаются перевести по 1000 рублей Анне:



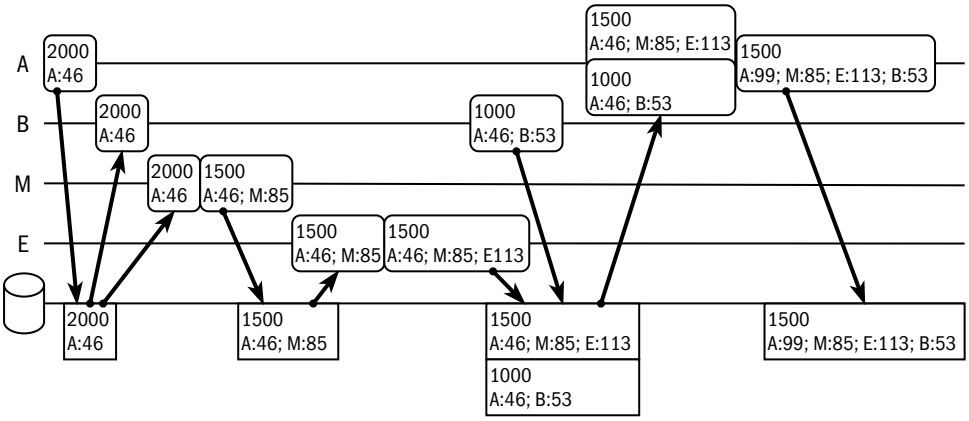
В распределённых системах сохранение последнего значения приводит к тому, что при разрешении конфликта записи выбор актуального значения зависит от настроек часов на разных узлах кластера, то есть фактически является случайным. Важно также помнить, что многие распределённые платформы поддерживают атомарное изменение единственного значения и поэтому просто не имеют функции блокировки.

Сохранять последнее значение имеет смысл только в том случае, если конкурирующие изменения исключены самим дизайном приложения — например, каждый клиент сам генерирует ключи записей и не изменяет данные других клиентов. В большинстве же случаев распределённые платформы предоставляют более сложные механизмы версионирования объектов.

Векторные часы (vector clock) представляют собой массив, количество элементов в котором равно количеству узлов. При выполнении любой операции или при передаче данных между узлами каждый узел увеличивает соответствующее ему значение на единицу. При записи значения в базу вместе с ним записывается и значение векторных часов в момент записи. Если в векторе нет отметки какого-либо узла, считается, что его отметка равна нулю.

Событие считается произошедшим после другого, если все элементы векторных часов этого события не меньше, чем элементы векторных часов другого события, и хотя бы один элемент больше. Если какие-то элементы векторных часов больше, чем соответствующие элементы часов другого события, а какие-то меньше, то считается, что последовательность этих событий не определена.

Пусть Анна, Василий, Мария и Евгений пытаются договориться о сумме взноса на праздник и записывают свои предложения в базу данных. Обратите внимание, что объект считывается из базы вместе с векторной временно́й меткой.

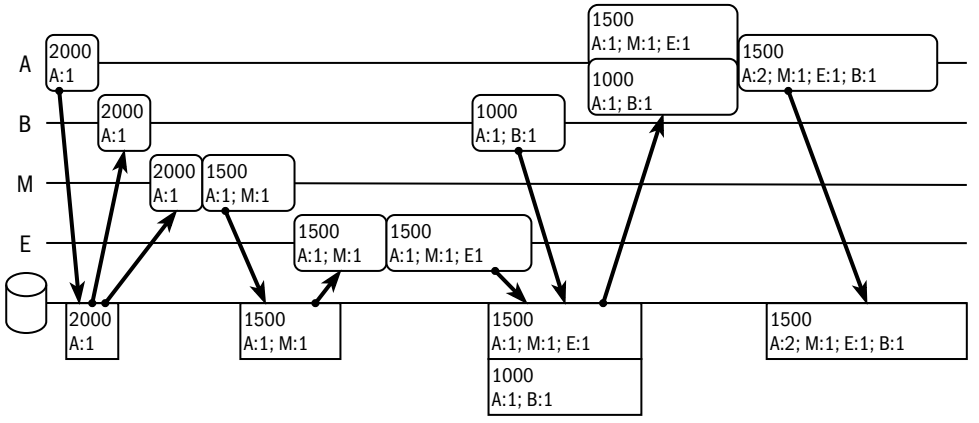


В приведённом примере Анна предлагает собрать по 2000. Мария предлагает собрать по 1500, и поскольку её предложение поступило позднее предложения Анны, данные изменяются. Евгений поддерживает Марию, а вот Василий в ответ на предложение Анны предлагает собрать по 1000. Порядок предложений Евгения и Василия не определён, поэтому база вынуждена сохранить оба предложения. Анна считывает предложение, получая в ответ два значения, и, используя некоторую прикладную логику, вычисляет новое значение и записывает его в базу с таким значением векторных часов, которое гарантированно новее, чем значения часов и предложения Евгения и предложения Василия. Для этого Анна берёт максимальные известные ей отметки Евгения и Василия и добавляет собственную отметку, которая больше всех предыдущих собственных отметок.

Чтобы база данных самостоятельно разрешала конфликты записи, необходимо реализовать логику разрешения на стороне базы в виде хранимой процедуры. Новое значение может быть как одним из конфликтующих, так и результатом каких-либо действий над конфликтующими значениями.

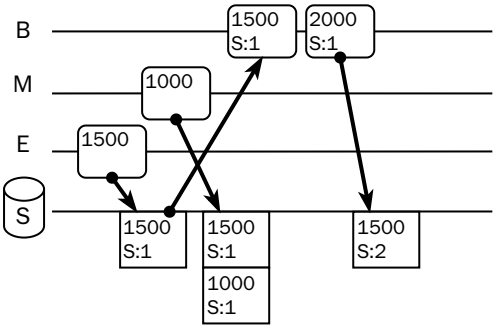
Вектор версий (version vector) устроен точно так же, как и векторные часы, но вместо временны́х меток в метаданных объекта хранятся номера его версий. Такой подход позволяет упорядочивать не любые события, а только лишь версии конкретного объекта, но зато вектор версий значительно компактнее, чем значение векторных часов.

На рисунке ниже изображён тот же пример, что и на предыдущем, но с использованием вектора версий.



В примерах, приведённых выше, временны́е и версионные метки генерируются клиентами. Очевидное достоинство этого подхода в точности, но столь же очевидный недостаток — в плохом масштабировании. К базе данных могут быть подключены тысячи клиентов, и идентификатор клиента при повторном подключении может меняться. Объём метаданных вполне может превысить объём полезных данных.

Если генерировать метки на сервере, то вектор версий сталкивается с неразрешимыми проблемами:

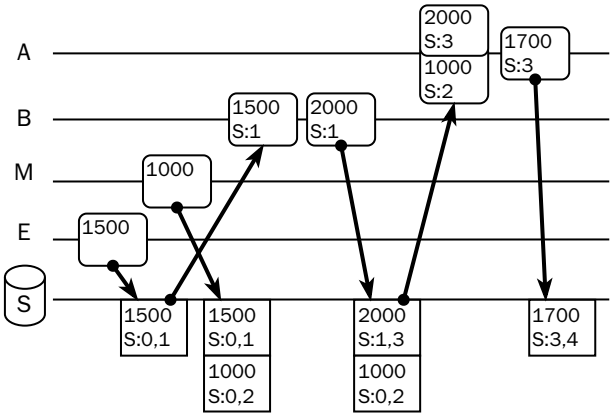


Пусть Евгений и Мария одновременно записывают в базу свои предложения, но изменение Марии попадает на сервер чуть позже. К моменту обработки её запроса на сервере уже есть версия объекта с номером 1. Если сервер присвоит версии Марии номер 2, то получится, что она более новая, чем версия Евгения, но на самом деле это не так. Сервер вынужден присвоить версии Марии номер 1 и сохранить два значения с одинаковым номером версии.

Василий считывает версию Евгения, изменяет её и записывает в базу. Поскольку эта версия основана на версии 1, сервер присваивает ей номер 2 и заменяет этой версией все имеющиеся значения, что некорректно, т. к. версия Василия не учитывает изменения, сделанные Марией.

Для разрешения таких коллизий применяется **точечный вектор версий (dotted version vector)**. Элемент такого вектора состоит из двух чисел — версии, известной на момент начала операции, и новой версии (max_known и new). Один объект считается более новым, чем другой, если его версия на начало операции больше или равна, чем новая версия другого объекта.

Рассмотрим ещё раз предыдущий пример с применением точечного вектора версий.



Евгений и Мария ничего не знали о предыдущем состоянии данных, поэтому max_known их объектов будет установлен в 0, а номера версий (new) — в 1 и 2 соответственно. Василий считывает объект Евгения, имеющий версию 1, которая для Василия станет максимальной из известных. После операции Василия новая версия будет равна 3, поскольку версия 2 уже сохранена Марией. Объект, записанный Василием, перезапишет объект Евгения, потому что его вектор (1,3) больше, чем вектор Евгения (0,1), но не перезапишет объект Марии, поскольку версия объекта на начало транзакции (max_known) Василия меньше, чем новая версия (new) Марии. Когда Анна считывает данные, она видит обе версии. Её max_known устанавливается в максимальный из номеров версий считанных объектов, и когда она записывает свою версию, та затирает обе существующие до этого, т. к. она новее.

8.6. Безопасные типы данных

Ещё один подход к синхронизации данных в распределённых системах — использование безопасных типов данных (Conflict-free Replicated Data Types, CRDT). Безопасными называются структуры данных, для которых определены правила разрешения конфликтов, не зависящие от логики приложения.

Безопасные типы могут базироваться как на репликации состояния (Convergent Replicated Data Type, CvRDT), так и на репликации операций (Commutative

Replicated Data Type, CmRDT). Безопасные типы, основанные на репликации операций, более требовательны к доставке сообщений, поэтому они используются главным образом в системах совместной работы, например wiki-редакторах. Безопасные типы с репликацией состояний используются как в файловых системах (NFS, Coda), так и в распределённых СУБД (Amazon DynamoDB, Riak KV).

Каждый тип описывается следующими характеристиками:

- хранимое значение (payload) и его начальное состояние;
- набор операций, допустимых над объектами этого типа;
- процедура разрешения конфликтов (merge).

Процедура разрешения конфликтов должна быть коммутативной, то есть результат объединения нескольких значений не может зависеть от порядка поступления этих значений.

Счётчик (counter)

Счётчик представляет собой объект, подсчитывающий количество каких-либо событий. Он поддерживает операцию увеличения, а также позволяет узнать текущее значение.

Внутри счётчика хранится массив, количество элементов в котором равно количеству узлов в системе. Каждый узел может увеличивать элемент, соответствующий этому узлу. При запросе состояния значения всех элементов суммируются. Когда узел получает новое состояние объекта от другого узла, он записывает в каждый элемент массива максимальное значение этого элемента из текущего и полученного объектов.

Операция вычисления максимального значения коммутативна, но она не позволяет реализовать уменьшение счётчика. Счётчик, поддерживающий уменьшение, должен содержать два массива — положительный и отрицательный, а его значение будет равно разности сумм элементов этих массивов. Для объединения двух состояний также используется операция вычисления максимального значения — поочерёдно для обоих массивов.

К сожалению, безопасной реализации неотрицательного счётчика (то есть счётчика, значение которого не может быть меньше нуля) не существует. Если нужен такой объект (например, счётчик кредитов в игре), то разработчик должен использовать более строгую модель согласованности либо вводить излишне строгие ограничения — например, разрешать каждому узлу уменьшений не больше, чем увеличений.

Множество (set)

Множество позволяет определить, принадлежит ли ему какой-либо элемент. Кроме того, над множествами определена операция включения, отвечающая на вопрос, является ли одно множество подмножеством другого. Для изменения множества определены как минимум операции добавления и удаления элементов, и эти операции не коммутативны. Следовательно, классическое множество не может быть безопасным типом. Однако существует несколько безопасных типов, которые так или иначе приближаются к реализации множества.

Растущее множество (grow-only set или g-set) — структура, в которой операция удаления вообще не реализована. Операции добавления элементов коммутативны, поэтому проблем с реализацией этой структуры не возникает.

Двухкомпонентное множество (two-phase set или 2P-set) — структура, в которую элементы могут быть добавлены, могут быть удалены, но не могут быть добавлены повторно. Она реализуется как два растущих множества, одно из которых содержит добавленные элементы, а второе — удалённые. Элемент принадлежит двухкомпонентному множеству, если он принадлежит множеству добавленных элементов, но не принадлежит множеству удалённых.

Множество с последним значением (last-write-wins set) — двухкомпонентное множество, где объекты хранятся вместе с временны́ми метками. Если при добавлении (удалении) элемента такой элемент уже существует (удалён), то операция просто обновляет временну́ю метку. То же самое происходит и при объединении состояний — для каждого элемента каждого из двух множеств выбирается максимальная временна́я метка. Элемент принадлежит множеству с последним значением, если он есть среди добавленных, но его нет среди удалённых, или временная метка удалённого элемента меньше, чем временная метка добавленного.

Множество с последним значением полностью реализует классическое множество и является безопасным типом, но только в том случае, если у нас есть надёжный способ генерировать уникальные и монотонно возрастающие временны́е метки на всех узлах кластера.

Наиболее точное приближение множества — **множество с удалением видимых элементов (observed-removed set)**. При добавлении элемента в такое множество генерируется уникальный идентификатор элемента. Один и тот же элемент может присутствовать в множестве несколько раз с разными идентификаторами. При удалении элемента узел удаляет все его вхождения, явно перечис-

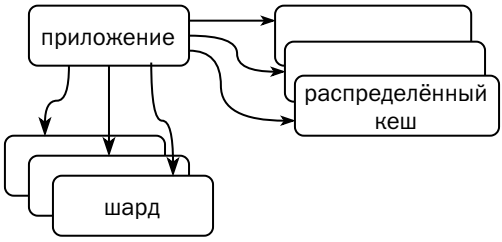
ляя их идентификаторы. Если одновременно с удалением другой узел добавляет этот же элемент ещё раз, то элемент будет присутствовать в множестве, т. к. его идентификатор не попадёт в список удаляемых вхождений.

Множество — структура данных, на базе которой могут быть построены производные структуры — контейнеры, графы и т. д.

8.7. Архитектура распределённых платформ

На рынке присутствует множество распределённых платформ, в архитектуре каждой из которых есть свои особенности. Выбирая платформу для своего проекта, необходимо понимать её устройство, чтобы сформировать адекватные ожидания от платформы и разрабатывать приложения в соответствии с этими ожиданиями.

Приложение как координатор транзакции



Координацию распределённых транзакций можно возложить не только на платформу, но и на приложение. Так, например, в Java координатор распределённых транзакций описан в спецификации Jakarta Transaction API и реализован в пакете `javax.transaction.xa`. Поддержка распределённых транзакций появилась также в .NET Framework версии 2.0.

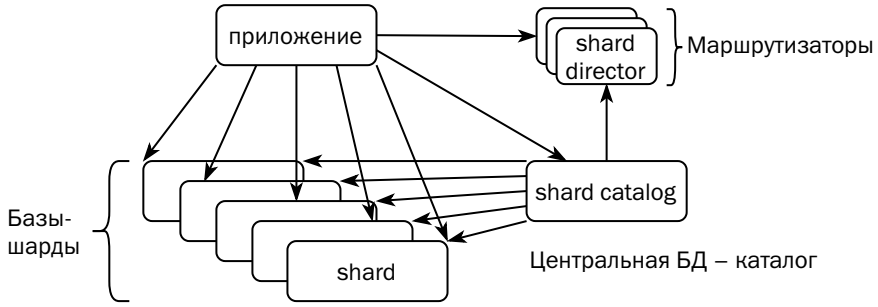
Приложение Oracle Banking Platform построено именно по такому принципу: данные хранятся в независимых БД Oracle, а распределённые транзакции при необходимости координирует WebLogic Application Server. Для защиты от отказа координатора используется распределённый кеш в памяти на базе Oracle Coherence.

Социальная сеть Facebook использует шарды на базе СУБД MySQL. При необходимости согласованно изменить данные в нескольких шардах используется сага,

оркестрируемая приложением Facebook. Для хранения контекста саги используется распределённый кеш на базе Memcached.

Буквальная реализация спецификации X/Open

Платформой, практически буквально реализующей спецификацию X/Open, можно назвать Oracle с опцией Sharding. Платформа состоит из множества баз данных Oracle, каждая из которых хранит свою часть данных (shard). Кроме того, есть специальная база (shard catalog) с информацией о том, какая часть данных в каком шарде хранится.

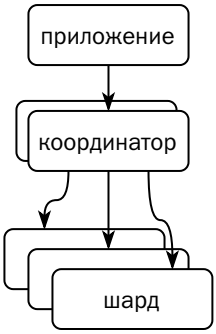


Драйвер Oracle, с которым работает приложение, пытается определить, какой шард может обработать запрос. Если это получается, то приложение обращается напрямую к нужному шарду, а если нет, то к центральному узлу-каталогу, который координирует распределённые транзакции.

Каждый шард хранит уникальные данные, и если он выходит из строя, то эти данные становятся недоступны. Если выходит из строя каталог, то недоступными становятся распределённые запросы и DDL-операции, но одношардовые запросы по-прежнему продолжают работать, т. к. информация о принадлежности данных определённому шарду реплицируется в специальные компоненты-маршрутизаторы (shard director). Для обеспечения отказоустойчивости баз данных в платформе Sharding используются другие опции Oracle — Real Application Cluster (RAC) и Active Data Guard.

Выделенный координатор транзакций

В составе платформы может быть специальный узел или набор узлов, который является точкой входа для приложений. Они принимают пользовательские



запросы, распределяют задачи между узлами, координируют распределённые транзакции, производят окончательную обработку данных и возвращают результат пользователю.

Подход с выделенным координатором хорошо показал себя в системах, рассчитанных на аналитические запросы, таких как Teradata, Greenplum, Microsoft PDW, Netezza (переименованная в IBM PureData System for Analytics, упразднённая в 2017 году и возрождённая под старой торговой маркой в 2019-м), SAP HANA. Объём чтения в таких системах на 3–4 порядка больше, чем объём записи, а транзакции весьма редки, и каждая транзакция обрабатывает большой объём данных.

В Teradata и Microsoft PDW все компоненты реализованы как виртуальные машины, а слой хранения доступен всем серверам: в PDW — в рамках всего кластера, а в Teradata — в рамках группы серверов, «клики» (clique). В PDW работает единственный узел-координатор (control node) и множество узлов-обработчиков (compute node). В Teradata узлов-координаторов (parsing engine) может быть много, как и обработчиков (access module processor).

В Greenplum и Netezza координатор — отдельный узел (master host и просто host соответственно), совмещённый с каталогом, хранящим карту распределения данных по шардам. У координатора есть дублирующий узел, куда синхронно реплицируется состояние. В случае отказа основного узла он берёт на себя роль координатора. Данные распределены по отдельным физическим узлам (segment host и s-blade соответственно), причём каждый узел хранит как свои данные, так и резервную копию части данных других узлов. В случае сбоя узлов-обработчиков координатор меняет карту распределения, а при добавлении узлов запускает процесс балансировки данных и управляет этим процессом.

Citus, в отличие от перечисленных выше платформ, рассчитан на обработку транзакций. Его архитектура похожа на Greenplum — выделенный отказоустойчивый узел-координатор и множество узлов с данными, хранящих, помимо собственных данных, реплики чужих.

Postgres XL, так же как и Citus, является расширением PostgreSQL, позволяющим масштабировать базу на несколько узлов. В составе Postgres XL несколько узлов-координаторов и множество узлов с данными. Поскольку данные интенсивно изменяются, платформе необходимо уделять особое внимание изоляции транзакций, поэтому она включает централизованный компонент — менеджер

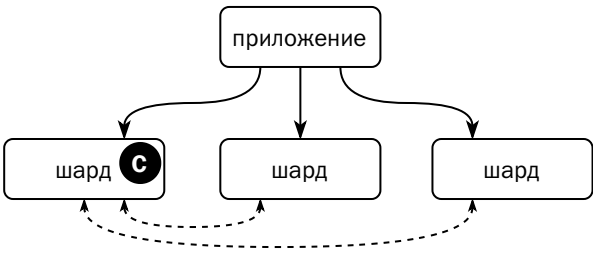
транзакций (global transaction manager), выдающий идентификаторы транзакций. Координатор получает идентификатор у менеджера транзакций, а затем выполняет транзакцию. Если транзакция требует изменения данных на нескольких узлах, то координатор использует протокол двухфазной фиксации. Дублирования данных между узлами и механизмов отказоустойчивости координатора на уровне платформы не предусмотрено — их должна обеспечить инфраструктура.

В SAP HANA также есть выделенный узел-координатор, однако в целях увеличения производительности координатором транзакции может выступать клиент с использованием механизма statement routing. Координатор разбирает запрос, сообщая клиенту, какие узлы будут участвовать в обработке этого запроса, после чего клиент может самостоятельно скоординировать выполнение.

«Настоящее горизонтальное масштабирование»

Ряд платформ, позиционирующих себя как «по-настоящему горизонтально масштабируемые», состоят из множества равноправных узлов. Узел не обязан существовать вечно — его можно развернуть, например, на физическом сервере с внутренними дисками: в отличие от виртуальной машины, перезапускаемой на соседнем сервере фермы виртуализации, физический сервер при отказе исчезает навсегда, и в кластер может быть введён другой узел. Восстановление целостности данных при сбоях целиком берёт на себя платформа.

В кластере может быть «главный узел», но набор его функций минимален, а роль главного в разное время могут играть разные узлы. Любой узел может координировать распределённые транзакции: координатором становится тот узел, на котором транзакция началась, — на рисунке узел-координатор текущей транзакции помечен символом C.



Примером таких систем могут служить MongoDB, Apache Ignite/GridGain, Hazelcast, а также аналитическая платформа Micro Focus Vertica.

Все перечисленные системы используют протокол двухфазной фиксации.

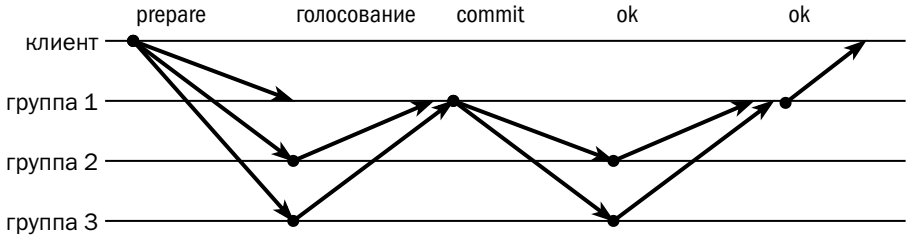
Детали реализации могут отличаться. Так, например, в MongoDB координатором является не узел, а шард (replica set), участвующий в транзакции. В случае отказа узла, хранящего основную копию шарда, транзакцию завершает одна из реплик. В Hazelcast узел-координатор реплицирует состояние транзакции на узлы-участники, чтобы в случае его отказа транзакцию мог завершить другой узел. В Apache Ignite в случае отказа одного из участников, включая координатора, инициируется обмен сообщениями между всеми участниками транзакции с целью её корректного завершения.

Также по-разному разные платформы подходят к оптимизации. Ignite может заменить двухфазную фиксацию однофазной, а Hazelcast требует, чтобы клиент при создании транзакции явно указал, будет ли она распределённой и потребуется ли ей двухфазная фиксация.

Google Spanner

СУБД Spanner хранит данные в ячейках хранения, «таблетках» (tablets). Надёжность записи обеспечивается за счёт объединения нескольких «таблеток» в разных центрах обработки данных в группы, консенсус в которых достигается собственной реализацией Multi-paxos, где инициатор записи вместе с обещанием получает мандат на 10 секунд.

В транзакции могут участвовать несколько групп, и для фиксации транзакции используется двухфазный протокол. Клиент накапливает изменения в своей памяти, а по команде commit отсылает их всем участникам транзакции. Одну из групп-участников клиент назначает координатором транзакции, о чём уведомляет всех участников.



Участники выполняют подготовительные действия, но о результатах сообщают не клиенту, а лидеру группы-координатора, который и завершает двухфазную фиксацию.

Google использует оригинальную методику создания временны́х меток транзакций.

Очевидно, что нельзя непосредственно сравнивать временны́е метки, созданные на разных серверах: если часы одного сервера отстают от часов другого, то может оказаться, что метка, созданная первым сервером, меньше метки, созданной вторым сервером, но событие на первом сервере произошло позже. В то же время, если одна временна́я метка на час меньше другой, мы с уверенностью можем сказать, что первое событие произошло раньше.

Минимальная разница между временны́ми метками событий, при которой мы можем уверенно сказать, что одно событие произошло раньше другого, называется доверительным интервалом (confidence interval). В кластере доверительный интервал равен максимальной разнице между часами разных узлов этого кластера.

Протокол NTP позволяет синхронизировать время на серверах в разных ЦОДах с точностью до десятков миллисекунд, но надо помнить, что точность часов в сервере составляет ±0,2 мс/с, и при ежедневной синхронизации разница показаний часов на серверах может достигать нескольких секунд. Google устанавливает в каждом ЦОДе серверы точного времени (time masters), оборудованные одновременно GPS-приёмниками и атомными часами, а также использует для синхронизации собственный протокол TrueTime. Синхронизация часов каждого сервера раз в полчаса позволяет добиться, чтобы разница между показаниями часов любых двух узлов георaспределённого кластера в любой момент времени не превышала 15 миллисекунд.

Когда клиент подтверждает транзакцию, сервер приписывает ей временну́ю метку завершения. Взяв текущее время сервера и добавив к нему доверительный интервал, сервер гарантирует, что полученная метка будет больше, чем метка любой транзакции, завершённой ранее. Начав реальную запись только по истечении доверительного интервала, сервер гарантирует, что ни одна транзакция, начавшаяся до подтверждения текущей транзакции, не увидит её изменений. Временна́я метка для чтения также генерируется как показания часов узла, обслуживающего клиентский сеанс, с добавлением доверительного интервала.

CockroachDB

Создатели СУБД CockroachDB поставили перед собой весьма амбициозную цель: построить горизонтально масштабируемую реляционную СУБД со строги-

ми ACID-гарантиями. В CockroachDB реализован оптимизированный протокол двухфазной фиксации, вторая фаза которого выполняется асинхронно.

Базовая структура CocroachDB — хранилище «ключ—значение», разделенное на шарды. Каждый шард представляет собой Raft-группу, участники которой расположены на разных узлах. Движок SQL преобразует запросы в последовательность операций над ключами, с которыми и работает механизм транзакций.

Ключевая идея транзакционного механизма — хранение в БД объекта-транзакции.

CockroachDB может хранить несколько версий одной и той же строки. У каждой версии могут быть временна́я метка и идентификатор транзакции, создавшей эту версию. Если метка и идентификатор заданы, то строка заблокирована.

При обработке транзакции координатор выполняет следующие действия:

- генерирует идентификатор транзакции с тем расчётом, чтобы он попал в тот же шард, где находится первая изменяемая строка;
- создаёт новую версию первой строки, устанавливает ей идентификатор транзакции и временну́ю метку, соответствующую началу транзакции;
- создаёт объект-транзакцию в состоянии «ожидает» (pending);
- создаёт новые версии остальных строк, также устанавливая им временны́е метки и идентификатор транзакции;
- записывает в объект-транзакцию ключи всех изменённых строк;
- если во время записи произошла какая-то ошибка, то меняет состояние транзакции на «отменена» (aborted), иначе — на «подтверждена» (committed);
- сообщает клиенту, что транзакция успешно завершена, — на этом синхронная часть протокола окончена;
- перебирает все строки, изменённые в транзакции, и либо удаляет их, либо обнуляет временну́ю метку и идентификатор транзакции (снимает блокировку);
- удаляет объект-транзакцию.

Если какая-то из операций обнаруживает версию объекта с непустым идентификатором транзакции, то она ищет объект-транзакцию. Дальнейшие действия зависят от результата:

- если транзакция в состоянии «подтверждена», то любая операция, включая чтение, обязана обнулить временну́ю метку и идентификатор и работать с текущей версией;
- если транзакция в состоянии «отменена», то любая операция обязана удалить версию строки и работать с предыдущей версией;

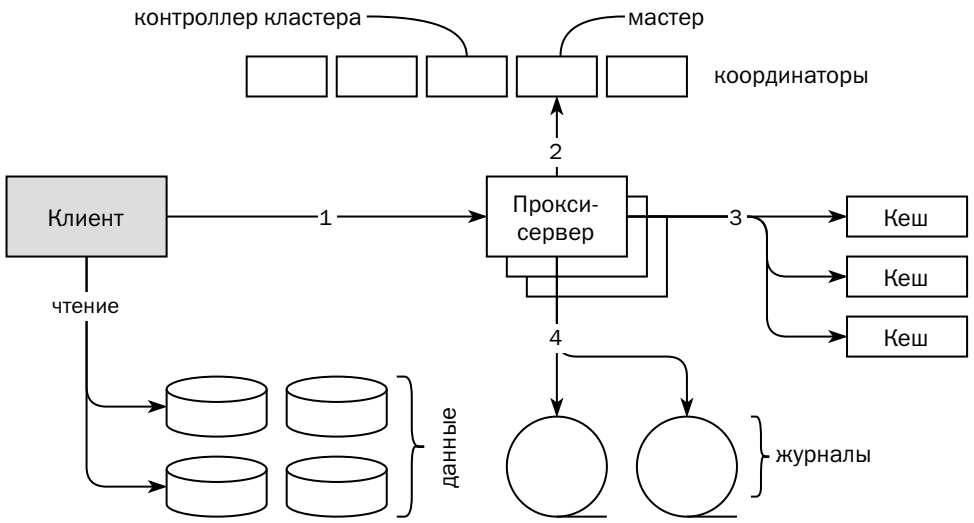
- если транзакция в состоянии «ожидает», это значит, что возник конфликт: читающая транзакция должна работать с предыдущей версией, а пишущая должна встать в очередь и ждать завершения активной транзакции;
- если объект-транзакция не найден, то операция анализирует временну́ю метку: если метка новее, чем метка текущей транзакции, то операция действует так же, как если бы обнаружила ожидающую транзакцию, а если нет, то так, будто транзакция отменена.

Полное описание алгоритма, приведённое в документации, содержит некоторые дополнительные оптимизации.

FoundationDB

Создатели FoundationDB ставили перед собой несколько более скромную задачу, чем авторы CockroachDB. Данная платформа представляет собой хранилище класса «ключ—значение» со строгими гарантиями сериализации (strict serializable), на базе которого каждый желающий может построить, например, реляционную СУБД. Таким образом, эта платформа является как бы основой (foundation) для других.

В FoundationDB реализован оригинальный алгоритм управления распределёнными транзакциями, напоминающий механизм детерминированных транзакций, но имеющий ряд существенных отличий.



Кластер FoundationDB состоит из множества узлов, играющих различные роли.

Координаторы образуют Paxos-группу, выбирающую узел-контроллер кластера (cluster controller) и узел-мастер (master). Контроллер следит за состоянием остальных узлов и отвечает за восстановление при потере какого-либо узла. Мастер служит источником временны́х меток для транзакций.

Вся информация о текущей транзакции хранится на клиенте. Когда приложение начинает транзакцию, даже читающую, клиент обращается ко всем кеширующим узлам, чтобы узнать максимальную временну́ю метку завершённых транзакций. Полученная метка используется, чтобы при чтении получить нужную версию данных.

Чтение выполняется непосредственно с серверов хранения данных (storage servers). Эти серверы хранят несколько версий объектов и при запросе от клиента отдают ту версию, которая соответствует временно́й метке читающей транзакции.

Если транзакция изменила данные, то для её завершения клиент обращается (1) к прокси-серверу (proxy). Информация, передаваемая прокси-серверу, содержит ключи всех прочитанных объектов и их версии.

Прокси-сервер первым делом обращается (2) к узлу-мастеру, чтобы получить временну́ю метку завершения транзакции. Мастер обрабатывает запросы пакетами и может выдать одну и ту же метку завершения нескольким транзакциям, поэтому масштабирование мастера не требуется.

Затем прокси обращается (3) к кеширующим узлам (resolvers), которые хранят последние изменения данных. Каждый узел отвечает за некоторый диапазон ключей. Кеширующий узел получает ключи изменяемых объектов и их новые версии. Если полученные версии новее, чем те, которые известны кеширующему узлу, то узел обновляет известные версии и разрешает завершение транзакции. Получив одобрение всех кеширующих узлов, прокси записывает (4) изменения в журналы, откуда их асинхронно считывают серверы хранения.

Если какие-то кеширующие узлы не одобрили транзакцию, прокси отвечает клиенту, что транзакция не удалась. При этом часть кеширующих узлов могла одобрить транзакцию и обновить известные версии. Если параллельная транзакция попытается обновить объекты, обновление которых было одобрено для отменённой транзакции, то эта транзакция тоже будет отменена, хотя на самом деле она вполне могла бы успешно выполниться.

Кеширующие узлы хранят ключи в течение пяти секунд, и продолжительность транзакции также ограничена пятью секундами. Если при запросе одобрения

у кеширующего узла нет запрошенного ключа, то он считает, что с момента начала транзакции объект с этим ключом не изменялся, и разрешает транзакцию.

Узлы хранения данных хранят в памяти все версии объектов за последние пять секунд, а более старые изменения записывают на диск.

Независимые узлы — участники саги

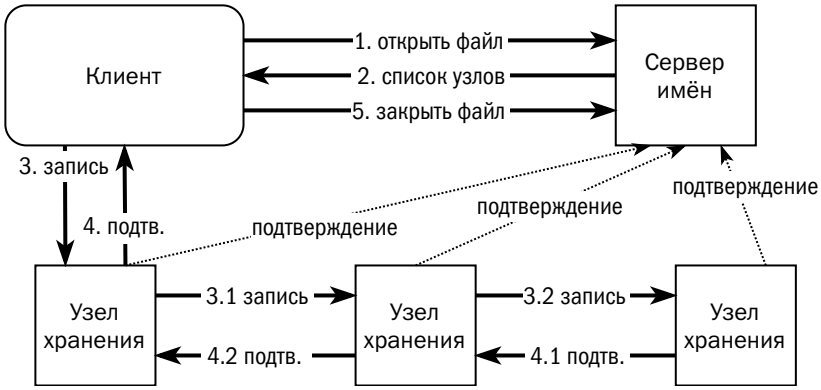
Использование саг характерно для распределённых приложений, компоненты которых работают с собственными экземплярами баз данных. Однако есть и СУБД, где распределённые транзакции организованы по принципу саги.

Так, например, в СУБД Tarantool отсутствуют блокировки — изоляция транзакций обеспечивается за счёт того, что каждый набор данных (шард) обрабатывается единственным потоком. Транзакция в Tarantool представляет собой процедуру на языке Lua. Экземпляры Tarantool взаимодействуют по модели акторов: получив сообщение, экземпляр может изменить собственные данные и послать сообщения другим экземплярам.

Если приложению, работающему с Tarantool, требуется согласованно изменить данные нескольких шардов, то оно должно составить распределённую транзакцию как сагу, обеспечить идемпотентность всех шагов и разработать компенсирующие операции для каждого шага (которые, разумеется, тоже должны быть идемпотентны).

Запись данных в Hadoop также можно рассматривать как сагу, оркестрируемую клиентом:

1. Клиент обращается к серверу имён (name node) с намерением открыть файл.
2. Сервер имён выдаёт мандат на запись и список узлов, на которые должны быть записаны данные.
3. Клиент открывает соединение с одним из узлов хранения (data node) и начинает записывать данные. Узел сохраняет данные на свои диски и параллельно начинает репликацию на следующий узел — таким образом выстраивается цепочка (pipeline).
4. Закончив запись, узел информирует об этом сервер имён и предыдущий узел в цепочке. Первый узел в цепочке информирует клиента.
5. Когда запись закончена, клиент сообщает серверу имён об успешно законченной транзакции, закрывая файл.



Поскольку данные в HDFS неизменны, блок доступен для чтения сразу же после того, как он записан.

Если клиент выходит из строя в процессе записи, то сервер имён сохраняет его мандат, чтобы приложение могло восстановиться и продолжить запись. Если же мандат истёк, а приложение не восстановилось, то HDFS считает, что запись файла завершена — мандат уничтожается, а файл закрывается. Отказ узлов с данными в процессе записи обрабатывается прозрачно для клиента.

По принципу саги устроены также многодокументные транзакции в Couchbase.

Литература

- Leslie Lamport. Paxos made simple.
  lamport.azurewebsites.net/pubs/paxos-simple.pdf
- Agreement in Distributed Systems / Columbia university.
  www.cs.columbia.edu/~du/ds/assets/lectures/lecture17.pdf
- Режимы кворума WSFC и участвующая в голосовании конфигурация (SQL Server).
  docs.microsoft.com/ru-ru/sql/sql-server/failover-clusters/windows/wsfc-quorum-modes-and-voting-configuration-sql-server?view=sql-server-ver15
- Diego Ongaro, John Ousterhout. In Search of an Understandable Consensus Algorithm.
  raft.github.io/raft.pdf
- Henrik Ingo. Four modifications for the Raft consensus algorithm.
  www.openlife.cc/sites/default/files/4-modifications-for-Raft-consensus.pdf

- ZAB in words.
  cwiki.apache.org/confluence/display/ZOOKEEPER/Zab1.0
- Jepsen analysis: MongoDB 3.4.0-rc3.
  jepsen.io/analyses/mongodb-3-4-0-rc3
- Henrik Ingo. MongoDB and Raft. PGDay Russia, 2017.
  henrikingo.github.io/presentations/PGDay Russia 2017 - MongoDB and Raft/index.html
- Jonathan Ellis. Lightweight transactions in Cassandra 2.0.
  www.datastax.com/blog/2013/07/lightweight-transactions-cassandra-20
- Jepsen: Cassandra.
  aphyr.com/posts/294-jepsen-cassandra
- Mohan, Lindsay, Obermarckt. Transaction Management in the R* Distributed Database Management System.
  pdos.csail.mit.edu/6.824/papers/rstar.pdf
- X/Open group. Technical Standard Distributed Transaction Processing: The XA Specification.
  pubs.opengroup.org/onlinepubs/009680699/toc.pdf
- Daniel J. Abadi et al. Calvin: Fast Distributed Transactionsfor Partitioned Database Systems.
  cs.yale.edu/homes/thomson/publications/calvin-sigmod12.pdf
- Matt Freels. Consistency without Clocks: The FaunaDB Distributed Transaction Protocol.
  fauna.com/blog/consistency-without-clocks-faunadb-transaction-protocol
- Aaron Ploetz. ACID Transactions Change the Game for Cassandra Developers.
  thenewstack.io/acid-transactions-change-the-game-for-cassandra-developers/
- Benedict Elliott Smith. CEP-15: General Purpose Transactions.
  cwiki.apache.org/confluence/display/CASSANDRA/CEP-15%3A+General+Purpose+Transactions
- Caitie McCaffrey. Distributed sagas: A Protocol for Coordinating Microservices.
  speakerdeck.com/caitiem20/distributed-sagas-a-protocol-for-coordinating-microservices
- Hector Garcia-Molina, Kenneth Salem. Sagas.
  www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf
- Компоненты параллельного хранилища данных — система аналитики платформы.
  docs.microsoft.com/ru-ru/sql/analytics-platform-system/parallel-data-warehouse-overview?view=aps-pdw-2016-au7

- S. Vithal. Netezza Failover — High Availability Architecture.
  dwgeek.com/netezza-failover-high-availability-architecture.html/
- SAP Statement Routing.
  help.sap.com/viewer/7e4aba181371442d9e4395e7ff71b777/2.0.03/en-US/25aaa3f8b2f34a2b9c158fff7c2575be.html
- SAP Distributed Transaction Handling.
  help.sap.com/viewer/7e4aba181371442d9e4395e7ff71b777/2.0.03/en-US/0c020e58e9464747aa45c919d73288d7.html
- Hazelcast: Creating a Transaction Interface.
  docs.hazelcast.org/docs/4.0.3/manual/html-single/index.html#creating-a-transaction-interface
- Postgres XL Global Transaction Management.
  www.postgres-xl.org/documentation/xc-overview-gtm.html
- Dmitriy Setrakyan. One-Phase-Commit — Fast Transactions For In-Memory Caches.
  gridgain.blogspot.com/2014/09/one-phase-commit-fast-transactions-for.html
- James C. Corbett et al. Spanner: Google's Globally-Distributed Database.
  static.googleusercontent.com/media/research.google.com/en/us/archive/spanner-osdi2012.pdf
- CockroachDB transaction layer.
  www.cockroachlabs.com/docs/v20.1/architecture/transaction-layer.html
- Mohammad Roohitavaf. Distributed Transactions in FoundationDB.
  www.mydistributed.systems/2020/08/foundationdb.html
- Денис Рысцов. Транзакции в MongoDB.
  habr.com/ru/post/153321/
- Tarantool transaction control.
  www.tarantool.io/ru/doc/2.5/book/box/atomic/
- Yongjun Zhang. Understanding HDFS Recovery Processes (Part 1).
  blog.cloudera.com/understanding-hdfs-recovery-processes-part-1/
- Apache Cassandra: Repairing nodes.
  docs.datastax.com/en/cassandra-oss/3.x/cassandra/operations/opsRepairNodesTOC.html
- Bryan Fink. Why Vector Clocks are Easy.
  riak.com/why-vector-clocks-are-easy/index.html
- Carlos Baquero. Version Vectors are not Vector Clocks.
  haslab.wordpress.com/2011/07/08/version-vectors-are-not-vector-clocks/
- Nuno Preguiça. Dotted Version Vectors: Logical Clocks for Optimistic Replication.
  arxiv.org/pdf/1011.5808.pdf

- Ricardo Tomé Gonçalves. Dotted Version Vector.
  awesomeopensource.com/project/ricardobcl/Dotted-Version-Vectors
- Marc Shapiro, Nuno Preguiça, Carlos Baquero, Marek Zawirski.
  A comprehensive study of Convergent and Commutative Replicated Data Types.
  hal.inria.fr/file/index/docid/555588/filename/techreport.pdf
- Denis Rosa. How we implemented Distributed Multi-document ACID
  Transactions in Couchbase.
  blog.couchbase.com/distributed-multi-document-acid-transactions/

Часть V

Восстановление при сбоях

Глава 9
Репликация

> Повторяться, но каждый раз по-новому — разве не это есть искусство?
>
> *Станислав Ежи Лец, из книги «Непричёсанные мысли»*

Словарь определяет репликацию как процесс поддержания двух (или более) наборов данных в согласованном состоянии. Что такое «согласованное состояние наборов данных» — отдельный большой вопрос, поэтому переформулируем определение проще: **репликация — процесс изменения некоторых наборов данных, называемых репликами, в ответ на изменения эталонного набора данных**. Совсем не обязательно наборы-реплики при этом будут идентичны набору-источнику.

Репликация — одна из важнейших функций СУБД: почти у каждой сколько-нибудь важной базы данных есть реплика, а то и не одна.

Среди задач, решаемых репликацией, можно назвать как минимум следующие:

- поддержка резервной базы данных на случай потери основной;
- снижение нагрузки на базу за счёт переноса части запросов на реплики;
- перенос данных в архивные или аналитические системы.

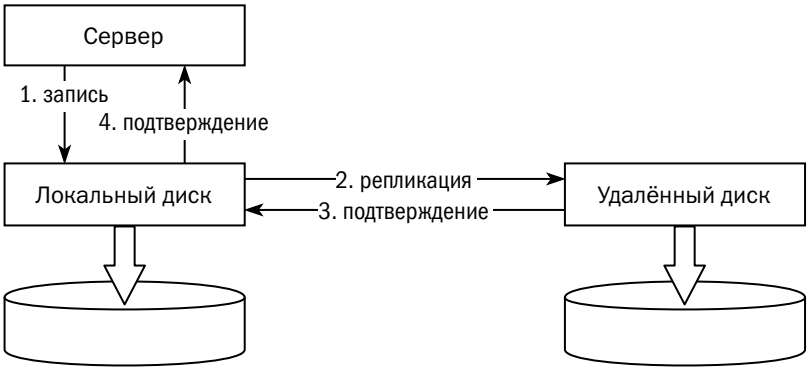
Можно выделить три подхода к репликации:

1) блочная репликация на уровне системы хранения данных;
2) физическая репликация на уровне СУБД;
3) логическая репликация на уровне СУБД.

9.1. Блочная репликация

При блочной репликации каждая операция записи выполняется не только на основном диске, но и на резервном. Таким образом то́му на одном массиве соответствует зеркальный том на другом массиве, с точностью до байта повторяющий основной том.

К достоинствам такой репликации можно отнести простоту настройки и надёжность.



Записывать данные на удалённый том может дисковый массив, дополненный опциями репликации. Название опции зависит от производителя массива:

Производитель	Торговая марка
Dell/EMC[2]	SRDF (Symmetrix Remote Data Facility)
IBM	Metro Mirror — синхронная репликация; Global Mirror — асинхронная репликация
Hitachi	TrueCopy
Hewlett-Packard Enterprise	Continuous Access
Huawei	HyperReplication

Если у дискового массива нет функции репликации, между экземпляром БД и диском может быть установлен посредник, способный параллельно записывать идентичные данные на несколько томов. Роль посредника может играть как отдельное устройство (EMC[2] VPLEX), так и программный компонент (HPE PeerPersistence, Windows Server Storage Replica, DRBD). В отличие от дискового массива, который может работать только с таким же массивом или, как минимум, с массивом того же производителя, посредник может работать с совершенно разными дисковыми устройствами.

Главное назначение блочной репликации — обеспечение отказоустойчивости. Если потерян основной том, содержащий базу данных, то можно перезапустить экземпляр с использованием зеркального тома.

Блочная репликация хороша своей универсальностью, но за универсальность приходится платить.

Пока работает блочная репликация, никакой сервер не может работать с зеркальным томом, поскольку записью на этот том фактически управляет опера-

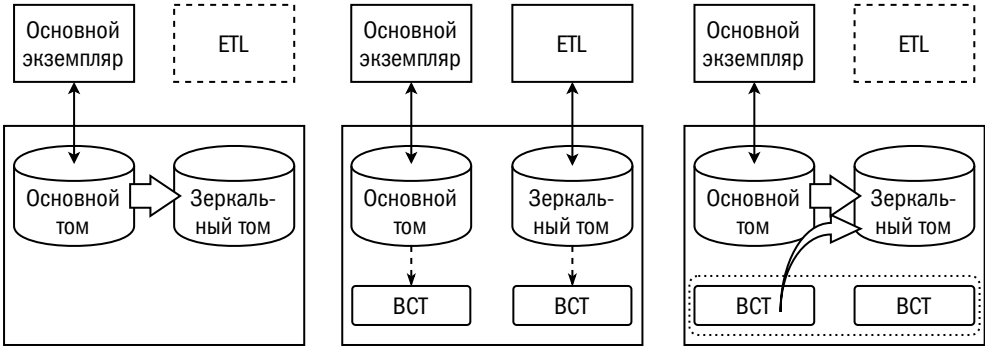
ционная система сервера, работающего с основным томом. Из этого следует и дополнительное ограничение — блочная репликация не может использоваться для распределения нагрузки.

Экземпляр БД на резервном сервере может быть запущен только после монтирования диска к резервному серверу. Запуск экземпляра может занимать продолжительное время: так, например, в некоторых операционных системах память под буферный кеш размечается при выделении, и время разметки пропорционально объёму выделяемой памяти. Мало того, после запуска экземпляра кеш будет пуст.

После запуска экземпляра на резервном сервере он обнаружит, что данные на диске не согласованы и нужно потратить значительное время на восстановление данных.

При аварии диска — например, при отказе массива, содержащего основной том, или при потере всего ЦОДа, где находится основной сервер, — следует остановить репликацию, размонтировать основной том, смонтировать зеркальный том на резервном сервере и запустить экземпляр БД. Как только появится возможность, следует перезапустить репликацию в обратном направлении. Использование кластерного ПО упрощает настройку, но не уменьшает время переключения.

Для обновления хранилища данных иногда используется похожая схема, когда зеркальный том находится в том же массиве, что и основной. У Dell/EMC[2] и HPE эта схема называется BCV, при этом одна компания расшифровывает аббревиатуру как Business Continuance Volume, а другая — как Business Copy Volume. У IBM на этот случай нет специальной торговой марки, схема с зеркалированием тома так и называется — «mirrored volume».



1. Зеркалирование 2. Зеркало разорвано 3. Восстановление зеркала

В массиве создаются два тома, и операции записи синхронно выполняются на обоих (1). В определённое время зеркало разрывается (2), то есть тома становятся независимыми. Зеркальный том монтируется к серверу[1], выделенному для обновления хранилища, и на этом сервере поднимается экземпляр базы данных. Экземпляр будет подниматься так же долго, как и при восстановлении с помощью блочной репликации, но это время может быть существенно уменьшено за счёт разрыва зеркала в период минимальной нагрузки. Дело в том, что разрыв зеркала по своим последствиям эквивалентен аварийному завершению экземпляра, а время восстановления при аварийном завершении существенно зависит от количества активных транзакций в момент аварии. База данных, предназначенная для выгрузки, доступна как для чтения, так и для записи. Идентификаторы всех блоков, изменённых после разрыва зеркала как на основном, так и на зеркальном томе, сохраняются в специальной области Block Change Tracking, BCT.

После окончания выгрузки зеркальный том размонтируется (3), и зеркало восстанавливается путём копирования с основного тома на зеркальный блоков, помеченных изменёнными на одном из томов. Через некоторое время зеркальный том вновь догоняет основной и становится его копией.

9.2. Физическая репликация

Журналы базы данных содержат информацию обо всех изменениях в этой базе. Идея физической репликации состоит в том, что изменения из журналов повторно применяются к другой базе (реплике), и таким образом данные в реплике в точности, до байта, повторяют данные в основной базе.

Возможность использования журналов базы данных для обновления реплики появилась в релизе Oracle 7.3, который вышел в 1996 году, а уже в релизе Oracle 8i доставка журналов с основной базы в реплику была автоматизирована и получила название Data Guard. Технология оказалась настолько востребованной, что сегодня механизм физической репликации есть практически во всех современных СУБД.

[1] Сервер обозначен аббревиатурой ETL — Extract, Transform, Load.

СУБД	Компонент, отвечающий за репликацию
Oracle	Active Data Guard
IBM Db2	HADR
Microsoft SQL Server	Log shipping / Always On
PostgreSQL	Log shipping / Streaming replication
MySQL	Alibaba physical InnoDB replication

Опыт показывает, что если использовать сервер только для поддержания реплики в актуальном состоянии, то ему достаточно примерно 10 % процессорной мощности сервера, на котором работает основная база.

Журналы СУБД не предназначены для использования вне этой платформы, документации по ним нет в открытом доступе, а формат может меняться без предупреждения. Поэтому физическая репликация возможна только между экземплярами одной и той же версии одной той же платформы. Отсюда же возникают и возможные ограничения на операционную систему и архитектуру процессора, которые тоже могут влиять на формат журнала.

Естественно, никаких ограничений на модели СХД физическая репликация не накладывает. Более того, файлы в базе-реплике могут располагаться не так, как на базе-источнике, — надо лишь описать соответствие между томами и каталогами, в которых расположены эти файлы.

Oracle Data Guard позволяет удалить часть файлов из базы-реплики — в этом случае изменения в журналах, относящиеся к этим файлам, игнорируются.

Физическая репликация базы данных имеет множество преимуществ перед репликацией средствами СХД:

- объём передаваемых данных меньше за счёт того, что передаются только журналы, но не файлы с данными; эксперименты показывают уменьшение трафика в 5–7 раз;
- переключение на резервную базу происходит значительно быстрее: экземпляр-реплика уже запущен, поэтому при переключении ему нужно лишь откатить активные транзакции; более того, к моменту сбоя кеш реплики уже частично прогрет — в нём есть данные, которые были изменены, но нет данных, которые на основной базе были только прочитаны, но не обновлены;
- на реплике можно выполнять читающие запросы, сняв тем самым часть нагрузки с основной базы; в частности, реплику можно использовать для создания резервных копий.

Возможность читать данные с реплики появилась в 2007 году в релизе Oracle 11g — именно на это указывает эпитет «active», добавленный к названию технологии Data Guard. В других СУБД возможность чтения с реплики также есть, но в торговых марках это никак не отражено.

Запись данных в реплику невозможна, поскольку изменения в неё приходят побайтно, и реплика не может обеспечить конкурентное исполнение своих запросов. Oracle Active Data Guard в последних релизах разрешает запись в реплику, но это не более чем «сахар»: на самом деле изменения выполняются в основной базе, а клиент ждёт, пока они докатятся до реплики.

В случае повреждения файла в основной базе можно просто скопировать соответствующий файл с реплики (прежде чем делать такое со своей базой, внимательно изучите руководство администратора!).

При сравнении файла на реплике с файлом в исходной базе средствами операционной системы может оказаться, что файлы не идентичны. Дело в том, что при расширении файла новые блоки в целях ускорения могут ничем не заполняться. Кроме того, база может использовать не всё пространство блока (например, в блоке может оставаться свободное место). Содержимое таких неиспользованных областей может быть случайным, но задействованное пространство совпадает побайтно.

Физическая репликация может быть как синхронной, так и асинхронной. При асинхронной репликации всегда есть некоторый набор транзакций, которые завершены в основной базе, но ещё не дошли до резервной, и в случае перехода на резервную базу при сбое основной эти транзакции будут потеряны. При синхронной репликации завершение операции `COMMIT` означает, что все журнальные записи, относящиеся к данной транзакции, переданы на реплику. Важно понимать, что получение репликой журнала не означает применения изменений к данным. При потере основной базы транзакции не будут потеряны, но если приложение пишет данные в основную базу и считывает их из реплики, то у него есть шанс получить старую версию этих данных.

В PostgreSQL есть возможность сконфигурировать репликацию так, чтобы операция `COMMIT` завершалась только после применения изменений к данным реплики (параметр `synchronous_commit = remote_apply`), а в Oracle можно сконфигурировать всю реплику или отдельные сеансы, чтобы запросы не выполнялись, если реплика отстаёт от основной базы (параметр `STANDBY_MAX_DATA_DELAY = 0`). Однако всё же лучше проектировать приложение так, чтобы запись в основную базу и чтение из реплик выполнялись в разных модулях.

При поиске ответа на вопрос, какой режим выбрать, синхронный или асинхронный, нам на помощь приходят маркетологи Oracle. Data Guard предусматривает три режима, каждый из которых максимизирует один из параметров — производительность, сохранность данных, доступность базы — за счёт остальных:

- maximum performance: репликация всегда асинхронная;
- maximum protection: репликация синхронная; если реплика не отвечает, транзакция на основной базе не завершается;
- maximum availability: репликация синхронная; если реплика не отвечает, то репликация переключается в асинхронный режим и, как только связь восстанавливается, реплика догоняет основную базу, и репликация снова становится синхронной.

Несмотря на бесспорные преимущества репликации средствами БД, администраторы во многих компаниях, особенно со старыми традициями надёжности, до сих пор очень неохотно отказываются от блочной репликации. Тому есть две причины.

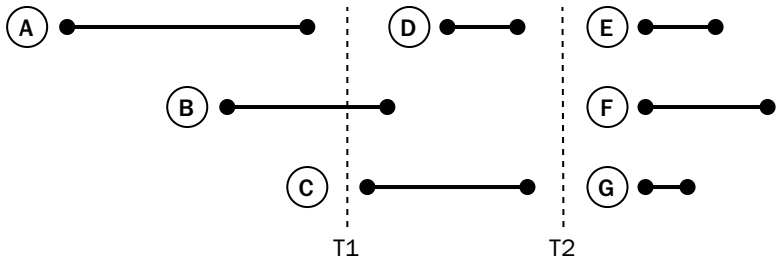
Во-первых, в случае репликации средствами дискового массива трафик идёт не по сети передачи данных (LAN), а по сети хранения данных (SAN). Зачастую в инфраструктурах, построенных давно, SAN гораздо надёжнее и производительнее, чем сеть передачи данных.

Во-вторых, синхронная репликация средствами СУБД стала надёжной относительно недавно. В Oracle прорыв произошёл в релизе 11g, который вышел в 2007 году, а в других СУБД синхронная репликация появилась ещё позже. Конечно, 10–15 лет по меркам сферы информационных технологий — гигантский срок, но когда речь идёт о сохранности данных, многие администраторы руководствуются принципом «как бы чего не вышло»...

9.3. Логическая репликация

Все изменения в базе данных происходят в результате вызовов её API — например, в результате выполнения SQL-запросов. Очень заманчивой кажется идея выполнять одну и ту же последовательность команд (транзакций) силами самой платформы в двух разных базах. Для корректной репликации необходимо придерживаться двух правил:

1. Нельзя начинать транзакцию, пока не завершены все транзакции, которые должны закончиться раньше. Так, на рисунке ниже нельзя запускать транзакцию D, пока не завершены транзакции A и B.
2. Нельзя завершать транзакцию, пока не начаты все транзакции, которые должны начаться до завершения текущей транзакции. Так, на рисунке ниже даже если транзакция B выполнилась мгновенно, завершить её можно только после того, как начнётся транзакция C.



Репликация команд (statement-based replication) реализована, например, в MySQL. К сожалению, на практике эта простая схема не приводит к появлению идентичных наборов данных. Существуют два источника расхождений.

Во-первых, не все API детерминированы. Например, если в SQL-запросе встречается функция `now()` или `sysdate()`, возвращающая текущее время, то на разных серверах она вернёт разный результат — из-за того, что запросы выполняются не одновременно. Кроме того, к различиям могут привести разные состояния триггеров и хранимых функций, разные национальные настройки, влияющие на порядок сортировки, и многое другое.

Во-вторых, репликацию, основанную на исполнении идентичных команд, невозможно корректно приостановить и перезапустить.

Если репликация остановлена в момент T1, транзакцию B необходимо прервать и откатить. При перезапуске репликации исполнение транзакции B может привести реплику к состоянию, отличному от состояния базы-источника: на источнике транзакция B началась до того, как закончилась транзакция A, а значит, она не видела изменений, сделанных транзакцией A.

Репликация запросов может быть остановлена и перезапущена только в момент T2, когда в базе нет ни одной активной транзакции. Разумеется, на сколько-нибудь нагруженной промышленной базе таких моментов не бывает.

Обычно для логической репликации используют детерминированные запросы, то есть запросы, обладающие двумя свойствами:

- запрос обновляет (или вставляет, или удаляет) единственную строку, идентифицируя её по первичному (или уникальному) ключу;
- все параметры запроса явно заданы в самом запросе.

В отличие от репликации команд (statement-based replication) такой подход называется репликацией строк (row-based replication).

Предположим, что у нас есть таблица сотрудников со следующими данными:

ID	Name	Dept	Salary
3817	Иванов Иван Иванович	36	1800
2274	Петров Пётр Петрович	36	1600
4415	Кузнецов Семён Андреевич	41	2100

Над этой таблицей была выполнена следующая операция:

```
update employee set salary = salary*1.2 where dept=36;
```

Чтобы корректно реплицировать данные, процесс репликации выполнит такие команды:

```
update employee set salary = 2160 where id=3817;
update employee set salary = 1920 where id=2274;
```

Команды, выполненные в базе-реплике, приводят к тому же результату, что и команды в исходной базе, но при этом они не эквивалентны.

База-реплика открыта и доступна не только для чтения, но и для записи. Это позволяет использовать её для выполнения части запросов, в том числе для построения отчётов, требующих создания дополнительных таблиц или индексов.

Важно понимать, что логическая реплика будет эквивалентна исходной базе только в том случае, если в неё не вносится никаких дополнительных изменений. Например, если в примере выше в реплике добавить в 36-й отдел Сидорова, то он повышения не получит, а если Иванова перевести из 36-го отдела, то он всё равно получит повышение.

Логическая репликация предоставляет ряд возможностей, отсутствующих в других видах репликации:

- настройка набора реплицируемых данных на уровне таблиц и колонок (при физической репликации — на уровне файлов и табличных пространств, при блочной репликации — на уровне томов);
- построение сложных топологий репликации — например, консолидация нескольких баз в одной или двунаправленная репликация;

- уменьшение объёма передаваемых данных;
- репликация между разными версиями СУБД или даже между платформами разных производителей;
- обработка данных при репликации, в том числе изменение структуры, обогащение, сохранение истории.

Есть и недостатки, которые не позволяют логической репликации вытеснить физическую:

- все реплицируемые данные обязаны иметь первичные ключи;
- логическая репликация поддерживает не все типы данных — например, возможны проблемы с большими объектами (BLOB);
- логическая репликация на практике не бывает полностью синхронной: время от получения изменений до их применения слишком велико, чтобы основная база могла ждать;
- логическая репликация создаёт большую нагрузку на реплику, поскольку в реплике выполняется много команд изменения данных — иногда на порядок больше, чем на источнике;
- при переключении приложение должно иметь возможность убедиться, что все изменения с основной базы применены на реплике; сама СУБД не может этого определить, так как для неё режимы реплики и основной базы эквивалентны.

Два последних недостатка ограничивают использование логической реплики как средства отказоустойчивости. Если запросы в основной базе изменяют сразу много строк, реплика может существенно отставать. А возможность смены ролей требует недюжинных усилий как со стороны разработчиков, так и со стороны администраторов.

Есть несколько способов реализации логической репликации, и у каждого из этих способов есть сильные и слабые стороны:

- репликация триггерами;
- использование журналов СУБД;
- использование программного обеспечения класса change data capture;
- прикладная репликация.

Репликация триггерами

Триггер — хранимая процедура, которая исполняется автоматически при какой-либо модификации данных. Триггеру, который вызывается при измене-

нии каждой строки, доступен её ключ, а также старые и новые значения полей. При необходимости триггер может сохранять новые значения строк в буферную таблицу, откуда специальный процесс на стороне реплики будет их вычитывать. Объём необходимого кода велик, поэтому существует специальное программное обеспечение, генерирующее такие триггеры, например «Репликация слиянием» (merge replication) — компонент Microsoft SQL Server или отдельные продукты для репликации, такие как Slony-I, Londiste или SymmetricsDS.

Преимущества репликации триггерами:

- независимость от версий основной базы и реплики;
- широкие возможности преобразования данных.

Недостатки:

- нагрузка на основную базу;
- большая задержка при репликации.

Репликация с помощью журналов СУБД

Сами СУБД также могут предоставлять возможности логической репликации. Источником данных, как и для физической репликации, являются журналы. К информации о побайтовом изменении добавляется информация об изменённых полях (supplemental logging в Oracle, `wal_level = logical` в PostgreSQL), а также значение уникального ключа, даже если он не меняется. В результате объём журналов БД увеличивается — по разным оценкам от 10 до 15 %.

Терминология и инструменты, используемые для настройки репликации, зависят от конкретной платформы. Например, Oracle использует термин «логическая реплика» (logical standby), а PostgreSQL или Microsoft SQL Server встроенными средствами платформы позволяет развернуть сложную систему взаимных подписок (subscription) и публикаций (publication).

К недостаткам данного подхода можно отнести увеличение объёма журналов.

Репликация с помощью CDC

Существует целый класс программного обеспечения, предназначенного для организации логической репликации — CDC, change data capture. Вот список наиболее известных платформ этого класса:

- Oracle GoldenGate (компания GoldenGate приобретена в 2009 году);
- Vision Solutions DoubleTake/MIMIX (ранее — Vision Replicate1);

- IBM InfoSphere Data Replication (ранее — InfoSphere CDC; ещё ранее — DataMirror Transformation Server, компания DataMirror приобретена в 2007 году);
- Qlik Data Integration Platform (ранее — Attunity);
- Informatica PowerExchange CDC;
- Debezium;
- StreamSets Data Collector.

В задачу платформы входит чтение журналов базы данных, преобразование прочитанных данных, передача информации на реплику и применение. Как и в случае репликации средствами самой СУБД, журнал должен содержать информацию об изменённых полях. Использование дополнительного приложения позволяет «на лету» выполнять сложные преобразования реплицируемых данных и строить достаточно сложные топологии репликации.

Сильные стороны:

- возможность репликации между разными СУБД, в том числе загрузка данных в отчётные системы с минимальным отставанием от транзакционной базы данных;
- широчайшие возможности обработки и преобразования данных;
- минимальный трафик между узлами — CDC-платформа может отсекать ненужные данные и сжимать трафик;
- встроенные возможности мониторинга состояния репликации.

Недостатков не так много:

- увеличение объёма журналов, как при логической репликации средствами СУБД;
- новое ПО — сложное в настройке и, возможно, не дешёвое.

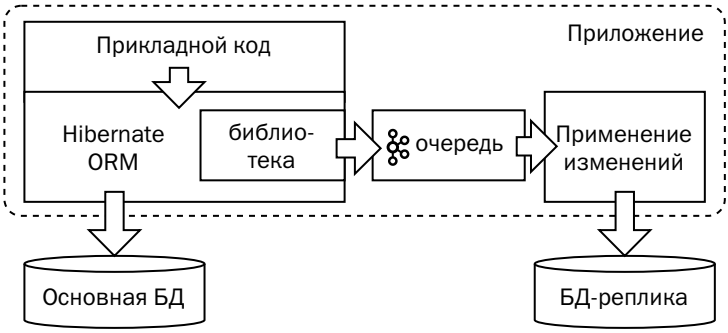
Именно CDC-платформы традиционно используются для обновления корпоративных хранилищ данных в режиме, близком к реальному времени.

Прикладная репликация

Наконец, ещё один способ репликации — формирование векторов изменений непосредственно на стороне клиента. Клиент должен формировать детерминированные запросы, затрагивающие единственную строку. Добиться этого можно, используя специальную библиотеку работы с базой данных, например Borland Database Engine (BDE) или Hibernate ORM.

Когда приложение завершает транзакцию, подключаемый модуль Hibernate ORM записывает вектор изменений в очередь и выполняет транзакцию в базе

данных. Специальный процесс-репликатор вычитывает векторы из очереди и выполняет транзакции в базе-реплике.



Этот механизм хорош для обновления отчётных систем. Может он использоваться и для обеспечения отказоустойчивости, но в этом случае в приложении должен быть реализован контроль состояния репликации.

Преимущества данного подхода:

- возможность репликации между разными СУБД, в том числе загрузка данных в отчётные системы;
- возможность обработки и преобразования данных, мониторинга состояния и т. д.;
- минимальный трафик между узлами — платформа может отсекать ненужные данные и сжимать трафик;
- полная независимость от базы данных — как от формата, так и от внутренних механизмов.

Достоинства этого способа бесспорны, однако есть два очень серьёзных недостатка:

- ограничения на архитектуру приложения;
- огромный объём собственного кода, обеспечивающего репликацию.

9.4. Так что же лучше?

Однозначного ответа на этот вопрос, как и на многие другие, не существует. Но надеюсь, что таблица ниже поможет сделать правильный выбор для каждой конкретной задачи:

	Блочная репликация СХД	Блочная репликация агентом	Физическая репликация	Репликация триггерами	Логическая репликация СУБД	Change data capture	Прикладная репликация
Воспроизведение источника	Побайтно	Побайтно	Побайтно	Логически	Логически	Логически	Логически
Выборочная репликация	Тома	Тома	Файлы	Таблицы и строки	Таблицы и строки	Таблицы и строки	Таблицы и строки
Объём трафика[1]	X	X	$X/7 - X/5$	$\leq X/10$	$\leq X/10$	$\leq X/10$	$\leq X/10$
Время переключения	5 мин — часы	5 мин — часы	1–10 мин	1–2 мин	1–2 мин	1–2 мин	1–2 мин
Гарантия эквивалентности реплики	+	+	+++	нет	±	±	нет
Доступность реплики	нет	нет	RO	RW	RW	RW	RW
Топология репликации	тчк-тчк тиражир.	тчк-тчк тиражир.	тчк-тчк тиражир. каскад	тчк-тчк тиражир. каскад слияние двунаправ.	тчк-тчк тиражир. каскад слияние двунаправ.	тчк-тчк тиражир. каскад слияние двунаправ.	тчк-тчк тиражир. каскад слияние двунаправ.
Нагрузка на источник	нет	нет	–	–––	––	––	нет[2]
Простота настройки	+++	++	++	–	+	+	–––
Стоимость доп. ПО	––	––	–	–	нет	–––	нет
Гетерогенные среды	нет	±[3]	+[4]	++	+	+++	+++

[1] Символом X обозначен объём изменённых страниц в базах данных со страничной организацией.

[2] Нагрузка есть, но вся логика обрабатывается серверами бизнес-логики, а не сервером БД.

[3] Разные конфигурации СХД.

[4] Экземпляры платформы на единой аппаратной платформе с разной конфигурацией серверного оборудования и СХД.

- Блочная репликация имеет смысл, когда других способов репликации нет — например, для файлов; для баз данных её лучше не использовать.
- Физическая репликация хороша, когда требуется обеспечение отказоустойчивости инфраструктуры или перенос части читающих приложений на реплики.
- Логическая репликация подходит для обеспечения отказоустойчивости только в том случае, если приложение знает об этой репликации и умеет в случае аварии ждать синхронизации реплик.
- Логическая репликация идеальна для непрерывного обновления отчётных баз данных.
- Репликация триггерами имеет смысл, если база сильно нагружена, а реплицировать нужно крайне ограниченное количество информации.
- Платформы CDC хороши, если у вас большое количество реплицируемых баз или если есть необходимость сложных преобразований данных.
- Разработка прикладной репликации оправдана только в случае разработки собственной облачной платформы или фреймворка.

Литература

- Сергей Петруня. Эволюция репликации в MySQL и MariaDB.
  www.osp.ru/netcat_files/userfiles/Hadoop_TBD_2_2016/Petrunya_tbd_2.pdf
- Windows Server: общие сведения о реплике хранилища.
  docs.microsoft.com/ru-ru/windows-server/storage/storage-replica/storage-replica-overview
- Jan Wieck. Slony-I. A replication system for PostgreSQL Concept.
  www.slony.info/images/Slony-I-concept.pdf
- Oracle Data Guard Concepts and Administration.
  docs.oracle.com/en/database/oracle/oracle-database/19/sbydb/data-guard-concepts-and-administration.pdf
- Administering Oracle GoldenGate.
  docs.oracle.com/en/middleware/goldengate/core/19.1/admin/configuring-online-change-synchronization.html
- Oracle Data Guard Evolution.
  subscription.packtpub.com/book/big_data_and_business_intelligence/9781849687904/1/ch01lvl1sec11/oracle-data-guard-evolution

Глава 10
Резервное копирование

> — О, никакое убежище не выдержит попадания метеорита. Но ведь у вас, как и у каждого, есть резерв, так что можете не беспокоиться.
>
> *Станислав Лем, «Звёздные дневники Ийона Тихого»*

Резервным копированием называется сохранение копии данных вне основного места их хранения.

Главное назначение резервного копирования — восстановление данных после их потери. В связи с этим нередко приходится слышать, что наличие реплики позволяет всегда восстановить данные после аварии и резервное копирование не нужно. На самом деле резервное копирование позволяет решить как минимум три задачи, которые не могут быть решены при помощи репликации.

Во-первых, резервная копия позволяет восстановить базу после логической ошибки. Например, бухгалтер удалил группу проводок, или администратор базы данных уничтожил табличное пространство. Обе операции абсолютно корректны с точки зрения СУБД, и процесс репликации воспроизведёт их в базе-реплике.

Во-вторых, современные СУБД — весьма надёжные программные комплексы, однако изредка всё же происходит повреждение внутренних структур базы, после которого доступ к данным пропадает. Физическая репликация, используемая для защиты от инфраструктурных сбоев, может перенести повреждения внутренней структуры и на реплику. Что особенно обидно, такое нарушение происходит обычно при высокой нагрузке или при установке какого-нибудь обновления. Но как высокая нагрузка, так и регулярные обновления говорят о том, что база отнюдь не тестовая, и её содержимое весьма ценно.

Наконец, третья задача, решение которой требует наличия резервной копии, — это клонирование базы, например для целей тестирования или для целей инициализации реплики.

Резервное копирование баз данных так или иначе базируется на одном из двух принципов, которые далее мы рассмотрим подробнее:

1) выборка данных с последующим сохранением в произвольном формате;
2) сохранение файлов БД, включая журналы.

10.1. Выгрузка данных

В наборе утилит, прилагающихся к любой СУБД, обязательно есть инструменты для выгрузки и загрузки данных. Как правило, данные сохраняются либо в текстовом виде, либо в двоичном формате, специфичном для конкретной платформы. В таблице ниже приведён список таких инструментов:

	Двоичный формат	Текстовый формат
Oracle	DataPump Export и DataPump Import; Export и Import	SQL*Plus и SQL*Loader
PostgreSQL	pg_dump/pg_dumpall и pg_restore	pg_dump/pg_dumpall и psql
Microsoft SQL Server	bcp	bcp
Db2	unload и load	unload и load
MySQL		mysqldump/mysqlpump и mysql/mysqlimport
MongoDB	mongodump и mongorestore	mongoexport и mongoimport
Cassandra	nodetool snapshot и sstableloader	cqlsh

Текстовый формат хорош тем, что его можно редактировать или даже создавать внешними программами. Двоичный, в свою, очередь позволяет быстрее выгружать и загружать данные за счёт распараллеливания загрузки и экономии ресурсов процессора на преобразовании форматов.

Несмотря на простоту и очевидность идеи выгрузки данных, для резервирования нагруженных промышленных баз такой метод применяют редко. Вот причины, по которым выгрузка не подходит для полноценного резервного копирования:

- процесс выгрузки создаёт значительную нагрузку на систему-источник;
- сделать согласованную выгрузку всей базы данных при высокой нагрузке практически невозможно, поскольку СУБД вынуждена хранить снимок своего состояния на момент начала выгрузки. Чем больше транзакций совершено с момента начала выгрузки, тем больше объём снимка (неактуальных копий данных в PostgreSQL, пространства undo в Oracle, tempdb в Microsoft SQL Server и т. п.);

- даже из согласованной выгрузки можно восстановить только состояние базы на момент выгрузки, но ни на какой другой;
- выгрузка сохраняет логическую структуру данных, но не сохраняет их физическую структуру — индексы, параметры физического хранения таблиц и др.; восстановление индексов после загрузки данных может занимать значительное время.

Тем не менее у выгрузки есть и достоинства:

- высокая избирательность: можно выгрузить отдельные таблицы, отдельные поля и даже отдельные строки;
- выгруженные данные можно загрузить в базу данных другой версии, а если выгрузка сделана в текстовом формате, то и в другую базу данных.

Выгрузка используется в основном для таких задач, как резервирование небольших таблиц (например, справочников) или распространение наборов данных с очередным релизом приложения. Самым же распространённым методом резервного копирования баз данных является сохранение файлов базы данных.

10.2. Холодное сохранение файлов БД

Очевидная идея — остановить экземпляр и скопировать все файлы базы данных. Такая резервная копия называется холодной (cold backup). Способ крайне надёжный и простой, но у него есть два очевидных недостатка:

- из холодной резервной копии можно восстановить только то состояние базы данных, которое было в момент останова экземпляра; транзакции, сделанные после перезапуска экземпляра, в холодную резервную копию не попадут;
- далеко не у каждой базы данных есть технологическое окно, когда экземпляр можно остановить.

Если же холодное резервное копирование вас устраивает, нужно помнить о следующем:

- Холодная копия иногда должна включать в себя и журналы. Методы определения журналов, которые должны попасть в холодную копию, индивидуальны для каждой платформы. Например, в Oracle необходимо скопировать оперативные журналы (online redo logs), то есть фиксированное количество журнальных файлов в специальном каталоге, причём даже тогда, когда экземпляр остановлен корректно. В PostgreSQL нужно сохранить все журналы

начиная с того, который содержит последнюю контрольную точку, информация о которой, в свою очередь, содержится в управляющем файле[1].

- Каталог базы данных может содержать достаточно большие файлы временных табличных пространств, которые не обязательно включать в резервную копию. Кстати, это замечание верно и для горячего резервного копирования.

10.3. Горячее сохранение файлов БД

Большинство резервных копий современных баз данных выполняются путём копирования файлов базы данных без остановки экземпляра. Такая резервная копия называется горячей (hot backup).

Здесь видны несколько проблем:

- В момент начала копирования содержимое базы данных может не совпадать с содержимым файлов, т. к. часть информации находится в кеше и ещё не записана на диск.
- Во время копирования содержимое базы может меняться. Если используются изменяемые структуры данных, то меняется содержимое файлов, а при использовании неизменяемых структур меняется набор файлов: новые файлы появляются, а старые удаляются.
- Поскольку запись данных в базу и чтение файлов БД утилитой резервного копирования могут быть не синхронизированы, утилита может прочитать некорректное состояние страницы, где часть содержимого будет от старой версии, а часть — от новой.

Чтобы резервная копия получилась согласованной, у каждой СУБД существует команда, которая сообщает, что начат процесс резервного копирования. Синтаксически эта команда может выглядеть по-разному:

- в Oracle это две отдельные команды `ALTER DATABASE BEGIN BACKUP` и `ALTER TABLESPACE BEGIN BACKUP`;
- в PostgreSQL — функция `pg_start_backup()`;
- в Microsoft SQL Server и Db2 подготовка к резервному копированию выполняется неявно в процессе выполнения команды `BACKUP DATABASE`;

[1] Если не останавливать экземпляр PostgreSQL сигналом `SIGQUIT`, то файлы журналов при запуске экземпляра не понадобятся, но при холодном резервном копировании в этом надо убедиться.

- в MySQL Enterprise, Percona Server for MySQL, Cassandra и MongoDB подготовка неявно выполняется внешней утилитой — mysqlbackup, Percona XtraBackup, OpsCenter и Ops Manager соответственно.

Несмотря на синтаксические различия, процесс подготовки в похожих платформах включает в себя одни и те же шаги. Набор шагов зависит от типа дисковых структур.

Вот как выглядит подготовка к резервному копированию в СУБД с изменяемыми дисковыми структурами, то есть во всех традиционных дисковых реляционных системах:

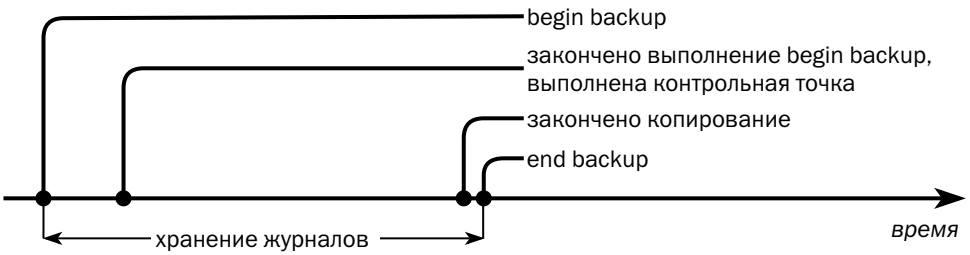
1. Запоминается момент начала резервного копирования; резервная копия должна будет содержать журналы базы данных начиная с этого момента.
2. Выполняется контрольная точка, то есть все изменения, которые произошли в страницах данных до запомненного момента, сбрасываются на диск. Это гарантирует, что журналы до момента начала резервного копирования при восстановлении не потребуются.
3. Включается особый режим журналирования: если страница данных изменилась в первый раз после загрузки с диска, то, вместо того чтобы записывать в журнал изменения страницы, база запишет туда страницу целиком. При выполнении подготовительной процедуры все страницы вытесняются на диск, и поэтому при первом изменении страница всегда будет записана в журнал целиком. Если в процессе резервного копирования страница снова будет вытеснена на диск, то в некоторых платформах следующее её изменение также приведёт к появлению в журнале полной копии страницы. Это гарантирует, что даже если при копировании файла с данными страница сохранится некорректно, применение журнала вернёт ей корректное состояние.
4. Блокируется изменение заголовков файлов данных, то есть тех областей, изменения которых не отражаются в журналах. Это гарантирует, что заголовок будет скопирован корректно, а потом к файлу данных будут корректно применены журналы.

Когда все перечисленные выше действия выполнены, можно копировать файлы данных средствами операционной системы — `cp`, `rsync` и другими. Включение режима резервного копирования снижает производительность базы данных: во-первых, увеличивается объём журналов, а во-вторых, если вдруг в режиме резервного копирования произойдёт сбой экземпляра, восстановление будет более продолжительным, т. к. заголовки файлов данных не обновляются, и экземпляр

будет вынужден накатывать все журналы с момента начала резервного копирования. Чем быстрее резервное копирование закончится, тем лучше для базы данных, поэтому здесь уместно применение таких средств, как снимок (snapshot) файловой системы или разрыв зеркала (BCV) в дисковом массиве. Одни СУБД (Oracle, PostgreSQL) оставляют администратору возможность самостоятельно выбрать способ копирования, другие (Microsoft SQL Server) предоставляют интерфейс для интеграции собственных утилит резервного копирования с механизмами файловых систем или СХД.

По окончании резервного копирования нужно перевести базу данных обратно в обычное состояние. В Oracle это выполняется командой `ALTER DATABASE END BACKUP` (или `ALTER TABLESPACE END BACKUP`), в PostgreSQL — вызовом функции `pg_stop_backup()`, а в других базах — внутренними подпрограммами соответствующих команд или внешних сервисов.

Вот как выглядит временна́я диаграмма процесса резервного копирования:



- Подготовка к резервному копированию (begin backup) занимает время, иногда значительное. Даже если используются зеркальные тома или файловые системы с возможностью изготовления снимков, процесс резервного копирования не будет мгновенным.
- Вместе с файлами данных необходимо сохранить журналы начиная с момента начала подготовки к резервному копированию и заканчивая моментом возврата базы в нормальное состояние.
- Восстановиться из этой резервной копии можно **на момент возврата базы в нормальное состояние**, то есть на момент end backup. Восстановление на более ранний момент невозможно.

С базами данных, использующими неизменяемые структуры данных (снимки памяти, LSM-деревья), ситуация проще. Подготовка к резервному копированию состоит из следующих шагов:

1. Данные из памяти сбрасываются на диск.
2. Фиксируется список файлов, попадающих в резервную копию. До тех пор, пока процесс резервного копирования не закончится, экземпляру запрещено удалять эти файлы из базы, даже если они становятся не нужны.

По сигналу об окончании резервного копирования экземпляр с неизменяемыми структурами снова может удалять из базы ненужные файлы.

10.4. Восстановление на точку

Резервная копия позволяет восстановить состояние базы данных на момент, когда завершилась команда возврата из режима резервного копирования. Однако авария, после которой потребуется восстановление, может произойти в любой момент. Восстановление состояния БД на произвольный момент называется «восстановлением на точку» (point-in-time recovery).

Чтобы обеспечить такую возможность, следует сохранять журналы БД начиная с момента окончания резервного копирования, а в процессе восстановления продолжить применять журналы к восстановленной копии. После того как БД восстановлена из резервной копии на момент окончания копирования, состояние базы (файлов и кешированных страниц) гарантированно корректно, поэтому особый режим журналирования не нужен. Применяя журналы до нужного момента, можно получить состояние базы данных на любую точку во времени.

Особо обратим внимание на то, что однозначное восстановление на точку возможно только для линеаризуемой системы, то есть системы, где порядок транзакций полностью определён. Инструментарий резервного копирования современных монолитных баз всегда предоставляет возможность восстановления на точку. В инструментах для резервного копирования распределённых баз данных обычно присутствуют какие-либо ограничения: либо восстановление возможно не на любую точку, либо оно требует вмешательства администратора.

Если скорость восстановления резервной копии ограничена лишь пропускной способностью диска, то скорость применения журналов обычно ограничена производительностью процессора. В основной базе данных изменения происходят параллельно, а при восстановлении все изменения выполняются последовательно — в порядке чтения из журнала. Таким образом время восстановления линейно зависит от того, насколько далеко точка восстановления отстоит от точки

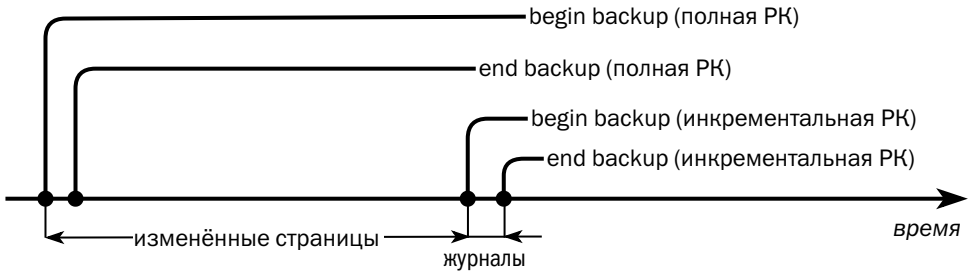
окончания резервного копирования. Из-за этого полные резервные копии приходится делать довольно часто — от еженедельного копирования баз с небольшой транзакционной нагрузкой до ежедневного резервирования высоконагруженных баз.

10.5. Инкрементальное резервное копирование

Чтобы ускорить восстановление на точку, хотелось бы иметь возможность выполнять резервное копирование как можно чаще, но при этом не занимать лишнего места на дисках и не нагружать базу.

Решение задачи — инкрементальное резервное копирование, то есть копирование только тех страниц данных, которые изменились с момента изготовления предыдущей резервной копии. Инкрементальное резервное копирование имеет смысл только для СУБД, использующих изменяемые структуры данных.

Тонкий момент заключается в том, что в инкрементальную копию включаются страницы, изменённые не с момента окончания предыдущего резервного копирования, а с момента его начала:



При наличии инкрементальной копии процесс восстановления на точку выглядит следующим образом:

- восстанавливаются файлы данных из полной резервной копии;
- поверх полной копии восстанавливаются файлы данных из инкрементальной копии;
- новые образы страниц, изменённых с момента начала снятия полной резервной копии, есть в инкрементальной копии, поэтому нет необходимости накатывать журналы из полной копии; достаточно накатить только журналы с начала последнего инкрементального резервного копирования до точки восстановления.

Инкремент может отсчитываться как от полной резервной копии (кумулятивная копия), так и от любой предыдущей копии (дифференциальная копия).

В случае наличия нескольких дифференциальных копий они восстанавливаются по очереди от самой старой к самой новой. По окончании восстановления накатываются только журналы из последней дифференциальной копии. Так, например, для восстановления состояния базы на точку между T_3 и T_4 необходимо восстановить две инкрементальные копии.

Наличие кумулятивной копии ускоряет процесс восстановления: очевидно, что её объём меньше, чем объём нескольких дифференциальных копий, потому что некоторые страницы изменились по нескольку раз, и каждая дифференциальная копия содержит свою версию страницы. Так, на диаграмме выше администратор в момент T_4 решил снять кумулятивную копию, и для восстановления на точку после T_4 достаточно будет восстановить только её одну.

К сожалению, единой терминологии для дифференциальных и кумулятивных копий не существует, и разные производители используют разные термины:

	Дифференциальная	**Кумулятивная**
Oracle	Differential	Cumulative
Postgres Pro	Incremental	—
Microsoft SQL Server	—	Differential
IBM Db2	Delta	Incremental
MySQL Enterprise	Incremental	Differential
Percona XtraBackup	Incremental	Incremental

Есть три способа создания инкрементальной копии:

1) создание полной копии и вычисление разницы с предыдущей полной копией;
2) разбор журналов, создание списка изменённых страниц и резервирование страниц, включённых в список;
3) запрос изменённых страниц в базе данных.

Первый способ экономит дисковое пространство, но не решает задачу снижения нагрузки на базу данных. Более того, если у нас есть полная резервная копия, то превращать её в инкрементальную бессмысленно, т. к. восстановление полной копии быстрее, чем восстановление предыдущей полной копии и инкремента. Задачу экономии дискового пространства при таком подходе лучше переложить на специальные компоненты со встроенными механизмами дедупликации. Это могут быть как специальные СХД (Dell/EMC[2] DataDomain, HPE StorageWorks VLS, вся линейка NetApp), так и программные продукты (ZFS, Veritas NetBackup PureFile, Windows Server Data Deduplication).

Второй и третий способы отличаются механизмом определения списка изменённых страниц. Для разбора журналов необходимо знать структуру журнальных файлов. Спросить у самой базы, какие именно страницы изменились, проще всего, но для этого ядро СУБД должно иметь функциональность отслеживания изменённых блоков (block change tracking). Эффективность того или иного способа зависит от характера нагрузки на базу данных.

Впервые функциональность инкрементального резервного копирования была создана в ПО Oracle Recovery Manager (RMAN), появившемся в релизе Oracle 8i. Поначалу для определения изменённых блоков приходилось анализировать журнал БД, а начиная с релиза 10g Oracle реализовал отслеживание изменённых блоков в ядре платформы.

Утилита `pg_probackup`, разработанная российской компанией Postgres Professional для инкрементального копирования данных PostgreSQL, как и ранние версии RMAN, определяет изменённые страницы путём анализа журнала. Однако компания поставляет и СУБД Postgres Pro, которая включает расширение ptrack, отслеживающее изменение страниц. При использовании `pg_probackup` с СУБД Postgres Pro утилита запрашивает изменённые страницы у экземпляра БД — так же, как и современные версии Oracle Recovery Manager.

Microsoft SQL Server, как и Oracle, отслеживает изменённые страницы, но позволяет делать только полные и кумулятивные резервные копии командой `BACKUP`.

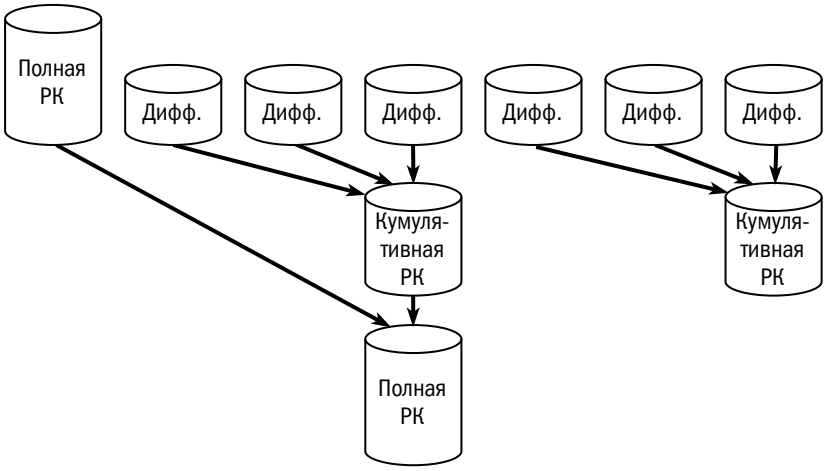
В Db2 есть возможность отслеживания измененных страниц, но по умолчанию она выключена. После включения Db2 позволит делать полные, дифференциальные и кумулятивные резервные копии.

Важное отличие описанных в этом разделе средств (кроме `pg_probackup`) от файловых средств резервного копирования в том, что они запрашивают образы страниц у экземпляра БД, а не читают данные с диска самостоятельно. Недостаток такого подхода — небольшая дополнительная нагрузка на экземпляр.

Однако этот недостаток с лихвой компенсируется тем, что прочитанная страница всегда корректна, поэтому нет необходимости во включении на время резервного копирования особого режима журналирования.

Ещё раз обратите внимание, что наличие инкрементальных копий не отменяет требований к наличию журналов для восстановления на произвольную точку во времени. Поэтому в промышленных базах данных журналы постоянно переписываются на внешний носитель, а резервные копии, как полные, так и инкрементальные, создаются по расписанию.

В таких утилитах, как RMAN или `pg_probackup`, возможность инкрементального резервного копирования доведена до совершенства: в них не обязательно регулярно выполнять полное резервирование. Система единственный раз создаёт полную копию базы данных, а потом делает только инкрементальные копии. Впоследствии утилиты позволяют объединять копии, создавая новые полные копии из инкрементальных. Такая политика носит название «вечного инкремента» (incremental forever).



Наилучшей на сегодня реализацией идеи инкрементального резервного копирования является программно-аппаратный комплекс (в терминологии Oracle — engineered system) Zero Data Loss Recovery Appliance — специализированное решение Oracle для резервного копирования собственной БД. Комплекс представляет собой кластер серверов с большим объёмом дисков, на которые установлена модифицированная версия ПО Recovery Manager, и может работать как с другими программно-аппаратными комплексами Oracle (Database Appliance, Exadata, SPARC Supercluster), так и с базами Oracle на традиционной инфраструктуре.

В отличие от многих похожих вопросов, вопрос «какой метод резервного копирования лучше» имеет однозначный ответ — лучше всего инкрементальное копирование с использованием родной для платформы утилиты.

Для администратора БД гораздо более важными являются вопросы выбора стратегии резервного копирования и интеграция средств резервирования баз данных в корпоративную инфраструктуру. Но эти вопросы выходят за рамки книги.

Литература

- Oracle Database Backup and Recovery User's Guide.
  docs.oracle.com/en/database/oracle/oracle-database/19/bradv/
- Zero Data Loss Recovery Appliance.
  docs.oracle.com/en/engineered-systems/zero-data-loss-recovery-appliance/12.2/ampdb/
- Anastasia Lubennikova. Advanced PostgreSQL Backup & Recovery Methods.
  www.slideshare.net/AnastasiaLubennikova/advanced-backup-methods-postgrescern
- MySQL Enterprise Backup User's Guide.
  dev.mysql.com/doc/mysql-enterprise-backup/8.0/en/mysqlbackup.backup.html
- Андрей Криушин. Что делает Oracle при begin backup?
  www.sql.ru/forum/78296/chto-delaet-orakl-pri-begin-backup?mid=562392#562392
- pg_probackup.
  postgrespro.ru/docs/enterprise/current/app-pgprobackup
- Db2 incremental backup and recovery.
  www.ibm.com/support/knowledgecenter/SSEPGG_11.5.0/com.ibm.db2.luw.admin.ha.doc/doc/c0006069.html
- SQL Server Backup.
  docs.microsoft.com/ru-ru/sql/t-sql/statements/backup-transact-sql?view=sql-server-ver15
- PostgreSQL Best Practice. Introduction to Cold Backup and Restoration.
  www.alibabacloud.com/forum/read-427
- MongoDB Ops Manager. Backup process.
  docs.opsmanager.mongodb.com/current/core/backup-overview/
- DataStax Enterprise OpsCenter User's Guide.
  docs.datastax.com/en/opscenter/6.8/opsc/online_help/services/opscBackupService.html
- Tarantool. Руководство пользователя.
  www.tarantool.io/ru/doc/2.5/book/admin/backups/

Часть VI

Эксплуатация баз данных

Глава 11
Управление базой данных

> Известная неисправность лучше неизвестной, а потому опасайся «исправных» машин.
>
> *Павел Иевлев, УАЗДао 2:28*

Управление базой данных, как и любым другим компонентом инфраструктуры, заключается в постоянном наблюдении за системами и процессами, фиксации результатов наблюдения и принятии управленческих решений на основании этих результатов.

11.1. Механизмы управления

Мониторинг и журналирование

Для наблюдения за информационными системами и управления ими используются несколько механизмов, которые дополняют друг друга и могут быть объединены на уровне рабочего места администратора. Важнейшие из этих механизмов — мониторинг и журналирование.

Мониторинг — сбор и обработка метрик, характеризующих состояние системы. Метрики обладают следующими свойствами:

- Метрика принадлежит постоянно существующему объекту, например серверу, узлу, очереди. Метрика не может принадлежать эфемерному объекту — запросу, сеансу и т. п.
- Метрика имеет численное значение (или вектор численных значений). Так, например, «объём занятого временного пространства» может быть метрикой, а «переполнение временного пространства» — нет. Однако если по каким-то причинам временное пространство экземпляра ограничено и переполнение является частым событием в многопользовательской аналитической системе, то полезной метрикой может оказаться «количество запросов, прерванных из-за переполнения временного пространства».

- Метрика часто и не всегда предсказуемо изменяется. Так, например, «объём использованной оперативной памяти» может быть метрикой, а «количество дней до истечения лицензии» — нет, т. к. первое значение может изменяться много раз за секунду, а второе — раз в сутки, причём абсолютно предсказуемо. Формального деления на «часто» и «редко» не существует, при классификации «редких» и «частых» событий приходится полагаться на здравый смысл.

Само по себе значение метрики не является основанием для каких-либо выводов или действий. Так, например, факт «в таблице 100 000 000 строк» ни о чём не говорит, если не принимать во внимание динамику изменения количества строк, максимальный объём таблицы и т. д.

Журналирование — фиксация событий, представляющих потенциальный интерес для администратора, в специально выделенном месте — журнале событий. В контексте баз данных следует различать журнал транзакций, обеспечивающий надёжность хранения данных на диске, и журнал событий, предназначенный для фиксации системных событий и информирования администратора. Так же, как и для журнала транзакций[1], для него нет устоявшегося термина. Он может называться event log, alert log, system log, server log, message log…

События, отражённые в журнале, обладают следующими свойствами:

- Событие записывается в журнал непосредственно в тот момент, когда оно происходит. Например, если запрос был прерван из-за переполнения временного пространства, то событие прерывания будет записано в журнал. Мониторинг при этом может не показать переполнения, если в начале и в конце интервала опроса объём временного пространства не превышал максимально допустимого значения.
- Если происходит несколько событий, представляющих интерес, то в журнале отражается каждое событие. Например, если в описанном выше примере в одном временно́м интервале было аварийно прервано три запроса, то в журнале появится три события. В дальнейшем внешняя по отношению к журналу система может формировать метрики мониторинга на основе обработанных, в том числе и агрегированных, журнальных записей.

Кроме событий, в журнале уместно отражать редко меняющиеся метрики либо предупреждения системы самодиагностики. Например, не нужно делать срок

[1] Список терминов, используемых для обозначения журнала транзакций, приведён в главе 5 «Гарантии корректности данных».

действия лицензии метрикой мониторинга, т. к. значение этой метрики изменяется на единицу раз в сутки, и передавать это значение каждые 10 секунд бессмысленно. Вместо этого уместнее при наступлении критического срока (например, за 30 дней до истечения лицензии) раз в сутки записывать предупреждающее сообщение в журнал.

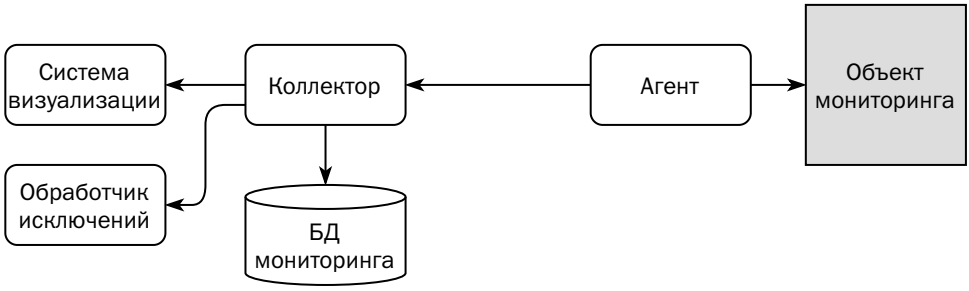
Несколько слов о мониторинге

Каждое значение метрики мониторинга состоит из временно́й метки, названия метрики и величины метрики.

В подавляющем большинстве случаев значения имеют один из следующих типов:

- измерение (gauge) — значение показателя, измеренное в текущий момент, например, текущий объём занятой оперативной памяти;
- счётчик (counter) — значение показателя накопленным итогом с момента начала измерения (или с момента сброса счётчика), например объём трафика, принятого по сети;
- гистограмма (histogram) — набор значений, показывающих распределение некоторой величины, например времени обработки пользовательского запроса;
- состояние (state set) — дискретное состояние объекта мониторинга, например «свободен», «занят», «выключен».

Концептуальная архитектура системы мониторинга изображена на рисунке:



Система состоит из следующих компонентов:

- Агент (agent) — небольшая программа, установленная, как правило, максимально близко к объекту мониторинга. В задачи агента входят сбор метрик с конкретного объекта, форматирование метрик и передача их в общую систему. В случае отсутствия связи с коллектором агент может какое-то время

накапливать информацию. Возможны и безагентские схемы мониторинга, когда коллектор сам обращается к объектам мониторинга и извлекает из них информацию.

- Коллектор (collector) — основной компонент системы, в задачи которого входит сбор метрик со всех объектов мониторинга, сохранение этих метрик в постоянное хранилище и дальнейшая маршрутизация полученной информации.
- База данных — постоянное хранилище метрик мониторинга, которое требуется для анализа истории. Коммерческие системы мониторинга с длинной историей используют реляционные БД, а более современные системы — БД временны́х рядов.
- Система визуализации служит для наглядного представления результатов мониторинга.
- Обработчик исключительных ситуаций (alerter) анализирует значения метрик и в случае обнаружения каких-либо проблем генерирует управляющие воздействия. Воздействия могут быть как элементарными (отправка администратору SMS или письма), так и достаточно сложными, включая автоматическое изменение параметров объекта мониторинга.

Изучая вопросы мониторинга, вы рано или поздно встретите упоминание ряда продуктов:

- платформы сбора и обработки информации с открытым исходным кодом: Prometheus, Zabbix, Nagios;
- системы визуализации, используемые в том числе и для данных мониторинга: Grafana, Kibana;
- базы данных временны́х рядов: Prometheus, kdb+, TimescaleDB, VictoriaMetrics;
- комплексные коммерческие системы мониторинга, содержащие все компоненты, включая модули управляющего воздействия на объекты мониторинга — Dynatrace, AppDynamics, Splunk, NewRelic, SolarWinds Observability;
- вендорские продукты для комплексного управления инфраструктурой баз данных, включая мониторинг, — Oracle Enterprise Manager Cloud Control, Microsoft System Center Operations & Monitoring, IBM Db2 Data Management Console, MongoDB Ops Manager и другие.

Мониторинг информационных систем — тема для отдельной большой книги, поэтому в оставшейся части главы мы сосредоточимся на метриках, важных именно для СУБД, и на том, как эти метрики собирать.

11.2. Мониторинг баз данных

Мониторинг доступности экземпляра

Администратору баз данных важно знать, что экземпляр запущен и может обрабатывать пользовательские запросы. Для того чтобы отслеживать доступность экземпляра, система мониторинга периодически выполняет какой-либо простой запрос — например, запрашивает текущее время[1]:

```
select sysdate from dual
```

Метрикой мониторинга является время выполнения контрольного запроса.

Гораздо более информативной метрикой было бы время исполнения какой-либо бизнес-операции. Такой мониторинг также применяется, но метрика должна быть разработана в приложении. Во-первых, приложение должно выполнять бизнес-операцию целиком и замерять время исполнения всех её этапов, а не только этапа записи в базу данных. А во-вторых, логика приложения должна отличать настоящие объекты от фиктивных, созданных процессами мониторинга.

Мониторинг сервера

Для работы базы данных, как и для работы любого программного обеспечения, важно состояние сервера, на котором запущен экземпляр. Сами по себе метрики сервера, снимаемые с операционной системы, не могут в полной мере охарактеризовать состояние экземпляра БД, однако неадекватные значения этих метрик сигнализируют о потенциальных проблемах.

Далее рассмотрим основные метрики, которые важно отслеживать на всех серверах баз данных.

Загрузка центрального процессора, состоящая из трёх компонентов:

- нагрузка, создаваемая пользовательскими вычислениями (`%user`);
- выполнение системных вызовов в ядре операционной системы (`%system`);
- ожидание ввода-вывода (`%iowait`).

Рекомендуется, чтобы суммарная загрузка процессора в штатном режиме работы не превышала 70 %: во-первых, такая нагрузка не выводит сервер из комфортного температурного режима, а во-вторых, позволяет обрабатывать пиковые нагрузки.

[1] Приведённый запрос соответствует диалекту SQL Oracle.

Больша́я доля ожидания ввода-вывода может сигнализировать о том, что дисковая подсистема не справляется с нагрузкой.

Объём используемой оперативной памяти. Дисковые платформы, используемые для обработки транзакций, используют оперативную память главным образом под буферный кеш. Размер этого кеша задаётся параметрами конфигурации экземпляра или выбирается экземпляром автоматически и при стабильной нагрузке на базу данных практически не изменяется. Аналитические системы активно используют память для сортировки и размещения хеш-таблиц, и если объём используемой памяти растёт быстрее, чем объём данных, это может быть сигналом о необходимости оптимизации запросов. Для базы данных в памяти исчерпание оперативной памяти означает невозможность сохранять новые данные, если только настройки не предусматривают автоматическое удаление наименее востребованных данных.

Подкачка (swapping). Практически универсальное эмпирическое правило: на сервере базы данных не должно быть подкачки. Одна из главных задач, решаемых разработчиками СУБД, — эффективное управление оперативной и дисковой памятью, и вмешательство операционной системы в работу менеджера памяти СУБД чревато критическим снижением производительности. Появление подкачки говорит о необходимости изменить настройки экземпляра или конфигурацию сервера.

Дисковая активность порождает множество метрик мониторинга — достаточно посмотреть вывод команд `vmstat` или `sar`. По-настоящему важной для сервера БД является длина очереди запросов на ввод-вывод. Если очередь растёт, значит, подсистема ввода-вывода не справляется с нагрузкой. Сам по себе объём прочитанных или записанных данных ни о чём не говорит, если только график не представляет собой бесконечную «полку», показывающую, что диск работает на пределе возможностей.

Объём свободного дискового пространства меняется не так быстро, как нагрузка на компоненты системы, однако отслеживать его необходимо. Большинство файловых систем требуют, чтобы часть тома оставалось незанятой, в противном случае выполнение файловых операций резко замедляется. А полное исчерпание дискового пространства грозит серьёзными проблемами вплоть до аварийной остановки экземпляра.

Сетевая активность также подлежит мониторингу, однако по опыту автора на сервере базы данных исправная сеть никогда не является узким местом, даже в том случае, когда ETL-процесс считывает по сети целиком несколько таблиц.

Мониторинг состояния экземпляра

Метрики, собранные с операционной системы, дают общее представление о функционировании программы, выполняемой на сервере, но не могут дать никакой детальной информации. За подробностями следует обращаться к самой программе, и все СУБД так или иначе такую информацию предоставляют.

Единого списка метрик, формата представлений этих метрик или типа интерфейса не существует. Для экспорта метрик экземпляр БД может использовать разные интерфейсы:

- специальные представления, выборка из которых показывает состояние экземпляра (например, в Oracle такие представления имеют префикс `V$`, в Microsoft SQL Server — `dm_`, а в PostgreSQL — `pg_stat_`);
- функции;
- интерфейс JMX (Java Management Extensions), если код платформы исполняется в виртуальной машине Java.

Для формирования метрик используются агенты, разработанные под конкретную платформу. В терминологии одной из наиболее популярных платформ мониторинга Prometheus такие специализированные агенты называются экспортёрами (exporters).

Для мониторинга экземпляра БД могут использоваться, например, следующие метрики:

- количество пользовательских сеансов и количество процессов;
- количество активных транзакций;
- объём считанных и записанных данных, а также частота попадания в кеш;
- объём журнальной информации и частота переключения файлов журнала;
- количество блокировок;
- ожидания запросов, то есть список ресурсов, которых запросам не хватает для завершения, и время доступа к этим ресурсам.

Выбор метрик зависит от режима нагрузки на базу данных и специфики решаемых задач, а также от предпочтений администратора. Одни и те же проблемы могут быть обнаружены разными способами.

С развитием интеллектуальных систем мониторинга с мощными обработчиками исключительных ситуаций поиск этих ситуаций во многом автоматизирован. Оператор мониторинга видит лишь индикатор, светящийся зелёным при нормальном функционировании экземпляра и красным в случае каких-либо проблем. Тем не менее рекомендуется понимать, какие именно метрики собирает

система мониторинга и как она их обрабатывает. Кроме того, надо помнить, что сбор метрик мониторинга может влиять на производительность экземпляра.

11.3. Настройка производительности

Использование динамических представлений

Представления, используемые для мониторинга экземпляра, могут быть использованы и для анализа проблем производительности. Поскольку инструменты анализа используют те же данные, что и системы мониторинга, работу по поиску проблем иногда называют «историческим мониторингом» или даже «стратегическим мониторингом».

Запрос текущего состояния представлений показывает состояние экземпляра в краткий миг запроса и никак не описывает его работу в течение длительного времени. Однако если несколько раз сохранить снимки состояний и сравнить несколько последовательных снимков между собой, то можно проанализировать, чем занимался экземпляр в промежутке между их изготовлением.

Впервые такая идея пришла в голову разработчикам Oracle, которые в 1998 году включили в релиз Oracle 8i набор скриптов под названием Statspack. Сегодня аналогичные инструменты существуют для многих популярных платформ:

Платформа	Инструмент
Oracle	AWR (average workload repository) — развитие Statspack; ASH (active session history) — дополнительный механизм в ядре, позволяющий получать информацию о выполняемых запросах в реальном времени
Db2	monreport
MySQL	PERFORMANCE_SCHEMA
PostgreSQL	pg_profile, pgpro_pwr[1]
Microsoft SQL Server	Сбор данных (Data collection)

Для ряда платформ (например, Cassandra или MongoDB) существуют аналогичные инструменты, разрабатываемые сторонними компаниями или независимыми разработчиками. Зачастую авторы вслед за Oracle называют свои инструменты «AWR», и именно по этому ключевому слову следует искать их в интернете.

[1] Разработка компании Postgres Professional, поставляется в составе продукта Postgres Pro.

Отчёт за период содержит ряд разделов, сообщающих администратору, на что следует обратить внимание.

Общие метрики работы экземпляра — количество установленных и прерванных сеансов, количество взаимоблокировок, количество подтверждённых и отменённых транзакций, количество выполненных и скомпилированных запросов — характеризуют качество приложения. Если, например, экземпляру приходится каждый раз разбирать и компилировать запрос перед выполнением, то проблема не в базе данных, а в реализации бизнес-логики.

События ожидания, на долю которых приходится наибольшее время, показывают узкое место системы. Если, например, запросы всё время ждут записи журнала, то бессмысленно увеличивать процессорную мощность или буферный кеш — в первую очередь следует решить проблему скорости записи.

Список запросов, на долю которых приходится максимальная нагрузка, позволяет указать разработчикам и администраторам, какие запросы следует оптимизировать в первую очередь.

Список объектов (таблиц и индексов), а также физических объектов — файлов и табличных пространств, на долю которых приходится наибольшее количество операций ввода-вывода. Анализ этих объектов позволяет оптимизировать размещение данных на диске.

Платформы, даже схожие между собой идеологически, достаточно сильно отличаются друг от друга в деталях, поэтому и отчёты о производительности различны. Содержание отчётов зависит от архитектуры платформы: так, например, отчёт `pg_profile` о производительности PostgreSQL содержит информацию о процессе очистки (vacuum), а отчёт AWR о производительности Oracle — об использовании пространства отката (undo space). Кроме того, содержание отчёта напрямую зависит от количества собираемых метрик, которое в свою очередь характеризует коммерческую зрелость платформы: так, например, отчёты инструментов, созданных сторонними разработчиками, о производительности Cassandra или MongoDB значительно короче и беднее, чем отчёты Oracle, PostgreSQL или Microsoft SQL Server.

Другие методы настройки производительности

Помимо анализа метрик мониторинга, некоторые платформы предоставляют дополнительные средства настройки производительности — трассировку и автоматические советчики.

Трассировка (tracing) — вывод отладочной информации о сеансе или конкретном запросе. Информация может выводиться в консоль администратора или в файл на сервере базы данных. Отладочная информация содержит события с параметрами и временны́ми метками. Примерами таких событий могут служить начало и окончание разбора запроса, начало выполнения запроса, окончание выполнения запроса. У события окончания выполнения запроса, например, параметрами могут быть план выполнения, количество прочитанных блоков, набор событий ожидания, значения связанных переменных и т. д.

Трассировка может использоваться как разработчиком в режиме отладки приложения в среде тестирования, так и администратором промышленной системы. В реляционных платформах трассировка позволяет, например, выявить несоответствия между ожидаемыми планами запросов и реальными планами, которые оптимизатор выбирает исходя из объёма данных и нагрузки на систему.

Советчик (advisor) — встроенный в ядро платформы механизм, прогнозирующий производительность системы под текущей нагрузкой в зависимости от параметров конфигурации. Если существует конфигурация экземпляра, при которой производительность приложения выше текущей, советчик предложит изменить настройки, а «неадминистрируемая» Oracle Autonomous DB может даже изменить конфигурацию самостоятельно.

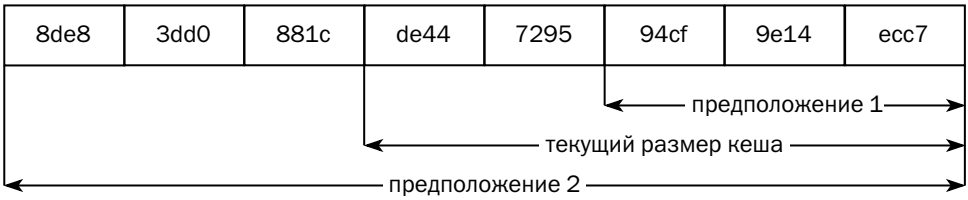
Советчики — сложная функциональность, встречающаяся только в зрелых коммерческих платформах, таких как Oracle, Db2, Microsoft SQL Server.

Основные изменения, которые могут предлагать автоматические советчики, — изменение параметров памяти и создание новых индексов.

Встретив часто исполняемый запрос, советчик предполагает, что запрос может выполняться быстрее, если будет построен дополнительный индекс. Затем советчик моделирует параметры индекса — количество блоков, высоту дерева и другие, при этом строить сам индекс не требуется. Оптимизатор строит планы запросов с использованием новых индексов, а советчик рассчитывает, какой была бы нагрузка на сервер, если бы запрос действительно выполнялся по альтернативному плану. Важно понимать, что ускорение одного запроса совсем не обязательно положительно скажется на производительности всего приложения, поскольку дополнительный индекс не только ускоряет чтение в некоторых случаях, но и замедляет обновление таблицы и занимает пространство в буферном кеше.

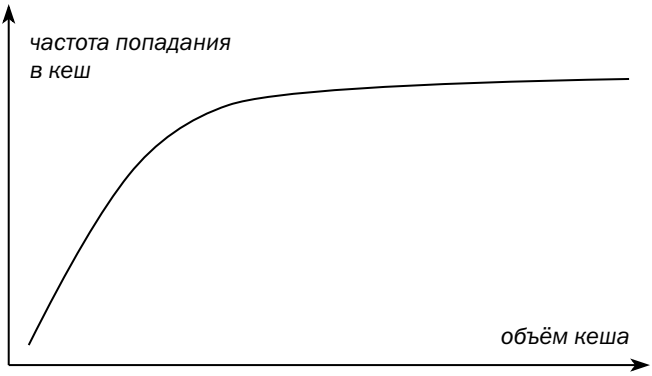
Если в течение заданного времени анализ показывает общее повышение производительности, советчик записывает рекомендацию по созданию индекса в специальную таблицу, служащую интерфейсом пользователя.

Для определения оптимального размера кеша (буферного кеша, кеша запросов и др.) советчик составляет список его возможных размеров и держит в памяти список прочитанных страниц или объектов, которые могли бы находиться в кеше с разными настройками размера. При запросе страницы (или другого объекта) процесс, собирающий метрики экземпляра, увеличивает счётчик попаданий в кеш только в том случае, если страница действительно прочитана из кеша, а советчик увеличивает счётчик для тех возможных настроек, при которых страница попала бы в кеш.



Обратите внимание, что предполагаемые размеры могут быть как больше, так и меньше текущего объёма.

Практика показывает, что объём «горячих» (постоянно используемых) данных для каждого приложения ограничен, поэтому увеличение кеша эффективно лишь до некоторого предела.



При настройке экземпляра стоит воспользоваться статистикой советчика, возможно, даже уменьшив размер буферного кеша и перераспределив часть памяти в пользу других структур.

Литература

- OpenMetrics specification.
  github.com/OpenObservability/OpenMetrics/blob/main/specification/OpenMetrics.md
- Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy Lohman. Db2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes.
  wwwiti.cs.uni-magdeburg.de/~eike/selftuning/sources/db2_advisor.pdf

Глава 12
Оборудование

> Экономящий же на инструменте подобен выбившему себе зубы, чтобы не тратиться на зубную пасту.
>
> *Павел Иевлев, УАЗДао 1:4*

Выбор платформы для хранения и обработки данных невозможен без понимания свойств инфраструктуры, обеспечивающей функционирование этих платформ. В данной главе кратко рассмотрим основные понятия, относящиеся как к инфраструктуре, так и к информационным системам в целом, а также совершим небольшой экскурс в историю серверного оборудования.

12.1. Серверы

Мейнфреймы

Первыми покупателями баз данных были крупные предприятия, требовавшие высочайшей скорости обработки и доступности данных. Для эксплуатации таких систем использовались мейнфреймы.

Сам термин mainframe используется с начала XX века для обозначения больших стоек с телекоммуникационным оборудованием. Однако история мейнфреймов отсчитывается с 7 апреля 1964 года, когда компания IBM выпустила пресс-релиз платформы System/360, назвав словом mainframe главную процессорную стойку.

Среди технических решений, обеспечивающих непревзойдённую производительность и надёжность мейнфреймов, следует отметить:

- использование мощных центральных процессоров с максимально возможной тактовой частотой, даже если приходится применять для них системы водяного охлаждения;
- использование всевозможных сопроцессоров для специфических задач — например, вычислений с плавающей или фиксированной точкой, ввода-вывода, криптографии;

- использование интерфейсов, позволяющих наращивать мощность мейнфреймов, добавляя дополнительные блоки к уже имеющимся (например, IBM Parallel Sysplex);
- дублирование всех компонентов, включая центральные процессоры.

Время наработки мейнфрейма на отказ исчисляется десятилетиями.

Системы управления базами данных были включены в состав системного программного обеспечения, поставляемого вместе с мейнфреймами. Сейчас реляционные СУБД для мейнфреймов выпускают IBM и Oracle. Кроме того, сохраняют популярность нереляционные платформы — например, ADABAS от Software AG.

Эпоха господства мейнфреймов продолжалась с 60-х по 80-е годы XX столетия. Лидерами рынка мейнфреймов были «IBM и семь гномов», или «IBM и пучок производителей»[1].

Аналитики уже не раз предсказывали «отключение последнего мейнфрейма», но тем не менее IBM продолжает выпуск мейнфреймов System z. Последняя на момент написания книги модель z16 была анонсирована 5 апреля 2022 года.

Несмотря на все преимущества, которые даёт использование мейнфреймов, автор считает, что системы хранения и обработки данных на их основе целесообразно строить только в тех организациях, где в инфрастуктуру мейнфреймов уже вложены значительные средства и накоплен опыт их эксплуатации. В остальных случаях следует предпочесть другое оборудование.

Мини-ЭВМ

Параллельно с мейнфреймами, предназначенными для больших предприятий, начиная с середины 60-х годов широкое распространение получили мини-ЭВМ (minicomputers). Термин «мини» означал всего лишь размер: мини-ЭВМ занимала одну или несколько стандартных 19-дюймовых стоек, в отличие от мейнфреймов, занимавших целые комнаты. Ведущими игроками в этом классе оборудования были DEC с семействами PDP и VAX, Hewlett-Packard с семействами HP 2100 и HP 3000 и IBM с System/38, превратившейся позже в AS/400.

Серверы 60-х годов не обладали вычислительной мощностью, достаточной для запуска реляционных платформ. Однако именно на ЭВМ PDP-11 в 1979 году появилась первая в мире коммерческая СУБД Oracle V2[2].

---

[1] BUNCH = Burroughs, UNIVAC, NCR, Control Data, Honeywell. Ещё два «гнома» — компании General Electric и RCA.

[2] Компания Relational Software, Inc., позже переименованная в Oracle Corporation, никогда не выпускала первой версии своей СУБД, чтобы внушить рынку мысль о зрелости продукта.

Среди пользователей мини-ЭВМ широкую популярность завоевал язык программирования MUMPS[1], разработанный в 1966 году для ЭВМ PDP-7. Это интерпретируемый язык, в который встроены средства мультипрограммирования (то есть возможность выполнять одновременно несколько задач), команды работы с разреженными массивами на диске (чем-то похожими на базы данных «семейства колонок») и механизм индексно-последовательного доступа к ним. По состоянию на начало 2000-х MUMPS, включая самую популярную реализацию — GT.M, широко использовался в информационных системах банков и медицинских учреждений, в том числе в России.

Мини-ЭВМ не дожили до наших дней, за исключением AS/400, пережившей несколько ребрендингов и доступной под именем IBM i. Такое поразительное долголетие обеспечивается архитектурой операционной системы OS/400, представляющей собой виртуальную машину, идеологически похожую на виртуальную машину Java. Виртуальная машина IBM i включает в себя и СУБД Db2/i, разделяющую с Db2 лишь торговую марку, но имеющую отдельную кодовую базу. Сегодняшняя IBM i работает на современных RISC-процессорах POWER и аппаратно не имеет ничего общего со своей прародительницей AS/400, но программное обеспечение, написанное полвека назад, запускается на ней без повторной компиляции.

Компании IBM мы обязаны также появлением термина mid-range servers (серверы среднего класса). Этим термином обозначались мини-ЭВМ, к которым применялись существенно более слабые требования по надёжности и производительности, чем к мейнфреймам.

Рекомендации автора относительно IBM i такие же, как и относительно мейнфреймов, — использовать только там, где уже накоплен многолетний опыт использования инфраструктуры IBM.

Открытые системы

90-е и 2000-е годы — время расцвета открытых систем, то есть систем с опубликованными спецификациями — от архитектуры процессоров и описания протоколов до физических размеров отдельных компонентов. Такой подход позволил существенно удешевить серверы за счёт конкуренции между производителями

[1] Это бэкроним, который расшифровывают как Massachusetts General Hospital Utility Multi-Programming System. Поскольку изначально язык был создан для автоматизации медицинских учреждений, его назвали «свинка», «mumps».

комплектующих. Правда, пострадала и производительность: разработчики открытых систем не могли по своему усмотрению добавлять компоненты, не соответствующие спецификации, а также по большей части избегали нестандартных компоновочных решений.

Ведущие производители открытых систем представлены в таблице ниже:

Производитель	Линейка	Процессор	ОС	Годы выпуска
DEC (позже Compaq и Hewlett-Packard)	DEC AXP; AlphaServer	Alpha	Digital UNIX; Tru64 UNIX	1992–2007
Hewlett-Packard	HP9000	PA-RISC	HP-UX	1996–2008
Hewlett-Packard (позже HPE Enterprise)	Integrity[1]	Intel Itanium[2]	HP-UX	2003–2025[3]
IBM	RS/6000; System p; pSeries; POWER systems	PowerPC; POWER	AIX	1990–
Sun Microsystems (позже Oracle)	SunFire; SPARCServer	SPARC	Solaris	1991–
Fujitsu	DS/90 7000; PrimePower; GranPower 7000	SPARC	Solaris	1991–2010

Архитектурно все перечисленные решения очень похожи между собой: это операционные системы, реализующие стандарт UNIX System V, на 64-битных RISC-процессорах[4].

Линейки включают в себя как высокопроизводительные серверы, по уровню надёжности близкие к мейнфреймам (hi-end), так и более дешёвые серверы среднего класса (mid-range). Класс сервера может быть как явно указан в названии (например, у HPE «Integrity Superdome» — сервер высшего класса надёжности, а «Integrity Blade» — более дешёвый), так и зашифрован в индексе (например,

[1] Сам бренд Integrity достался компании Hewlett-Packard при покупке Compaq, которая, в свою очередь, унаследовала его от Tandem Computers.

[2] С 2015 года под маркой Integrity выпускается также сервер Superdome X на базе процессоров Intel Xeon.

[3] Последний процессор Intel Itanium 9700 выпускался с 2017 по 2020 год. HPE с 2021 года не поставляет новых серверов на их базе, но гарантирует поддержку HP-UX на Itanium до 2025 года.

[4] Строго говоря, Intel Itanium построен на архитектуре EPIC (Explicitly Parallel Instruction Computing), являющейся наследницей архитектуры VLIW (Very Long Instruction Word). Однако во всех маркетинговых материалах этот процессор включают в группу RISC (Reduced Instruction Set Computer) наряду с процессорами SPARC и POWER.

в линейке IBM POWER 7 серверы 795 относились к классу hi-end, а 770 и 780 — к mid-range).

Общепризнанного списка критериев, позволяющих отнести сервер к классу hi-end, не существует, однако, как правило, отличительными чертами такого сервера являются:

- обработка сбоев, включая отказ любого процессора, без остановки сервера;
- возможность «горячей» (без отключения питания) замены любых компонентов, включая процессоры и память, а также реконфигурация сервера после ремонта «на лету», поддерживаемая не только оборудованием, но и программным обеспечением;
- возможность зеркалирования памяти;
- возможность аппаратного разделения сервера на домены (domains), то есть серверы меньшего размера.

В серверах среднего класса также реализованы различные решения, повышающие их доступность, — например, возможность «горячей» замены дисков и плат расширения, расчёт контрольных сумм для блоков памяти, дублирование шин и т. д. Для деления серверов на более мелкие домены применяются платформы виртуализации.

Для UNIX-серверов выпущено множество коммерческих реляционных платформ: Oracle, Db2, Sybase, Informix, Pervasive SQL, Progress...

До конца 2000-х годов коммерческая реляционная СУБД на одной из UNIX-систем была естественным и практически безальтернативным выбором для приложения, эксплуатируемого в финансовой компании или на промышленном предприятии. Однако сегодня и эта инфраструктура считается устаревшей (legacy[1]) и применяется только для тех приложений, перенос которых на другую инфраструктуру по каким-либо причинам нецелесообразен.

Семейство x86

Доминирование архитектуры x86 на рынке серверов — наглядный пример того, как удачное стечение обстоятельств и массовость производства оказываются важнее изначально правильных инженерных решений.

Процессор Intel 8086 появился в 1978 году. Его небывалая популярность обусловлена тем, что процессор Intel 8088, принадлежащий к этому же семейству,

[1] Как правило, слово legacy (наследие) не несёт негативной коннотации, но этот случай — исключение.

был выбран компанией IBM для разработки персонального компьютера — IBM PC. Спустя четыре десятилетия большая часть современных серверных процессоров поддерживают ту же систему команд, что их далёкий предшественник.

Compaq SystemPro, представленный в ноябре 1989 года, был первым сервером на базе процессора семейства x86: в нём были установлены два процессора Intel i386, а позднее — Intel i486. Единственной на тот момент операционной системой, способной использовать все возможности такого сервера, был SCO UNIX. Чуть позже, в сентябре 1992 года, IBM представила свою серверную линейку PS/2 Server[1] под управлением операционной системы OS/2.

Важной вехой в развитии x86 как серверной платформы стал выход в 1993 году операционной системы Windows NT 3.1. С этого момента появилась возможность запускать серверное программное обеспечение, включая СУБД, на обычных офисных компьютерах, по мере необходимости меняя их на «настоящие» серверы. К середине 90-х годов монополия Microsoft на рынке серверных операционных систем достигла настолько угрожающих размеров, что заставила Dell (производитель линейки серверов PowerEdge на базе процессоров x86 с 1996 года) и IBM объединить усилия по развитию и продвижению Linux как серверной операционной системы. Позднее к ним присоединилась и компания Hewlett-Packard, купившая Compaq. Развитие Linux, в свою очередь, вызвало взрывной рост популярности x86 как серверной платформы.

По отчётам компании Gartner[2] уже в 2005 году половина всей выручки от продажи серверного оборудования приходилась на серверы x86, и с тех пор их доля только растёт. Если учесть, что в среднем сервер на базе x86 в разы дешевле RISC-сервера и на порядок-другой дешевле мейнфрейма, очевидно, что сегодня подавляющее большинство серверов построены на платформе x86.

Часть серверов на платформе x86 по степени отказоустойчивости соответствуют классу mid-range — у них, например, может быть расширенный контроль целостности памяти и возможность «горячей» замены дисков. Есть даже линейки, по параметрам отказоустойчивости вплотную приближающиеся к классу hi-end, — например, HPE Superdome X, Fujitsu PrimeQuest, Huawei KunLun. Однако большая часть продаваемых сегодня серверов на платформе x86 относятся к кате-

[1] Позже линейка была переименована в PC Server, затем в Netfinity, потом в eServer xSeries и, наконец, в System x. В 2014 году бизнес по производству серверов на платформе x86 вместе с брендом System x был продан компании Lenovo.

[2] Ежегодные исследования под общим заголовком «Market Trends: Transformation of Non-x86 Servers».

гории low-end, или, как её ещё называют, commodity hardware, «обычное железо». Компоненты, используемые при сборке таких серверов, близки по характеристикам к компонентам настольных компьютеров, а в сервере дублируются только вентиляторы, диски и блоки питания. С использованием таких серверов практика администрирования инфраструктуры сместилась от парадигмы «питомцев» (pets), когда каждый сервер поддерживается в рабочем состоянии, к парадигме «стада» (cattle), когда отказавший сервер просто отключается, а его место занимает такой же типовой сервер из резервного пула. Огромный толчок в развитии получили программные средства обеспечения отказоустойчивости — как на уровне гипервизора (перезапуск виртуального сервера на другом хосте), так и на уровне СУБД (репликация данных между экземплярами и автоматизация переключения нагрузки).

Сегодня практически все СУБД, включая традиционные коммерческие реляционные платформы (Oracle, Db2, Microsoft SQL Server), имеют версию, работающую на аппаратной платформе x86 под управлением ОС семейства Linux. Для большинства современных коммерческих платформ (SAP HANA, SingleStore, Vertica) Linux является «родной» системой, а для проектов с открытым исходным кодом, включая те, что работают в виртуальной машине Java, версия под Linux — либо единственная, либо приоритетная.

Таким образом, сегодня ОС семейства Linux на платформе x86 является практически безальтернативным выбором для платформы обработки или хранения данных.

Важное решение, которое должен принять администратор, — размещать экземпляр СУБД в виртуальной машине или использовать физический («железный») сервер (bare metal). Единственный технический недостаток виртуального сервера — накладные расходы на виртуализацию, главным образом замедление ввода-вывода. Кроме того, долгое время переход корпоративных ИТ-подразделений на полностью виртуальную инфраструктуру задерживался из-за лицензионных войн Oracle и VMware. Достоинства виртуальной инфраструктуры — возможность выбора мощности сервера под задачу, скорость восстановления при сбоях и др. — значительно перевешивают её недостатки, и поэтому практически наверняка сервер, на котором будет работать ваша СУБД, будет виртуальным[1].

Единственный сценарий, в котором целесообразно использование физических серверов, — большие кластеры БД, работающие в многопользовательском

[1] Если только приложение не требует большого количества процессорных ядер под базу данных.

(multitenant) режиме и разграничивающие ресурсы разных клиентов средствами самой платформы. Примером такого кластера могут служить Hadoop или программно-аппаратный комплекс Oracle Exadata. Есть основания полагать, что облачные платформы, предоставляемые клиентам по модели PaaS, — например, Google Spanner, Amazon DynamoDB, YDB — также работают на физических серверах, но, к сожалению, описание инфраструктуры, лежащей под этими платформами, недоступно.

Что дальше?

Наконец, законный вопрос: если в каждом историческом периоде для сервера баз данных существует «безальтернативная» аппаратная платформа, то что придёт на смену текущему лидеру — Linux/x86?

Сегодня как ведущие технологические компании (Intel, IBM, Samsung, SUSE), так и сотни тысяч независимых разработчиков по всему миру прикладывают огромные усилия к развитию операционных систем семейства Linux, и нет никаких предпосылок к тому, что на смену Linux придёт какая-то другая операционная система. А вот смена доминирующей аппаратной платформы весьма вероятна.

В качестве возможных конкурентов семейству x86 имеет смысл присмотреться к платформам ARM и RISC-V.

Обе архитектуры являются «чистыми» RISC, в отличие от x86, в котором начиная с Pentium Pro реализован механизм трансляции сложного набора команд (CISC) в команды RISC-ядра. Самое же главное, что, в отличие от x86, ни у одной компании нет монополии на производство этих процессоров. Для производства процессоров ARM нужно купить лицензию, а спецификация RISC-V вообще находится в открытом доступе.

На текущий момент известно не так много серверов с ARM-процессорами — например, семейство Huawei TaiShan с процессорами Kunpeng, линейка Ampere Altra и серверы Amazon на процессоре AWS Graviton. Производители пока очень осторожно относятся к эксплуатации СУБД на этих серверах. Однако, как показывает история, рано или поздно «безальтернативному» решению будет найдена альтернатива — возможно, она уже найдена за время, прошедшее между написанием и публикацией книги.

Ещё одна важная тенденция — контейнерная виртуализация, то есть замена изоляции операционных систем средствами аппаратной виртуализации на изоляцию приложений в рамках одного экземпляра операционной системы за счёт

механизмов самой ОС. Первой реализацией такого подхода можно считать технологию «тюремной камеры» (jail) в ОС FreeBSD 4.0, а наиболее успешной коммерческой реализацией — зоны (zones) в ОС Solaris. Сегодня наиболее популярной платформой контейнерной виртуализации является Docker.

Безусловное преимущество контейнерной виртуализации перед аппаратной — существенная экономия ресурсов за счёт отсутствия гипервизора и единственного экземпляра операционной системы на сервере. Главный недостаток заключается не столько в самой технологии контейнерной виртуализации, сколько в особенностях систем оркестрации контейнеров, наиболее популярной из которых является Kubernetes.

Оркестрация устроена так, что контейнер в любое время может быть остановлен и перезапущен на другом сервере. Для серверов бизнес-логики, не содержащих состояния (stateless), перемещение проходит незаметно, поэтому значительная часть приложений уже переехала в контейнеры. Для серверов баз данных важно, чтобы при любом переезде клиенты могли их найти, системы мониторинга и резервного копирования продолжали получать данные, а дисковые тома были одинаково доступны. Для автоматизации развёртывания баз данных в контейнеризированной инфраструктуре появился относительно новый класс программного обеспечения — операторы (Kubernetes operator), например Stolon, Crunchy Data, Zalando, StackGres.

Ещё один существенный недостаток контейнерной виртуализации — отсутствие механизма управления выделением памяти и процессорных ядер. Если гипервизор выделяет для виртуальной машины фиксированные ядра и оперативную память, физически ближайшую к выделенным ядрам, то контейнеризованное приложение может перемещаться между ядрами, замедляя доступ к памяти.

На момент написания книги уровень зрелости платформ контейнерной виртуализации и операторов позволяет успешно эксплуатировать небольшие базы данных, требующие одного-двух процессорных ядер, но пока недостаточен для промышленной эксплуатации более крупных экземпляров.

12.2. Системы хранения данных

Дисковый массив или локальные диски?

Вопрос, вынесенный в заголовок, — первый из тех, на который следует ответить инженеру, проектирующему инфраструктуру хранения данных. Термину

«локальные диски» соответствует английский термин «direct-attached storage», или DAS, то есть диски, непосредственно подключенные к адаптеру шины без промежуточного сетевого устройства.

Преимуществами локальных дисков считаются скорость доступа (за счёт отсутствия посредников между накопителем и сервером) и низкая стоимость. При использовании современного оборудования оба аргумента оказываются несостоятельными.

Из таблицы ниже видно, что доступные локальные интерфейсы (SAS и SATA) уступают современным сетевым стандартам в пропускной способности:

Интерфейс	Скорость передачи данных
SATA (Serial Advanced Technology Attachment)	6 Гбит/с
SAS (Serial Attached SCSI)	до 24 Гбит/с
NVMe (Non-Volatile Memory) over PCI-e	63 Гбит/с (PCI-e 4.0, 4 линии); 252 Гбит/с (PCI-e 5.0, 8 линий)
FC (Fibre Channel[1])	28 Гбит/с (Gen6)
Ethernet	10 Гбит/с (802.3-2018); 40 Гбит/с и 100 Гбит/с (IEEE 802.3ba-2010)

Что касается стоимости, она зависит главным образом от маркетинговой политики производителя серверов и систем хранения данных, и вполне возможна ситуация, когда диск в составе дискового массива дешевле, чем точно такой же диск, установленный в сервер.

Преимущества выделенной системы хранения данных перевешивают все её настоящие и мнимые недостатки: выделенная СХД позволяет распределять дисковое пространство между серверами в точном соответствии с потребностями, а также обслуживать (расширять, модернизировать) дисковую инфраструктуру независимо от серверной.

Практически во всех случаях выделенная дисковая инфраструктура лучше, чем диски, встроенные в сервер. Но есть и несколько исключений:

- база данных в небольшой компании, где инфраструктура недостаточно развита для развёртывания выделенного слоя хранения данных;

[1] Чтобы избежать путаницы со спецификацией оптоволокна, стек протоколов для СХД называется на британском английском «Fibre channel» — в отличие от американского «Fiber channel». Тем более что спецификация включает и передачу данных по медным кабелям.

- база данных, для которой небольшое повышение производительности критично, но при этом сохранность данных не важна, — например, сервер для расчёта витрин по данным из хранилища под управлением таких платформ, как ClickHouse или Neo4j; или сервер для выдачи однотипных параметризованных отчётов на базе агрегатов из хранилища под управлением монолитной реляционной СУБД — PostgreSQL или MySQL;
- большой кластер из физических серверов под управлением кластерной СУБД (MongoDB, Cassandra, ScyllaDB), где масштабирование и отказоустойчивость обеспечиваются добавлением в кластер новых серверов: так, документация по MongoDB прямо не рекомендует использовать внешние СХД.

Протоколы сетевого доступа к дискам

Не раз и не два автору приходилось видеть серьёзные рассуждения на тему «NAS или SAN?». Давайте разберёмся, что это и какое отношение эти аббревиатуры имеют к базам данных.

Термин NAS (network-attached storage) означает систему хранения данных, подключённую к общей сети. Как правило, такая система предоставляет доступ по протоколам NFS (network file system) и SMB (Server Message Block)[1]. Это протоколы прикладного уровня, позволяющие работать с файлами. Они базируются на транспортных протоколах (TCP, UDP), не зависят от сетевого оборудования, но и не могут использовать преимущества конкретного оборудования. Последнее время набирают популярность хранилища с интерфейсом S3.

NAS, подключенный к быстрой сети предприятия, может обеспечить высокую пропускную способность (throughput), но задержка при обработке запросов (latency) будет чувствительна — за счёт глубокого стека протоколов. NAS прекрасно подходит для хранения резервных копий данных, но в качестве системы хранения для транзакционной СУБД это не лучший вариант.

Есть два сценария, в которых файловые или объектные хранилища можно использовать в качестве основного для базы данных:

- аналитическая платформа, спроектированная для массированного извлечения большого объёма данных и обработки их в памяти, — например, Vertica или Amazon RedShift, использующие хранилища с интерфейсом S3;

[1] Ранние версии протокола известны под названием CIFS (Common Internet File System).

- унаследованная со стародавних времён файловая база данных (например, Microsoft Access®), к которой необходимо обеспечить многопользовательский доступ.

Термином SAN (storage area network) обозначают специальный сегмент сети, предназначенный для подключения серверов к системам хранения данных. В инфраструктуре крупных предприятий этот сегмент чаще всего построен на стеке технологий FC. Этот стек включает в себя как спецификацию оборудования, так и протокол блочного устройства, поверх которого операционная система должна самостоятельно сформировать файловую систему[1].

Сигнальный протокол стека FC обеспечивает, в отличие от ранних спецификаций Ethernet, гарантированную упорядоченную доставку кадров, что критически важно для обеспечения быстрого отклика систем хранения данных. Однако с развитием стека Ethernet[2] появилась возможность построения SAN на его базе, что существенно дешевле, чем построение инфраструктуры FC. Единственная рекомендация, остающаяся неизменной, — физически отделять сетевые сегменты общего назначения (LAN, local area network) от сегмента СХД (SAN).

Специалисты много спорят о том, какой из блочных протоколов обеспечивает наилучшее соотношение стоимости, производительности и надёжности — FCP, NVMe over Ethernet, iSCSI... Обсуждение архитектуры систем хранения данных выходит за рамки данной книги. Ограничимся замечанием, что подавляющее большинство высоконагруженных баз данных располагаются на СХД, представляющих свои ресурсы серверам по блочным протоколам через специально выделенный сетевой сегмент.

Hi-end или mid-range?

Как и серверное оборудование, системы хранения данных до недавнего времени делились на классы: «корпоративный» (enterprise grade, hi-end) и «средний» (mid-range). Оборудование разных классов отличалось по таким параметрам, как надёжность, производительность и набор сервисных функций (зеркалирование томов, сжатие, автоматический тиринг[3] и т. д.). Однако развитие технологий

[1] Некоторые СУБД, например Oracle и Db2, способны работать с блочным устройством напрямую, без файловой системы.

[2] В частности, с появлением протокола RDMA (Remote Direct Memory Access).

[3] От английского tier — уровень. Тиринг — перенос данных, доступ к которым происходит редко, на более дешёвые и медленные диски.

и стандартов привело к тому, что сегодня граница между классами СХД практически стёрлась.

Появление интерфейса SAS, пропускная способность которого больше, чем скорость чтения с диска, привело к исчезновению дисков с интерфейсом FC — отныне этот интерфейс остался только между сервером и дисковым массивом. Стоимость дисков, используемых в дорогих и дешёвых массивах, сравнялась.

С массовым внедрением накопителей на основе flash-памяти оказалось, что тип накопителя влияет на производительность СХД сильнее, чем всё остальное. Одно время сложилась парадоксальная ситуация, когда массивы, формально относящиеся к среднему классу, но наполненные исключительно твердотельными накопителями (AFA, all-flash array), по производительности существенно превосходили любые hi-end системы с традиционными жёсткими дисками.

Резкое удешевление и рост производительности процессоров общего назначения, в особенности x86, свели на нет и другие преимущества hi-end оборудования: использование таких процессоров с урезанным ядром Linux позволило существенно ускорить разработку программного обеспечения для дисковых массивов. Сервисные функции, бывшие визитной карточкой дорогого оборудования, появились и в более дешёвых моделях, а использование отработанных технологий кластеризации позволило довести надёжность модульных систем среднего класса до высочайших показателей.

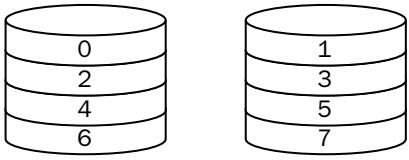
Таким образом, сегодня выбор системы хранения для базы данных сводится к выбору типа накопителей — твердотельные или традиционные жёсткие диски. Учитывая незначительную разницу в стоимости, ответ для транзакционной системы практически всегда будет однозначным — СХД на основе flash-памяти. Традиционные диски сохраняют свои позиции благодаря приложениям, хранящим огромные объёмы данных, на которых экономия может быть существенной. Это прежде всего массивы неструктурированных данных, предназначенных для обработки и загрузки в хранилище (data lake), а также архивы систем, выведенных из эксплуатации.

Отказоустойчивость дисковых массивов

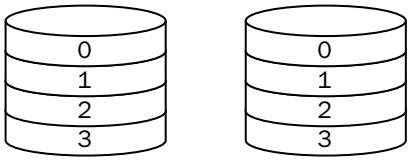
Наконец, последний вопрос: как обеспечить отказоустойчивое хранение данных на дисках? Традиционно для этого несколько дисков объединяются в один виртуальный том (volume). Такая конструкция называется RAID, Redundant Array of Inexpensive Disks.

Базовых конфигураций (уровней) RAID всего четыре:

- RAID0 (stripe) — объединение дисков в один том без дублирования. Объём виртуального тома равен сумме объёмов всех входящих в него дисков, производительность существенно возрастает за счёт «размазывания» операций по разным дискам. При отказе хотя бы одного диска теряются данные всего тома. Конфигурация RAID0 может использоваться только для тестовых сред или для достижения экстремальной производительности обработки данных, которые можно восстановить из надёжного источника.

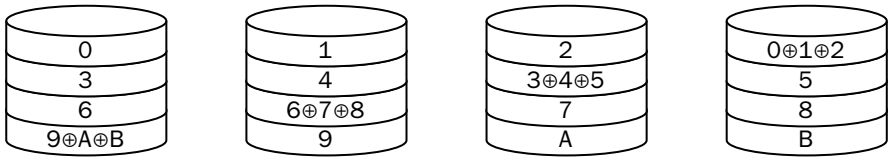


- RAID1 (mirror) — зеркалирование дисков, то есть полное дублирование информации, записанной на каждый диск, на запасном диске или нескольких запасных дисках. Объём виртуального тома равен объёму одного физического диска (то есть $1/N$ суммарного объёма всех дисков), производительность записи равна производительности записи на один диск, производительность чтения может быть выше, чем у одного диска, за счёт параллельного чтения с нескольких дисков. Данные виртуального тома теряются только при одновременном выходе из строя всех дисков.

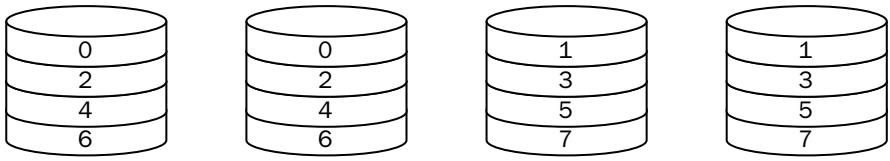


- RAID5 — набор дисков с контрольной суммой. Похож на RAID0, где блоки данных распределены между дисками, но один из блоков хранит сумму данных остальных блоков, рассчитанную с помощью «исключающего или» (xor). Потеряв диск, можно восстановить все его блоки, просуммировав блоки с оставшихся дисков. Таким образом, к потере тома приводит потеря двух дисков из группы. Скорость чтения с виртуального тома RAID5 и скорость поточной записи такие же высокие, как и у RAID0. А вот произвольная запись — медленная, т. к. требует чтения всех блоков, участвующих в контрольной сумме, и записи двух блоков — непосредственно изменяемого и блока с контрольной суммой. Количество дисков в группе RAID5

может быть любым, объём виртуального тома равен объёму $N-1$ физических дисков.



- RAID10 (RAID1+0) объединение дисков (RAID0), где каждый из объединяемых дисков является «зеркалом» (RAID1). Соединяет в себе преимущества RAID0 и RAID1: высокую скорость чтения и записи и высокую надёжность. Для потери тома необходим отказ обоих дисков в одном из зеркал. Объём виртуального диска, как и в RAID1, равен $1/N$ (как правило, $1/2$) объёма входящих в него физических дисков.



В современных системах используются уровни RAID5 и RAID10. RAID5 хорош для аналитических систем с поточной записью, нагруженных преимущественно чтением данных. Для систем с интенсивной записью следует предпочесть RAID10, даже если диск собирается из твердотельных накопителей. Если вы строите аналитическую систему, которая выполняет запросы над гигантскими объёмами данных, не помещающимися а оперативную память, то тома для данных можно собрать в RAID5, а вот тома для временных областей следует собирать в RAID10.

В современных дисковых массивах могут находиться неиспользуемые диски, зарезервированные для горячей замены (spare disk). В случае отказа диска перераспределение данных (data rebalancing) на запасные диски запускается немедленно.

Определённую популярность завоевала файловая система ZFS, которая, в отличие от традиционных файловых систем, строится поверх физических дисков (raw disk) и включает в себя менеджер томов и реализацию RAID-Z, являющуюся усовершенствованной модификацией RAID5.

Важно также знать, что некоторые платформы (например, Apache Cassandra, Hadoop или Oracle с опцией ASM, Automatic Storage Management) рассчитаны

на работу с ненадёжными дисками. Эти платформы на программном уровне дублируют данные на несколько дисков и восстанавливают степень дублирования при отказе диска. Таким платформам может быть выгоднее предоставить дисковое пространство в режиме JBOD (Just a Bunch Of Disks), в котором каждый виртуальный том соответствует одному физическому диску.

Литература

- Рейтинг и сравнение серверных процессоров Intel Xeon.
  smb.marvel.ru/blog/komplektuyushchie-i-optsii/reyting-servernykh-protsessorov-intel-xeon/
- Георгий Шуклин. Как правильно мерять производительность диска.
  habr.com/ru/post/154235/
- Origin of the word “mainframe”?
  english.stackexchange.com/questions/28290/origin-of-the-word-mainframe
- Al Grega. A Brief History of the Mainframe World.
  community.ibm.com/community/user/ibmz-and-linuxone/blogs/destination-z1/2019/12/23/a-brief-history-of-the-mainframe-world
- IPSJ Computer Museum.
  museum.ipsj.or.jp/en/index.html
- A UNIX History Diagram.
  www.levenez.com/unix/
- Gavin McCance. CERN Data Centre Evolution.
  www.slideshare.net/gmccance/cern-data-centre-evolution
- Николай Богданов. Обзор операторов PostgreSQL для Kubernetes. Часть 1: наш выбор и опыт.
  habr.com/ru/company/flant/blog/520616/
- Введение в системы хранения данных.
  habr.com/ru/company/dell_technologies/blog/125828/

Глава 13
Коммерческие вопросы эксплуатации

> — Это ж надо, — ни к кому не обращаясь, громко сказала она. — Это ж надо! Так прямо и написано — главной задачей общества является извлечение прибыли в интересах акционеров. Ну вы подумайте! Ничего не боятся!
>
> *Юлий Дубов, «Меньшее зло»*

13.1. Надёжность и производительность

Тестирование производительности баз данных

Вопрос, на который всем хотелось бы знать ответ, — «какая платформа самая быстрая?».

Первый ответ на этот вопрос попыталась дать компания IBM в 1970 году, предложив тест TP1, обрабатывавший банковские транзакции в пакетном режиме. Производители СУБД один за другим отчитывались о результатах тестирования, достигавших десятков тысяч транзакций в секунду (tps, transactions per second), но реальные пользователи получали при использовании этих платформ сотни транзакций в секунду. Такая разница была связана с тем, что тест не учитывал ни характеристики сети, ни другие факторы, влияющие на время обработки транзакции.

Для наведения порядка в отрасли и прекращения «войны бенчмарков» в 1982 году доцент Висконсинского университета Дэвид Девитт (David DeWitt) создал более детерминированный эталонный тест. Однако СУБД Oracle показывала в этом тесте весьма скромные результаты, и главным итогом появления этого теста стала так называемая «оговорка Девитта», запрещающая публикацию результатов каких-либо тестов производительности без согласования с производителем СУБД. Такой запрет по сей день содержится, например, в лицензионных соглашениях Oracle и Microsoft.

В 1985 году компания Tandem Computers попыталась вернуть к жизни тест TP1, добавив в него новые ограничения, — в частности уточнялась методика расчёта стоимости системы, устанавливались правила пропорционального масштабирования количества пользователей и объёма данных, вводилось ограничение на время обработки каждой отдельно взятой транзакции.

Заставить всех производителей СУБД следовать методике Tandem было практически невозможно, поэтому в 1988 году была создана некоммерческая организация TPC (Transaction Processing Performance Council), которой давалось право публикации результатов тестирования баз данных и обязанность проверять все публикуемые результаты. В организацию вошли как разработчики СУБД (Oracle, IBM, Informix, Sybase, Software AG и др.), так и производители оборудования (DEC, Unisys, Hitachi...).

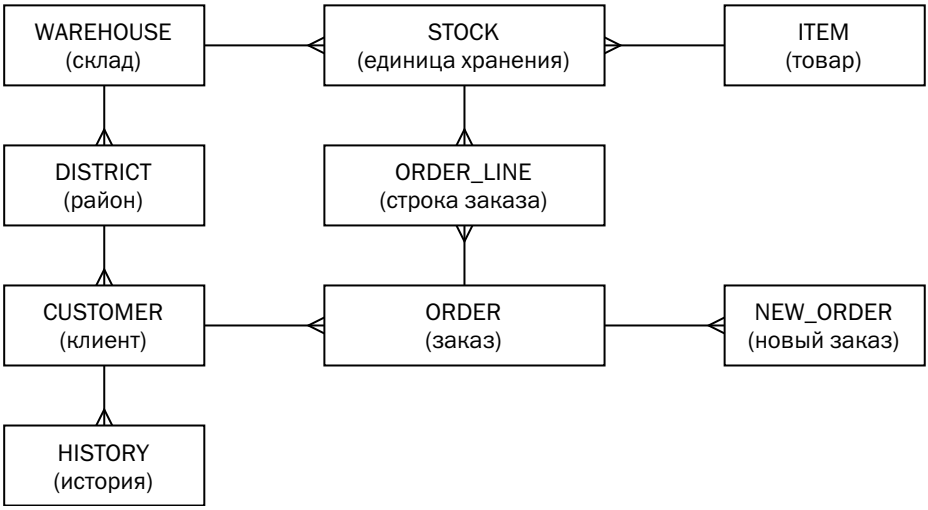
На первом собрании консорциума были утверждены спецификации тестов TPC-A и TPC-B. Если первый требовал эмуляции всей сопутствующей инфраструктуры — терминалов и сети — и учитывал время реакции пользователей, то второй выполнялся в пакетном режиме. До сих пор утилита pgbench, входящая в состав PostgreSQL, для оценки производительности выполняет именно тест TPC-B.

Первый результат теста TPC-A был опубликован в 1990 году, а уже в 1994 году результаты теста были улучшены на два порядка при снижении стоимости транзакции на порядок. Такой рост говорил не столько о прогрессе вычислительной техники, сколько об умении производителей приспосабливаться к условиям теста. Поэтому в 1995 году оба теста были признаны несостоятельными, и им на смену пришёл TPC-C.

В тесте TPC-C была изменена предметная область (оптовый склад вместо банка), увеличено количество таблиц (9 вместо 4), добавлены операции чтения (примерно 8 % от общего количества операций), изменены правила масштабирования нагрузки. Модель данных для теста TPC-C приведена на рисунке ниже.

За годы существования теста было опубликовано огромное количество результатов, полученных на самых разных конфигурациях оборудования. Каждый, кто желает провести такой тест на своей БД, может воспользоваться свободно распространяемой утилитой HammerDB.

С накоплением результатов стало ясно, что производительность базы данных в тесте не связана напрямую с производительностью всей системы под реальной нагрузкой, и появились даже специальные коэффициенты пересчёта результатов теста, выраженных в «условных единицах» tpmC, в «реальные транзакции» в зависимости от особенностей приложений.



Одновременно стало понятно, что способность платформы исполнять транзакции ничего не говорит о её производительности при выполнении аналитических запросов. Для измерения производительности под OLAP-нагрузкой был разработан тест TPC-D, оказавшийся неудачным, и в 1999 году ему на смену пришёл TPC-H. В этом тесте параллельно с выполнением запросов запущены ETL-процессы, обновляющие данные. Кроме того, при публикации результата обязательно указывается объём базы данных, на которой результат получен, то есть соревнования идут в нескольких «весовых категориях».

В 2006 году консорциум опубликовал новый тест для транзакционных систем — TPC-E. В нём схема базы данных стала существенно сложнее (33 таблицы вместо 9), добавились декларативные ограничения целостности, существенно изменился профиль нагрузки — отношение чтения к записи составляет 77:23, что гораздо ближе к профилю нагрузки промышленных баз данных. Однако за всё время существования теста на сайте было опубликовано менее сотни результатов — время интереса к синтетическим тестам прошло.

Тест	Тип	Кол-во таблиц	Кол-во ограничений	Чтение : запись	Предметная область
TPC-A/B	Транзакционный	4	0	0 : 100	Банк
TPC-C	Транзакционный	9	0	8 : 92	Оптовый склад
TPC-E	Транзакционный	33	22	76,9 : 23,1	Брокерский дом
TPC-H	Аналитический	8	17	92 : 8	Розничная торговля

Сегодня очевидно, что производительность системы в целом зависит не только от производительности СУБД, но в первую очередь от решаемой задачи и даже от конкретного прикладного программного обеспечения, решающего задачу. Поэтому актуальность сохранили только специфические тесты, показывающие производительность вычислительных комплексов под управлением тиражируемых приложений (standard application benchmark), распространяемые, например, компаниями SAP, Oracle, Microsoft, 1C. Во всех остальных случаях имеет смысл верить лишь собственным нагрузочным тестам, где тестируется именно то приложение, которое и будет работать с базой данных.

Влияние оборудования на производительность БД

Мощность сервера, используемого для обслуживания базы данных, в большинстве случаев исчерпывающе определяется количеством процессорных ядер и объёмом оперативной памяти.

Опыт показывает, что для дисковых реляционных платформ (Oracle, PostgreSQL, Microsoft SQL Server) оптимальное соотношение объёма памяти в гигабайтах к количеству ядер составляет 8, то есть, например, четырёхъядерный сервер должен иметь 32 ГБ памяти. Под ядром здесь понимается именно вычислительное ядро, возможно, снабжённое несколькими наборами регистров[1], способное выполнять несколько потоков одновременно и опознаваемое операционной системой как несколько «логических ядер». Такое же соотношение предлагает для серверов баз данных и самый популярный облачный сервис Amazon EC2[2]. Оптимальное соотношение для отдельных задач может отличаться, но это определяется только нагрузочным тестированием.

Поскольку значительную часть времени внутри СУБД процессор проводит в ожиданиях ввода-вывода и блокировках, тактовая частота процессора влияет на производительность незначительно. Повышение частоты на четверть (например, с 2,4 ГГц до 3 ГГц) может вообще не показать прироста производительности, но при этом существенно увеличит стоимость сервера.

---

[1] Технология, позволяющая на одном физическом ядре выполнять одновременно несколько потоков, называется «Hyperthreading» в процессорах Intel x86 и «Multithreading» в процессорах IBM POWER и Oracle SPARC, причём Oracle различает «Vertical Multithreading» (в SPARC64 VI) и «Simultaneous Multithreading» (SPARC64 VII и более поздние).

[2] EC2 предлагает 4 ГБ на каждый «виртуальный процессор» (vCPU), который представляет собой «логическое ядро», то есть один поток двухпоточного ядра Intel x86.

Гораздо важнее, чем тактовая частота, объём кеш-памяти третьего уровня, в особенности для СУБД в памяти, которые способны работать быстрее, если служебные области менеджера памяти поместятся в кеш. Если вы арендуете сервер, то в нём уже будет серверный процессор, а вот если самостоятельно собираете сервер начального уровня для небольшой компании, то более правильным решением будет купить специализированный серверный процессор с большим объёмом кеш-памяти, а не процессор для игр, пусть даже из верхнего ценового сегмента. Так, например, собирая сервер для базы данных, стоит предпочесть Xeon даже начальной серии E «топовому» Core i9.

Есть, разумеется, и отдельные случаи, когда частота процессора играет существенную роль. Это важно для платформ, которые выполняют все изменения данных в одном потоке (VoltDB, Tarantool), а также для баз данных, обслуживающих приложения с длительными однопоточными операциями (например, 1С). В этом случае стоит выбирать серверный процессор с большей тактовой частотой и, если есть ограничения по стоимости, с меньшим количеством ядер.

При выборе системы хранения данных важны количество операций ввода-вывода в секунду (IOPS, Input/Output Per Second) и пропускная способность (throughput, или bandwidth). Если свести выбор к простой рекомендации, то для баз данных, обрабатывающих транзакции, следует обращать внимание на максимальное количество операций, а для аналитических систем — на пропускную способность. На самом деле тема измерения производительности СХД весьма обширна и выходит далеко за рамки книги, поэтому в заключение раздела — просто несколько тезисов:

- Строго говоря, пропускная способность дисковой подсистемы жёстко связана с количеством операций и равна произведению количества операций в секунду на размер блока. Отсюда очевидно, что тестирование должно проводиться на том же размере блока, с которым работает база данных.
- Одна и та же СУБД в зависимости от настроек экземпляра и профиля данных может работать с блоками данных разного размера.
- СХД может обеспечивать разное количество операций с блоками разного размера. Есть несколько оптимальных размеров, на которых СХД даёт наилучшую производительность, причём для разных дисков и для разных конфигураций СХД эти размеры могут отличаться.
- Количество операций существенно зависит от близости друг к другу обрабатываемых данных. Последовательные операции могут выполняться на порядки быстрее, чем операции над случайно расположенными данными.

Для транзакционных систем характерен случайный доступ, а для аналитических — последовательный.

- В некоторых результатах тестов заявляется какое-то количество «операций ввода-вывода» без раскрытия соотношения операций чтения и записи в профиле нагрузки. На самом деле операции записи могут выполняться в разы медленнее, чем операции чтения.
- Не существует однозначного соответствия между операцией над данными и операциями ввода-вывода. Так, например, поиск строки по значению первичного ключа в реляционной СУБД может потребовать от 0 до 7 операций чтения. Тем более не существует однозначного соответствия между бизнес-операцией и набором операций в базе данных.
- Производительность базы данных зависит не только от производительности системы хранения данных, но и от настроек кеширования в экземпляре БД и в операционной системе. Кроме того, производительность существенно зависит от профиля данных.

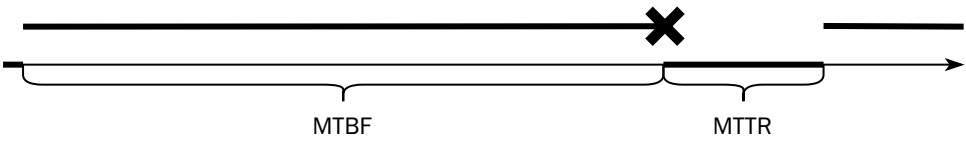
Таким образом, результаты синтетических тестов относительно пригодны для сравнения разных систем хранения между собой. На вопрос достаточности того или иного хранилища для работы с вашей базой данных даст ответ только нагрузочное тестирование именно вашей базы.

Обеспечение надёжности баз данных

Важнейшим интегральным показателем любого сервиса, в том числе базы данных, является доступность, то есть отношение времени, когда сервис доступен, к общему времени эксплуатации сервиса. Доступность современных информационных систем обычно измеряется в количестве девяток, необходимых для записи интересующего нас отношения. В таблице ниже приведено соответствующее каждому количеству девяток время годового простоя:

Количество девяток	Доступность, %	Годовой простой
2	99	88 ч
3	99,9	8 ч 45 мин
4	99,99	52 мин 30 с
5	99,999	5 мин 15 с

В документации на оборудование производитель обычно указывает среднее время наработки на отказ (MTBF, Mean Time Between Failures) и среднее время ремонта (MTTR, Mean Time To Repair). Доступность сервера вычисляется как $^{MTBF}/_{MTBF + MTTR}$.



Время наработки сервера на отказ — расчётная величина, при определении которой используется статистика по отказам отдельных компонентов сервера. Время доступности сервера определяется как время доступности всех его компонентов. Для ненадёжных компонентов применяется дублирование по одной из схем:

- основной/резервный (master/standby), когда работает один компонент, а второй вступает в работу только при отказе первого;
- балансировка нагрузки (load balance), когда для функционирования системы необходимо M компонентов, но в систему включено $N > M$ компонентов и до отказа $N - M + 1$ компонентов система сохраняет работоспособность.

В современных серверах начального и среднего уровней дублируются только вентиляторы, блоки питания и диски, а в многопроцессорных серверах высшей категории надёжности (hi-end) дублируется всё, включая центральный процессор: отказ одного процессора не приводит к полному отказу сервера, а иногда не приводит даже к остановке исполняемых приложений.

Время, затраченное на ремонт сервера, складывается из времени доставки этого компонента до вычислительного центра, где установлен сервер, и времени замены вышедшего из строя компонента. Если собственно время замены зависит только от конструктивных особенностей модели и одинаково для всех обладателей серверов этой модели, то время доставки зависит от уровня качества поддержки, предусмотренного контрактом между поставщиком оборудования и покупателем (SLA, Service Level Agreement). Заявляемая производителями доступность сервера средней ценовой категории с высшей категорией сервиса, включающей хранение запасных частей на территории заказчика, может достигать пяти с половиной девяток, то есть 99,9995 %.

Важно понимать, что высокая «в среднем» доступность оборудования не означает автоматически ни высокой доступности конкретного сервера, ни тем более

высокой доступности данных. Время загрузки операционной системы и запуска экземпляра базы данных существенно превосходит время собственно перезагрузки сервера, а по сравнению со временем ремонта оборудования с выездом сервисного инженера даже затраты на старт экземпляра базы данных находятся на уровне погрешности измерения. Поэтому для обеспечения доступности данных применяют кластеризацию.

Кластерное решение высокой доступности (HA-cluster) состоит из средств обнаружения и локализации сбоя, а также средств автоматизации восстановления, то есть переключения нагрузки на резервный экземпляр.

Кластеризация решает задачи отказоустойчивости (HA, High Availability) и катастрофоустойчивости (DR, Disaster Recovery). В общем случае это не одно и то же: задача отказоустойчивого решения — реакция на отказ отдельных компонентов, а катастрофоустойчивое решение должно восстановиться после катастрофы — недоступности всего вычислительного центра или потери сетевой связности.

Отказоустойчивость базы данных может быть обеспечена внешними средствами. Например, экземпляр базы данных может работать на виртуальной машине, запущенной на одном из узлов фермы виртуализации. Надёжность хранения данных обеспечивается дисковым массивом, к которому подключены все узлы, а при сбое виртуальной машины или всего узла машина просто будет перезапущена менеджером виртуальной инфраструктуры на другом узле.

Катастрофоустойчивое решение должно включать в себя механизм репликации — копирования данных из основной базы в резервную. Кластерное решение может управлять этим механизмом — запускать, останавливать, изменять направление репликации и т. д. Кроме того, для устойчивости к сетевому разделению (split-brain) желательно иметь в катастрофоустойчивом решении дополнительный элемент — сервер или просто сетевой диск, который предотвращает появление в системе двух ведущих узлов. Такой элемент называют кворумным устройством (quorum device), или арбитром (referee). Прежде чем объявить себя ведущим, узел должен заручиться поддержкой кворумного устройства, а поддержка двух ведущих узлов одновременно невозможна.

Несмотря на то что высокая доступность (отказоустойчивость) и катастрофоустойчивость — разные задачи, для решения этих задач может быть применено одно и то же средство. Классический пример такого средства — кластер из двух баз данных[1] в вычислительных центрах, находящихся недалеко друг от друга, как

[1] См. раздел 8.2 «Отказоустойчивый кластер с репликацией».

правило в крупном городе (metropolitan area). Центры строятся так, чтобы длина оптоволоконного кабеля между ними не превышала 50 километров, что примерно соответствует расстоянию до 30 километров по прямой. Базы данных связаны синхронной репликацией. Кроме того, где-то есть ещё один небольшой вычислительный центр, где находится кворумное устройство.

Важнейшими показателями кластера высокой доступности являются целевое время переключения (RTO, Recovery Time Objective) и максимальный объём потерянных при переключении данных, или, другими словами, точка во времени, на которую будет восстановлено состояние базы данных (RPO, Recovery Point Objective).

Репликации была посвящена отдельная глава, а здесь хотелось бы только отметить, что не существует механизма, обеспечивающего одновременно как мгновенное переключение (RTO = 0), так и полную сохранность данных (RPO = 0). Всегда приходится выбирать что-то одно, и в критичных системах, обслуживающих клиентов, применяют двухуровневую защиту. Бэк-офисная система, то есть система, непосредственно хранящая остатки на счетах или складские запасы, допускает кратковременный простой (RTO > 0), но не допускает потерю данных (RPO = 0). Фронт-офисная система, то есть система, взаимодействующая с пользователем, допускает потерю данных — например, информацию о текущем содержимом «корзины» (RPO > 0), но не допускает простоя (RTO = 0).

В заключение заметим, что конечному пользователю важна даже не доступность базы данных, а доступность сервиса. Таким образом, база данных не является «последним рубежом обороны» — приложения могут иметь собственные механизмы обеспечения высокой доступности, работающие на уровень выше, чем база данных. Обсуждение этих механизмов выходит далеко за рамки книги, поэтому отметим лишь два из них:

- Шардирование, то есть деление системы на несколько частей, каждая со своей базой данных. Часть шардированной системы всегда доступна, поэтому доступность всей системы рассчитывается как отношение успешно обработанных запросов к общему числу запросов. При сбое одного шарда (в частности, при сбое его базы данных) клиенты остальных шардов продолжают получать полноценный сервис.

- Альтернативный путь предоставления услуги. Если приложение недоступно, то бизнес-процесс может предусматривать оказание услуги с отложенной регистрацией в приложении — например, «запасное» приложение, договорённости в электронной почте или мессенджерах или даже оформление сделки на бумаге.

13.2. Классификация информационных систем

Даже в рамках одной аппаратной платформы стоимость оборудования для развёртывания экземпляра СУБД может кратно отличаться. На стоимость влияет класс серверов (commodity или mid-range), класс системы хранения данных, степень резервирования, регламент резервного копирования и ряд других факторов.

С одной стороны, избыточные решения дороги. С другой — убытки от простоя критически важной для бизнеса системы могут оказаться гораздо больше, чем любые вложения в инфраструктуру. В этом разделе поговорим о том, как найти компромисс между стоимостью системы и её надёжностью.

Перед тем как начать — важное замечание. Ещё в середине 2010-х годов нормальной практикой была разработка приложений, рассчитанных на надёжную инфраструктуру. Большинство современных приложений рассчитаны на работу в облаке, где отказ инфраструктуры является рядовым событием. Значительная часть задач отказоустойчивости решается самим приложением на программном уровне. Здесь будем говорить не о подходах к отказоустойчивости, а только о классификации информационных систем, позволяющей для каждой системы выбрать нужный подход, будь то аренда облака с лучшими характеристиками доступности или усложнение кода самого приложения.

При достижении предприятием определённого размера отдел ИТ-сопровождения стремится к стандартизации инфраструктуры. Конфигурации систем, доставшихся сопровождению в наследство с доисторических времён, могут быть настоящими произведениями искусства, однако для новых баз данных инженеры сопровождения стремятся выдавать стандартные конфигурации. Например, в 2010-х годах набор инфраструктурных решений для реляционной СУБД мог выглядеть так:

- серверы класса hi-end с дисковыми массивами класса hi-end плюс синхронная репликация;
- серверы и дисковые массивы среднего класса плюс синхронная репликация;
- серверы и дисковые массивы среднего класса плюс асинхронная репликация;
- commodity-серверы с дисковыми массивами среднего класса.

Современные решения вместо классов серверов могут отличаться, например, классами виртуальных машин.

Обратите внимание, что в классификации оборудования не фигурирует его мощность. Нагрузка на систему не связана с требованиями к надёжности, хотя, как правило, высоконагруженные системы весьма важны для эксплуатирующего их предприятия.

Как же теперь выбрать конфигурацию для конкретной базы данных, принадлежащей конкретному приложению?

Можно составить список «самых важных приложений, которые должны работать во что бы то ни стало», но такой подход не гарантирует ни рационального расходования денег, ни достаточной защиты.

С одной стороны, конфигурация оборудования для приложений, не попавших в заветный список, зависит от политического веса владельца системы. В результате какой-нибудь сервис электронных больничных листов работает на самом дорогом оборудовании, потому что это любимое детище главного бухгалтера, с которым никому не хочется ссориться. Налицо неразумная трата денег.

С другой стороны, некоторые приложения могут не войти в список, потому что про них не подумали. Например, все помнят про процессинг банковских карт, но забывают про проверку клиентов по «чёрным спискам», которая должна сопровождать каждую операцию. В результате отказ системы проверки становится неприятной неожиданностью и приводит к серьёзным проблемам.

Существует формальная методика, позволяющая сделать правильный выбор и защитить то, что нуждается в защите, не переплатив за то, за что можно не переплачивать. Эта методика относит каждое приложение к одному из классов критичности.

Исторически сложилось, что этих классов четыре — как раз по количеству рассмотренных нами ранее конфигураций инфраструктуры. Называться классы могут, например, так (в порядке убывания критичности): Platinum, Gold, Silver, Bronze. Или так: Mission critical, Business critical, Business operational, Office productivity. Общепринятой русской классификации, к сожалению, не существует.

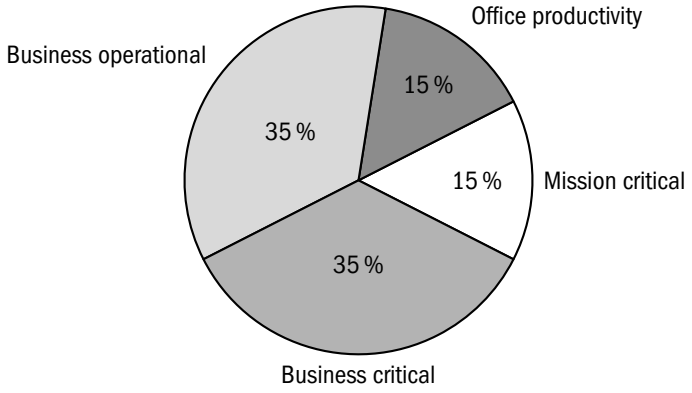
Для отнесения информационной системы к одному из классов ей выставляются формальные оценки. В процессе оценки важно соблюдать два правила:

- Систему оценивает её владелец, а не обслуживающее подразделение. Критичность определяется не тем, насколько трудоёмко обслуживание системы, а исключительно тем, какие убытки понесёт владелец от её простоя.
- Формулировки вопросов, из которых складывается оценка, должны предусматривать возможность верификации ответов. Разумеется, ответы всё равно основаны на экспертных мнениях, но эксперт по крайней мере может как-то своё мнение обосновать.

Что даёт системе присвоенный класс критичности?

- **Приоритет обслуживания** при массовых инцидентах. Безусловно, восстанавливать после аварии нужно всё, но если авария задела несколько систем, то в первую очередь нужно восстановить наиболее критичные.
- **Типовые значения SLA** (service level agreement). Если простой системы приносит убытки, то правильный путь — не жаловаться на администраторов, а повышать её уровень критичности.
- Выбор **инфраструктурного решения**. Каждое из стандартных решений обладает определёнными характеристиками надёжности, показывающими вероятность сбоев и скорость восстановления, а также определённой стоимостью.

В мировой практике сложилось примерно следующее распределение систем по классам:



Это не значит, что на любом предприятии распределение систем по классам должно быть именно таким. Но в любом случае если в класс Mission critical попало больше 15 % эксплуатируемых систем, это повод серьёзно задуматься.

На вопрос «насколько нужна та или иная система» любой владелец ответит «очень». Следовательно, нужно задавать другой вопрос: а что случится, если система остановится? Класс критичности системы зависит от тяжести последствий остановки системы и скорости их наступления. Рассмотрим возможные последствия на примере нескольких банковских систем.

Расчётная система обеспечивает (сюрприз!) расчёты между клиентами — юридическими лицами. Если вдруг крупный корпоративный клиент не сможет сделать платёж контрагенту, то банк потеряет весьма существенную сумму, поэтому расчётная система, без сомнения, попадёт в высший класс критичности.

Карточный процессинг обеспечивает авторизацию, то есть возможность расплачиваться картой. Если сотня-другая клиентов не смогут сделать покупки, потери банка будут невелики, но массовый отказ в обслуживании недопустим сам по себе, поскольку может заставить клиента задуматься о смене банка.

Если остановится система, обслуживающая банковские депозиты физических лиц, то убытки банка вновь будут невелики, а отказ в обслуживании не будет столь массовым, как в случае процессинга. Но нужна ли нам передовица в газете с заголовком «Банк отказывается выдавать вклады»? Вопрос риторический.

Наконец, если что-то случится с главной бухгалтерской книгой, то клиенты ничего не заметят, ведь эта система в обслуживании клиентов вообще не участвует. Но стоит задержать сдачу баланса, как санкции Центробанка не заставят себя ждать.

Итак, негативные последствия от простоя системы можно разделить на четыре типа:

- экономические (непосредственные убытки);
- клиентские (отказ в обслуживании);
- репутационные (негативные реакции в средствах массовой информации);
- юридические (от предупреждений и штрафов до судебных исков и отзыва лицензии).

Для каждого типа последствий следует сформулировать критерии тяжести и присвоить им оценки от 0 до 4. Например, таблица может выглядеть так:

	0	1	2	3	4
Экономические (от плановой прибыли)	нет	$\leq 0{,}1\,\%$	0,1–0,5 %	0,5–1 %	$> 1\,\%$
Клиентские (от числа клиентов)	нет	1	1–5 %	5–10 %	$> 10\,\%$

	0	1	2	3	4
Репутационные (огласка в СМИ)	нет	маловероятна	в локальных	в региональных	в федеральных
Юридические	нет	предупреждения регуляторов	штрафы регуляторов	гражданские иски	отзыв лицензии

Разумеется, все цифры условны, все методики подсчёта основываются исключительно на экспертной оценке, а простор для споров, что считать «региональными СМИ» и как относиться к негативным статьям в популярных блогах, поистине безграничен. Но в крупной корпорации наверняка найдётся и юридический отдел, и PR-служба, которые с готовностью выскажут компетентное мнение.

Следующим шагом нужно выбрать временны́е интервалы, на которых мы будем оценивать убытки. Например, час, 4 часа, 8 часов, 24 часа. Эти интервалы произвольны и не имеют никакого отношения к SLA, заключённым на оцениваемые системы. В дальнейшем, разумеется, было бы правильно привязать типовые SLA именно к этим интервалам.

В процессе оценки владелец каждой системы заполняет матрицу из 16 клеток, проставляя в каждой клетке оценку тяжести последствий, соответствующих строке, если система не будет функционировать в течение времени, соответствующего столбцу. Важно, что оценка последствий для более длинного интервала не может быть меньше, чем оценка последствий того же класса для более короткого интервала.

Ниже приведён пример заполнения матрицы:

	до 1 часа	1–4 часа	4–8 часов	8–24 часа
Экономические	1	1	3	3
Клиентские	1	2	2	3
Репутационные	0	0	1	2
Юридические	1	2	3	4

Так, в соответствии с этой матрицей простой системы в течение трёх часов приведёт к потере до 0,5 % прибыли, затронет до 5 % клиентов и грозит штрафами регуляторов, но не несёт репутационных рисков.

Чтобы из этой матрицы получить окончательную оценку, осталось выполнить три шага.

Шаг первый: для каждого временного интервала выбираем максимальную оценку:

	до 1 часа	1–4 часа	4–8 часов	8–24 часа
Максимум	1	2	3	4

Шаг второй: по таблице, приведённой ниже, транслируем полученные оценки в классы критичности. В отличие от оценок, приведённых выше для примера, эта таблица является неотъемлемой частью методики:

Баллы	до 1 часа	1–4 часа	4–8 часов	8–24 часа
4	МС	МС	ВС	ВС
3	МС	ВС	ВС	ВО
2	ВО	ВО	ВО	ОР
1	ВО	ВО	ОР	ОР

Для нашей гипотетической системы получаем следующие оценки:

	до 1 часа	1–4 часа	4–8 часов	8–24 часа
Класс	ВО	ВО	ВС	ВС

И наконец, из всех полученных оценок выбираем максимальную — в данном случае оцениваемая система должна быть отнесена к классу Business critical.

Отнеся систему к одному из классов критичности, мы можем обоснованно выбирать для неё то или иное инфраструктурное решение.

Осталось несколько нюансов, без которых описанная методология была бы неполной.

Если система обеспечивает работоспособность другой системы, то её класс критичности не может быть ниже, чем класс зависимой системы. Например, Active Directory[1] вообще никак не относится к бизнесу. Но если вдруг она встанет, то последствия для многих бизнес-приложений будут самые печальные, и поэтому AD однозначно относится к классу Mission critical.

[1] Служба каталогов, входящая в состав Microsoft Windows Server. Реализует, в числе прочих, функции аутентификации и авторизации.

Убытки, понесённые в результате простоя системы, не могут быть ниже, чем убытки, нанесённые прерыванием бизнес-процесса, который эта система обеспечивает. В свете этого правила очень интересно бывает оценить корпоративную систему электронной почты, ибо внезапно оказывается, что на неё завязан обмен критичной информацией.

Если одну систему используют несколько подразделений и их оценки отличаются, то следует использовать максимальную оценку. Мало того, даже критерии оценки могут быть разными. Так, например, оценка невозможности обслужить одного клиента может сильно отличаться в зависимости от того, что это за клиент — обычный «физик», VIP или крупная корпорация.

13.3. Базы данных в облаке

До сих пор мы рассматривали эксплуатацию базы данных на собственном оборудовании, однако жизнь может сложиться так, что потребуется база данных в публичном облаке. Эксплуатационные вопросы решает облачный провайдер, мы же в этом разделе посмотрим на вопрос с точки зрения клиента.

Простейший способ запуска БД в облаке — аренда виртуальной машины и самостоятельное развёртывание на ней экземпляра платформы. Большинство облачных провайдеров предлагают готовые образы виртуальных машин с установленным программным обеспечением.

Преимущество такого подхода в том, что клиент может выбрать любую платформу, в том числе и отсутствующую у провайдера. Расходы на эксплуатацию экземпляра будут складываться из расходов на аренду сервера и СХД, а также лицензионных отчислений разработчику платформы, если требуемое ПО не бесплатное.

Существенный недостаток самостоятельного развёртывания БД в облаке в том, что требуется развернуть и всю сопутствующую инфраструктуру — системы управления, мониторинга и резервного копирования, а также самостоятельно выполнять все регламентные работы. Поскольку вся инфраструктура и обслуживание также имеют свою стоимость, которая может быть сравнима со стоимостью самих баз, чаще всего база данных в облаке покупается в виде управляемого сервиса (managed service).

Условия продажи управляемого сервиса отличаются для монолитных и распределённых платформ.

Монолитные транзакционные платформы

«Золотым стандартом» для облачных сервисов стали PostgreSQL и MySQL. Лицензия, под которой выпускаются эти платформы, позволяет провайдерам предоставлять услуги на их основе. Есть и сервисы на основе популярных коммерческих СУБД, но далеко не всегда провайдеру и разработчику платформы удаётся прийти к взаимовыгодному соглашению:

	Amazon EC2	Microsoft Azure	Google Cloud	IBM Cloud	Oracle Cloud	Яндекс Облако	VK Cloud
PostgreSQL	+	+	+	+		+	+
MySQL	+	+	+	+	+	+	+
MariaDB	+	+					
EnterpriseDB				+			
Postgres Pro							+
Oracle RDBMS	+	+[1]	+[2]		+		
Microsoft SQL Server	+	+	+				
Db2				+			
Informix				+			

Кроме того, у некоторых провайдеров есть собственные платформы, поддерживающие интерфейс MySQL или PostgreSQL, — например, Google AlloyDB или Amazon Aurora, доступные исключительно как облачные сервисы.

Покупка уникальных сервисов может дать дополнительные возможности вашему приложению, но сделает весьма болезненным переход к другому облачному провайдеру.

Базу данных можно купить как экземпляр по требованию (on-demand instance) или как выделенный экземпляр (dedicated instance).

Экземпляр по требованию или выделенный экземпляр представляет собой виртуальную машину, запускаемую либо при необходимости, либо в момент заявки на длительный срок. Соответственно, и тарификация такого сервиса зависит от мощности виртуального сервера и от класса и объёма СХД, подключенной

[1] Услуга предоставляется компанией Oracle. Экземпляр БД работает в облаке Oracle, а специальный прокси-сервер обеспечивает маршрутизацию запросов между облаками Microsoft и Oracle.

[2] Только услуга на базе физических серверов (bare metal).

к этому серверу. Естественно, стоимость аренды сервера БД выше, чем стоимость «голой» виртуальной машины, — в цену включены установка и регулярное обновление программного обеспечения, подключение к инфраструктурным сервисам (резервное копирование, мониторинг и т. д.) и лицензионные отчисления.

Как и в случае виртуальных серверов, выделенный экземпляр дешевле, чем постоянно включенный экземпляр по требованию. При аренде экземпляра на долгий срок разница в стоимости может быть двухкратной.

Облачный провайдер берёт на себя резервное копирование ваших данных и предоставляет услугу восстановления на точку. Стоимость услуги зависит от объёма пространства, занятого резервными копиями.

Важный параметр, от которого зависит доступность и стоимость экземпляра БД в облаке, — степень резервирования. База данных может работать на единственном сервере, на нескольких серверах внутри зоны доступности (то есть ЦОДа) или на нескольких серверах в разных зонах доступности (то есть в разных ЦОДах, расположенных друг от друга на расстоянии синхронной репликации). Первая схема развёртывания самая дешёвая и применяется в основном для сред разработки и тестирования. Последняя обеспечивает наилучшую доступность, но, соответственно, она и самая дорогая.

Для особо критичных приложений можно создать базу данных, которая даже после потери целого региона (то есть нескольких зон доступности) продолжит работу в другом регионе. Пользователь может либо настроить асинхронную репликацию между регионами, либо хранить резервные копии БД в нескольких регионах. В любом случае при потере региона решение о возобновлении работы базы данных в другом регионе должен принять пользователь, а не провайдер, т. к. в этом случае потеря данных неизбежна (RPO > 0).

Для обеспечения высокой доступности БД большинство провайдеров используют физическую репликацию средствами СУБД. Из упоминаемых в этом разделе крупнейших провайдеров только Google использует блочную дисковую репликацию.

При необходимости пользователь может создать и дополнительные реплики базы данных — как синхронные, так и асинхронные — и использовать эти реплики для чтения, не нагружая основной экземпляр.

Amazon и Microsoft предлагают третий вариант доступа, когда пользователь получает только адрес базы данных, но не имеет представления о том, что стоит за этим адресом:

Облако	Английское название услуги	Русское название услуги	СУБД
Amazon EC2	Aurora Serverless	Бессерверная конфигурация Aurora	Amazon Aurora
Microsoft Azure	Azure SQL Database	База данных SQL Azure	SQL Server

Стоимость «бессерверного» варианта рассчитывается в «условных единицах» — DTU, Database Transaction Unit в Azure[1] и ACU, Aurora Capacity Units в Amazon EC2. Инфраструктура, лежащая в основе этих сервисов, не раскрывается.

Если у провайдера нет управляемого сервиса на основе какой-либо платформы, чаще всего можно запустить готовый виртуальный сервер или контейнер с образом этой платформы.

Распределённые платформы

В отличие от рынка монолитных реляционных платформ, в части распределённых платформ никакого стандарта нет. С некоторой натяжкой к «стандартным» можно отнести Apache Cassandra, MongoDB, кеш на базе Redis и полнотекстовый поиск на базе OpenSearch:

	Семейство колонок	Документная СУБД	Графовая СУБД	СУБД временны́х рядов	Распределённая РСУБД	Полнотекстовый поиск	СУБД в памяти
Amazon EC2	Keyspaces (Cassandra)	DocumentDB (MongoDB), DynamoDB	Neptune	Timestream		OpenSearch	Redis, Memcached
Microsoft Azure	Cassandra, CosmosDB	CosmosDB	CosmosDB		CosmosDB	Azure Search	Redis
Google Cloud	BigTable	Firestore			Spanner		Redis, Memcached
IBM Cloud	DataStax	Cloudant, MongoDB				Elasticsearch	Redis

[1] Базу данных SQL Server можно оценивать и по процессорным ядрам.

	Семейство колонок	Документная СУБД	Графовая СУБД	СУБД временны́х рядов	Распределённая РСУБД	Полнотекстовый поиск	СУБД в памяти
Oracle Cloud	Oracle NoSQL	Oracle NoSQL				OpenSearch	
Яндекс Облако		MongoDB			YDB	OpenSearch, Elasticsearch	Redis
VK Cloud		MongoDB				OpenSearch	Redis, Tarantool

Распределённые БД предоставляются пользователю либо как выделенные кластеры, либо как база данных в коммунальном кластере. Первый подход характерен для платформ, доступных в исходных кодах или дистрибутивах (Apache Cassandra, MongoDB и др.), второй — для собственных платформ, инсталляции которых существуют только в облаке единственного провайдера — YDB, Amazon DynamoDB, Google Spanner.

В обоих случаях стоимость сервиса складывается из платы за аренду серверов, за объём хранимых данных (включая объём резервных копий, если куплена услуга резервного копирования) и за сетевой трафик между зонами доступности и между регионами. Создавая базу данных в коммунальном кластере, клиент получает более точное ценообразование (одна «условная единица» для тарификации равна части процессорного ядра) в обмен на невозможность миграции в другое облако.

Аналитические платформы в публичных облаках представлены главным образом двумя стеками.

Первый стек — Hadoop. Тарифицируются отдельно хранение данных на HDFS и вычислительные ресурсы, затраченные на обработку этих данных. В качестве платформ для обработки предлагаются, как правило, Apache Spark и Apache Hive. В качестве дисков могут использоваться как внутренние диски физических серверов, так и тома на СХД.

Второй стек — объектное хранилище данных с интерфейсом S3 и движок аналитических запросов, работающий с внешними данными. Клиент оплачивает хранение данных и операции над ними — в зависимости либо от количества операций, либо от объёма прочитанных данных, — а также ресурсы процессора, затраченные на обработку этих данных. Примером аналитического движка может служить Apache Presto. У каждого облачного провайдера движок носит своё

название (Amazon RedShift и Amazon Athena, IBM Watson Query, Azure Synapse Analytics, Google Dataplex, Yandex Query), но кодовая база, на основе которой построены эти сервисы, не раскрывается.

Кроме того, некоторые провайдеры могут предоставлять сервисы на основе таких платформ, как Netezza, Greenplum или ClickHouse.

13.4. Процедура выбора платформы

Выбор СУБД для нового проекта может быть продиктован самыми разными соображениями — от стоимости лицензий и условий поддержки до наличия специалистов в штате компании. Но существует и формальная методика выбора платформы, применяемая в крупных корпорациях, особенно с государственным участием.

Разумеется, СУБД так никто не сравнивает, ибо их сильные и слабые стороны хорошо известны. Как правило, сравнению подлежат приложения, решающие какую-либо задачу бизнеса, но мы рассмотрим процедуру выбора на примере баз данных как предмета, не понаслышке знакомого читателям.

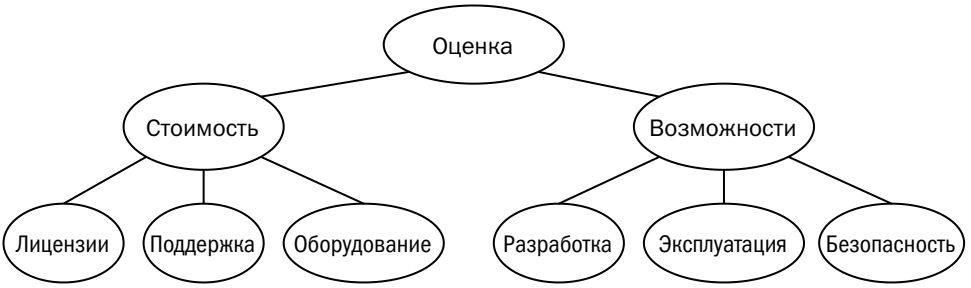
Формирование списка

Выбирая платформу для какой-либо задачи, компания пытается оценить, сколько придётся заплатить и что именно будет получено за эти деньги.

Вопросы, на которые следует ответить в процессе оценки, образуют иерархию. Корень этой иерархии — вопрос «насколько хороша данная платформа», ответом на который будет интегральная оценка, полученная из оценок стоимости и предоставляемых платформой возможностей, расположенных на втором уровне. Необходимо решить, что важнее — меньше заплатить или больше получить, и приписать каждому узлу вес. Допустим, что нам важнее качественное решение, чем дешёвое, поэтому узлу «Стоимость» припишем вес 40 %, а узлу «Возможности» — 60 %.

В крупных корпорациях обычно всё наоборот — вес стоимости не опускается ниже 50 %, а может доходить и до 80 %. В модельном примере важно лишь то, что для любого узла суммарный вес дочерних узлов должен быть равен 100 %.

Оценки, приписанные узлам второго уровня, также складываются из оценок дочерних узлов. Так, общая стоимость решения складывается из стоимости



лицензий, сопровождения и оборудования, а возможности логично оценивать в разрезе потребностей команд, которые будут этими возможностями пользоваться. В нашем примере это программисты, инженеры сопровождения и сотрудники информационной безопасности.

Теоретически дерево оценок может иметь неограниченную глубину, но обычно на практике используют трёхуровневое дерево, в исключительных случаях — четырёхуровневое.

Прежде чем приступать к оценке, следует отсечь те платформы, которые не стоит даже рассматривать. Различных СУБД известно несколько сотен, и оценить каждую из них по достаточно длинному списку критериев просто физически невозможно. Чтобы сократить пространство выбора, формулируются **отсекающие условия**, и если платформа этим условиям не удовлетворяет, то она не попадает в список сравнения.

Отсекающие условия могут относиться к технологическим особенностям — например, ACID-гарантии, реляционная модель данных, поддержка языка SQL (обратите внимание, это не то же самое, что «реляционная модель»), возможность горизонтального масштабирования. Могут быть критерии общего характера: наличие коммерческой поддержки, открытый исходный код, наличие платформы в РРПО[1], наличие экспертов на рынке (например, по результатам поиска названия платформы в резюме на сайте hh.ru).

В конце концов, могут быть критерии, специфичные для предприятия, — наличие специалистов в штате или совместимость с системой мониторинга X и с системой резервного копирования Y, на которые завязано всё сопровождение.

Главное, чтобы после исключения из списка платформ, не удовлетворяющих условиям, в нём осталось 3–5 элементов.

---

[1] Реестр российского программного обеспечения, поддерживаемый Министерством цифрового развития, связи и массовых коммуникаций Российской Федерации.

Оценка стоимости

Если сравниваемые системы относятся к одному классу (например, Microsoft SQL Server и PostgreSQL), то для простоты можно считать, что количество оборудования для того и другого решения будет примерно одинаковым. Это позволит не оценивать оборудование, сэкономив тем самым массу времени и сил. Если же приходится сравнивать совершенно разные системы (скажем, Oracle и Redis), то очевидно, что для корректной оценки необходимо сделать сайзинг — расчёт количества оборудования. Сайзинг несуществующей системы — занятие весьма неблагодарное, поэтому такого сравнения всё же стараются не допускать. Добиться этого просто: в отсекающие условия следует добавить нулевую потерю данных и реляционную модель или же, наоборот, нагрузку от 50 тысяч транзакций в секунду.

Для оценки стоимости ПО достаточно запросить у производителя платформы или его партнёров стоимость лицензии на фиксированное количество ядер и поддержки на фиксированный срок. Как правило, у компаний с производителями уже выстроены прочные отношения, и если отдел эксплуатации БД не может ответить на вопрос о стоимости самостоятельно, то для получения этой информации достаточно одного письма.

У разных производителей могут быть разные метрики лицензирования: по количеству ядер, объёму данных или количеству узлов. Резервная база может быть бесплатной, а может лицензироваться так же, как и основная. Если обнаружились какие-то различия в метриках, придётся детально описывать модельный стенд и считать стоимость лицензий для стенда.

Важный момент для корректного сравнения — одинаковые условия поддержки. Скажем, поддержка Oracle стоит 22 % от стоимости лицензии в год, а за поддержку PostgreSQL можно не платить. Корректно ли так сравнивать? Нет, потому что ошибка, которую не удастся устранить собственными силами, будет иметь совершенно разные последствия: в первом случае специалисты быстро помогут её устранить, а во втором случае есть риск задержки проекта или простоя готовой системы в течение неопределённого срока.

Уравнять условия расчёта можно тремя способами:

1) использовать Oracle без поддержки[1];
2) купить услугу поддержки PostgreSQL;
3) заложить в расчёт риски, связанные с отсутствием поддержки.

[1] Случаи отказа от фирменной поддержки в досанкционную эпоху автору неизвестны.

Расчёт рисков может выглядеть, например, так: в случае неустранимого сбоя базы данных простой системы составит один рабочий день. Планируемая прибыль от использования системы — 40 млрд условных единиц в год, вероятность возникновения аварии в течение года оценивается как $1/400$, таким образом, риск отсутствия сопровождения оценивается примерно в 100 млн условных единиц в год. Очевидно, что «планируемая прибыль» и «оценочная вероятность аварий» — величины виртуальные, но гораздо лучше иметь такую модель, чем не иметь никакой.

В действительности система может быть слишком важна, и репутационные потери от длительного простоя окажутся недопустимыми, поэтому поддержка потребуется. Если же простой допускается, то отказ от поддержки иногда может быть неплохим способом сэкономить.

Допустим, что после всех расчётов стоимость эксплуатации платформы A в течение пяти лет оказалась 800 млн условных единиц, стоимость эксплуатации платформы B — 650 млн у. е., а стоимость эксплуатации платформы C — 600 млн у. е. Платформа C как победитель получает за стоимость полновесный балл, а платформы A и B — чуть меньше, пропорционально тому, во сколько раз они дороже, в данном случае — 0,75 и 0,92 балла соответственно.

Оценка возможностей

Как мы договорились выше, в модельном примере мы оцениваем три группы возможностей — разработка, эксплуатация, безопасность. Допустим, что веса этих функций распределяются как 40 : 40 : 20.

К функциям разработки можно отнести:

- удобство манипуляции данными;
- масштабирование;
- наличие вторичных индексов.

Список критериев, как и их веса, очень субъективен. Даже при решении одной и той же задачи эти списки, веса пунктов и ответы будут существенно отличаться в зависимости от состава вашей команды. Так, например, Facebook для хранения данных использует MySQL, а Instagram построен на базе Cassandra. Вряд ли разработчики этих приложений заполняли такие таблицы. Можно лишь догадываться, что Марк Цукерберг выбрал полноценную реляционную модель, заплатив за это необходимостью прикладного шардирования, в то время как Кевин Систром заложил масштабирование средствами базы данных, пожертвовав удобством доступа к данным.

К функциям администрирования относятся:

- возможности системы резервного копирования;
- удобство мониторинга;
- удобство управления мощностями — дисками и узлами;
- возможности репликации данных.

Формулировки вопросов должны допускать количественную оценку. Можно даже договориться, как оценивать ту или иную функцию. Давайте, например, попробуем поставить оценки инструментам резервного копирования на примере инструментов, поставляемых с СУБД Oracle:

Инструмент	Комментарий	Оценка
imp/exp	Выгрузка и загрузка данных	0,1
begin/end backup	Копирование файлов	0,3
RMAN	Возможность инкрементального копирования	0,7
ZDLRA	Только инкрементальное копирование, быстрейшее восстановление на точку	1,0

Если чёткие критерии оценки отсутствуют, имеет смысл попросить нескольких экспертов выставить оценки, а затем их усреднить.

Наконец, просто перечислим функции информационной безопасности:

- наличие политик управления паролями;
- возможность подключения внешних средств аутентификации (LDAP, Kerberos);
- ролевая модель доступа;
- возможности аудита;
- шифрование данных на диске;
- шифрование при передаче по сети (TLS);
- защита данных от администратора.

Отдельно хотелось бы предостеречь от использования в качестве аргументов результатов каких-либо сторонних нагрузочных тестов.

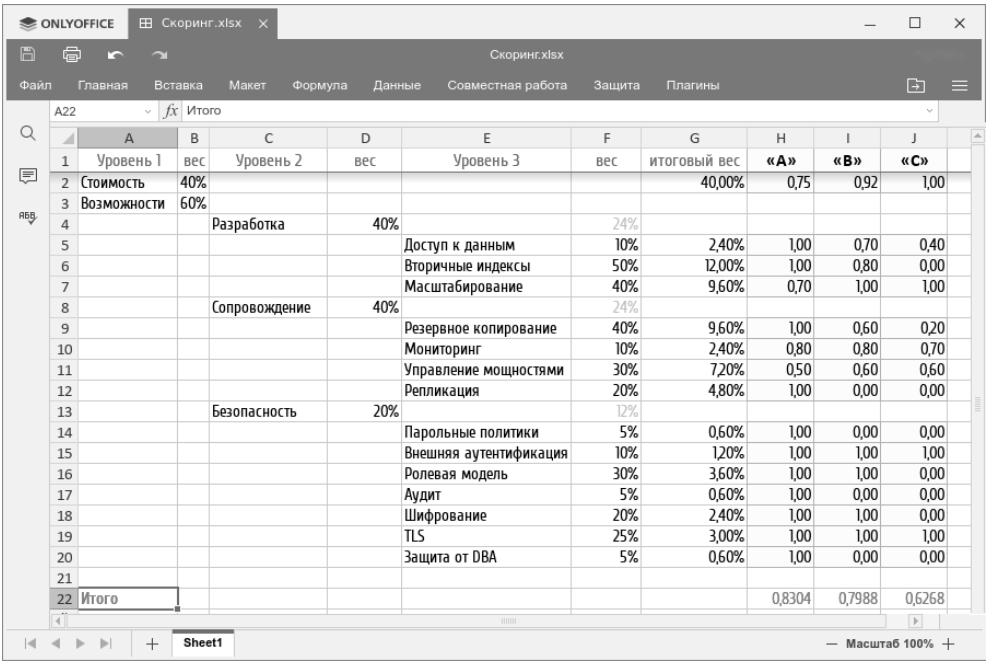
Во-первых, структура данных и профиль нагрузки тестируемых приложений могут существенно отличаться от той задачи, которую собираетесь решать вы. Во-вторых, производительность системы достаточно сильно зависит от того, под какую платформу изначально написан код и на каком оборудовании проводился тест. Автору приходилось видеть множество тестов, где Oracle сравнивали с PostgreSQL. Результаты — от безоговорочного превосходства одной системы

до столь же безоговорочного превосходства другой. И в-третьих, вы ничего не знаете о том, кто проводил тест. Важна как квалификация, влияющая на качество настройки ОС и платформы, так и мотивация, влияющая на результаты теста сильнее, чем все остальные факторы вместе взятые.

Если производительность является критически важным фактором, проведите тест самостоятельно, желательно с участием специалистов, которые будут настраивать и поддерживать промышленную систему.

Результат

Результатом проделанной работы должна стать электронная таблица, где все оценки сведены, перемножены и просуммированы:



	A	B	C	D	E	F	G	H	I	J
1	Уровень 1	вес	Уровень 2	вес	Уровень 3	вес	итоговый вес	«A»	«B»	«C»
2	Стоимость	40%					40,00%	0,75	0,92	1,00
3	Возможности	60%								
4			Разработка	40%		24%				
5					Доступ к данным	10%	2,40%	1,00	0,70	0,40
6					Вторичные индексы	50%	12,00%	1,00	0,80	0,00
7					Масштабирование	40%	9,60%	0,70	1,00	1,00
8			Сопровождение	40%		24%				
9					Резервное копирование	40%	9,60%	1,00	0,60	0,20
10					Мониторинг	10%	2,40%	0,80	0,80	0,70
11					Управление мощностями	30%	7,20%	0,50	0,60	0,60
12					Репликация	20%	4,80%	1,00	0,00	0,00
13			Безопасность	20%		12%				
14					Парольные политики	5%	0,60%	1,00	0,00	0,00
15					Внешняя аутентификация	10%	1,20%	1,00	1,00	1,00
16					Ролевая модель	30%	3,60%	1,00	1,00	0,00
17					Аудит	5%	0,60%	1,00	0,00	0,00
18					Шифрование	20%	2,40%	1,00	1,00	0,00
19					TLS	25%	3,00%	1,00	1,00	1,00
20					Защита от DBA	5%	0,60%	1,00	0,00	0,00
21										
22	Итого							0,8304	0,7988	0,6268

Разумеется, изменением весов и корректировкой оценок можно добиться любого требуемого результата, но это уже совсем другая история…

Литература

- Андрей Николаенко. Эталонные тесты СУБД: что было, что стало, что будет // Открытые системы. СУБД. 2017. № 02.
  www.osp.ru/os/2017/02/13052225
- Is it against license to publish Oracle and SQL Server performance test?
  stackoverflow.com/questions/12115397/is-it-against-license-to-publish-oracle-and-sql-server-performance-test
- Максим Тищенко. Трансформация ИТ и внедрение риск-ориентированного подхода к ИТ-управлению // VII Всероссийская конференция itSMF.
  www.itsmfcon.ru/2016/media/presentations/Bank Rossii.pdf

Часть VII

Безопасность баз данных

Глава 14
Разграничение доступа

> В жизни не думал, что могу сказать такое, и всё-таки: безопасность прежде всего!
>
> *Джоан К. Роулинг, «Гарри Поттер и Дары Смерти»*

Базовые принципы информационной безопасности составляют так называемую «триаду CIA»:

1) **конфиденциальность** (confidentiality) гарантирует, что информация не будет доступна неавторизованным лицам;
2) **целостность** (integrity) гарантирует, что информация будет сохранена в неизменном виде;
3) **доступность** (availability) гарантирует, что информация будет доступна и готова к использованию авторизованными лицами.

Мы уже рассмотрели механизмы, обеспечивающие доступность данных, а также целостность как защиту от инфраструктурных сбоев. В этой части книги будут рассмотрены механизмы обеспечения конфиденциальности, а также целостности как защиты от злонамеренных действий.

14.1. Ролевая модель доступа

Субъекты и объекты

Прежде чем говорить о разграничении доступа, необходимо договориться о терминологии.

Субъект — действующее лицо, актор, которому требуется доступ к чему-либо, в нашем случае — к базе данных.

Учётная запись (login, или **principal)** — уникальный идентификатор субъекта, человека или системы.

Пользователь базы данных (user) — учётная запись, от имени которой выполняются действия в базе данных. Пользователь может быть владельцем данных

(таблиц, коллекций) и других объектов базы данных — например, представлений и хранимых процедур.

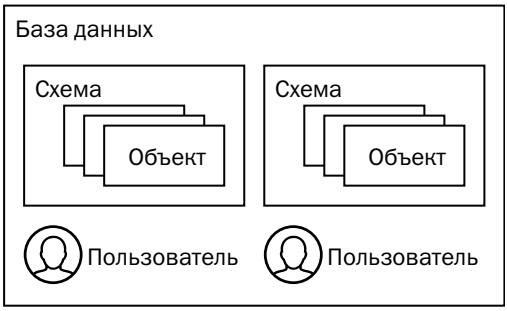
Прежде чем начать работу с данными, субъект должен быть аутентифицирован, то есть система должна убедиться, что он действительно тот, за кого себя выдаёт. Современные платформы поддерживают множество схем аутентификации, но все их можно разделить на несколько групп:

- **Доверие (trust)**. Считается, что если у субъекта есть сетевой доступ к базе данных, то есть и право подключаться к ней. Такой режим может быть установлен некоторыми платформами (например, PostgreSQL) в целях разработки или тестирования.
- **Внутренняя аутентификация (intertal authentication)**. Субъект передаёт экземпляру БД имя и пароль, и платформа проверяет соответствие переданных идентификационных данных тем, что хранятся в базе. Как правило, по сети передаётся не пароль, а его хеш-сумма. В базе также хранится хеш-сумма пароля.
- **Внешняя аутентификация (external authentication)**. Субъект использует для аутентификации операционную систему или какой-либо внешний сервис, поддерживающий один из протоколов аутентификации — LDAP, Radius, Kerberos и др.
- **Аутентификация по сертификату (certificate authentication)**. Клиент устанавливает защищённое соединение с взаимной аутентификацией (mTLS), и имя владельца клиентского сертификата используется сервером для аутентификации. Сертификат также может служить вторым фактором при внутренней аутентификации.

После аутентификации платформа ставит аутентифицированную учётную запись в соответствие пользователю БД. Затем наступает этап **авторизации**, то есть определения, какие действия может выполнять пользователь. В отличие от аутентификации, авторизация в базе данных всегда внутренняя, то есть экземпляр сам проверяет, может ли пользователь выполнить то или иное действие. Проверка происходит непосредственно в момент выполнения действия; если между двумя попытками выполнения одного и того же действия полномочия пользователя будут изменены, то результаты этих попыток могут быть разными.

Под **объектом безопасности** понимается любой объект базы данных — таблица, представление, хранимая процедура и др. Список классов объектов безопасности может быть достаточно длинным — так, в Db2 он насчитывает десяток позиций, а в Oracle — два десятка.

Идентификатор (полное имя) объекта включает идентификаторы базы данных, схемы и самого объекта. В записи различные части идентификатора, как правило, разделены точкой. В разных платформах разные части полного имени либо не требуются, либо могут быть опущены. Так, например, пользователи большинства платформ могут работать в каждый момент времени только с одной базой данных, поэтому указывать название базы не нужно, а в некоторых платформах (например, MongoDB) вообще нет понятия «схемы»[1], поэтому полное имя объекта состоит только из названия базы и имени объекта.



В Oracle и MySQL схемой называется набор объектов, принадлежащих пользователю[2], а в других платформах понятия пользователя и схемы разделены: пользователь создаёт схему, а затем может создавать объекты в этой схеме. Пользователь остаётся владельцем объекта, но имя объекта включает в себя не имя пользователя, а имя схемы. Недостаток подхода Oracle и MySQL в том, что владелец объекта не может быть изменён, в то время как другие платформы позволяют передать право владения объектом от одного пользователя другому.

При создании базы данных в ней создаётся как минимум один пользователь с административными полномочиями и как минимум одна схема, содержащая служебную информацию. Список предопределённых пользователей и схем можно найти в документации к платформе.

---

[1] В нереляционных платформах термин «схема», как правило, используется для обозначения схемы хранимого документа (например, JSON-schema), поэтому для схемы как пространства имён используются другие термины. Например, пространство имён в Apache Cassandra называется «пространство ключей» (keyspace).

[2] В Oracle начиная с релиза 18c появились пользователи без права подключения (schema-only users), однако эти пользователи по-прежнему являются владельцами объектов, содержащихся в их схемах.

Полномочия

Каждый пользователь базы данных обладает полномочиями для выполнения каких-либо действий, набор которых определяется в процессе авторизации. Эти полномочия делятся на три типа.

Полномочия на выполнение **действий над системой в целом** — над базой данных, всеми базами данных или экземпляром — выдаются администраторам системы. Действия над экземпляром — это изменение его параметров (объёма буферов памяти, ограничений на количество процессов и др.), а также работа с пользовательскими сеансами — трассировка и прерывание. К действиям над базой данных (или всеми базами данных) относится создание пользователей, резервное копирование и изменение физической структуры базы — добавление и удаление файлов данных и табличных пространств, управление файлами журналов и т. д.

Полномочия на выполнение **действий над классами объектов** позволяют их обладателям работать со множеством однотипных объектов — таблиц, представлений и др. Эти полномочия, в свою очередь, делятся на два подтипа: первые позволяют создавать собственные объекты, а вторые — работать с любыми объектами класса независимо от того, есть ли у пользователя полномочия на работу с конкретными объектами. Такие полномочия реализованы не во всех платформах.

Если, например, у пользователя БД Oracle есть полномочие `CREATE TABLE`, то он может создать таблицу в принадлежащей ему схеме. Если же у него есть полномочие `CREATE ANY TABLE`, то он может создать таблицу в любой схеме, даже не принадлежащей ему, но при этом не может ни изменить её структуру, ни прочитать из неё данные. Для того чтобы пользователь мог читать данные из любой таблицы, ему необходимо полномочие `SELECT ANY TABLE`.

В Microsoft SQL Server полномочия образуют иерархию. Так, например, получив полномочие `SELECT` на схему, пользователь может читать данные из любой таблицы этой схемы.

Полномочия на **действия над объектами** выдаются владельцем объекта другим пользователям. Сам владелец может сделать со своим объектом всё, что угодно (например, если речь о таблице, владелец может записать или прочитать данные, изменить структуру, построить индекс, удалить таблицу и т. д.), а с чужим — только то, на что получены полномочия.

Полный список объектных полномочий зависит от класса объекта. Для отношения (таблицы, представления) это чтение данных (`SELECT`), вставка (`INSERT`),

обновление (UPDATE), удаление (DELETE) и создание внешних ключей, для которых таблица является родительской (REFERENCES). Для хранимых процедур это исполнение (EXECUTE).

Полный набор полномочий в каждой конкретной платформе может быть существенно шире, чем описано здесь. Так, например, Oracle и PostgreSQL позволяют предоставлять возможность чтения не только всей таблицы, но и отдельных столбцов. PostgreSQL различает логическое (DELETE) и физическое (TRUNCATE) удаление данных, а также позволяет владельцам выдавать другим пользователям полномочия создавать триггеры на таблице (TRIGGER). MySQL и Oracle позволяют выдать полномочия на изменение (ALTER) и индексирование (INDEX) таблицы.

Логика разграничения полномочий на действия над объектами в нереляционных платформах весьма близка к логике реляционных платформ.

Как и во многих других случаях, ни единой терминологии, ни единого подхода к структуре полномочий не выработано. В таблице ниже приведены термины, используемые в разных платформах:

	Действия над системой	**Действия над классами объектов**	**Действия над объектами**
Oracle	Системные привилегии (system privileges)		Объектные привилегии (object privileges)
PostgreSQL	Атрибуты (attributes)	—	Права[1] (privileges)
Microsoft SQL Server	Разрешения уровня сервера (server level permissions)	Разрешения уровня БД (database level permissions)	Разрешения на объект (object permissions)
Db2	Полномочия (administrative authority) уровня системы, БД, схемы (system-level, database-level, schema-level)		Привилегии (privileges)
MySQL	Административные привилегии (administrative privileges)	Привилегии работы с БД (database privileges)	Объектные привилегии (privileges for database objects)
MongoDB	Привилегии над кластером или базой данных (cluser privileges, database privileges)	Привилегии над базой данных (database privileges)	Привилегии над коллекцией (collection privileges)
Cassandra	—	—	Разрешения (permissions)

[1] Как правило, термин «privileges» переводится на русский язык как «привилегии», но в переводе, выполненном компанией Postgres Professional, используется термин «права».

Получив какие-либо полномочия, пользователь может выполнять действия, предусмотренные этими полномочиями, но не может передавать полномочия другим пользователям. Однако если передача полномочий необходима, они могут быть выданы с правом передачи — в языке SQL это обычно делается командой `GRANT` с фразой `WITH GRANT OPTION`.

Управление полномочиями на уровне отдельного пользователя затратно, поэтому современные СУБД позволяют создавать **роли (roles)** — именованные наборы полномочий. Можно также рассматривать роли как группы, в которые входят несколько пользователей. Ряд платформ, например PostgreSQL и Apache Cassandra, не делают различия между «несколькими пользователями» и «одним пользователем», поэтому в них роль и пользователь — один и тот же объект.

Роль может получать полномочия и в свою очередь может быть выдана пользователю или другой роли (или, если рассматривать роль как группу, пользователь или роль может быть включён в роль). При планировании ролей важно избегать циклических зависимостей. При попытке создать такую зависимость платформа вернёт ошибку.

14.2. Ограничение доступа на уровне строк

Фильтры строк

Ролевая модель, рассмотренная нами в предыдущем разделе, определяет полномочия доступа к данным с точностью до таблиц, иногда — до столбцов. Однако многие платформы предоставляют механизмы разграничения доступа с точностью до конкретных строк. Одним из таких механизмов является фильтр строк.

В англоязычной документации встречается множество терминов для обозначения фильтра:

- Row-level security (RLS) — Oracle, Microsoft SQL Server, PostgreSQL;
- Virtual private database (VPD), Fine-grained access control (FGAC) — Oracle;
- Row and Column Access Control (RCAC) — Db2;
- Row Security Policy — PostgreSQL.

Существуют реализации подобных фильтров и для других платформ, в том числе нереляционных.

Прозрачно для пользователя фильтр добавляет к каждому запросу дополнительное условие, которому должны удовлетворять получаемые строки. Если усло-

вие выполняется, то пользователь получает доступ к строке, а если нет, то либо строка исключается из выборки, либо значения некоторых полей заменяются на пустые значения (null).

Работа с фильтром состоит из нескольких шагов.

На первом шаге создаётся условие, которому должны удовлетворять строки. Условие имеет тот же синтаксис, что и предложение `WHERE` в операторах `SELECT`, `UPDATE` и `DELETE`. Синтаксис этого условия зависит от платформы: в Oracle это функция, возвращающая подстроку для предложения `WHERE`, в Db2 и PostgreSQL — сама эта подстрока, в Microsoft SQL Server — функция, которая должна вернуть непустое отношение, если переданные ей значения полей допускают работу текущего пользователя с обрабатываемой строкой.

Помимо значений полей проверяемой строки, в условии может использоваться и информация о пользователе. Условие может проверять имя пользователя и его контекст (например, полученные роли), а также делать запросы к таблицам, кроме той, доступ к которой ограничивается.

На втором шаге к таблице привязывается политика проверки (policy), которая включает в себя созданное условие и действие. Когда политика включена, результаты любого запроса к данным, чтения или изменения, зависят от того, какой пользователь их выполняет.

Рассмотрим таблицу с музыкантами, сопровождающую нас на протяжении всей книги:

```
> select * from musician;

ID NAME     INSTR_ID
-- -------- --------
1  мартышка 1
2  осёл     2
3  козёл    1
4  мишка    3
```

Включим на таблице политику `instr_id = (select instr_id from musician where name=current_user)`, которая позволяет пользователю видеть только тех музыкантов, которые играют на том же инструменте, что и сам пользователь[1].

Теперь если мартышка получит доступ к базе данных и выполнит тот же запрос, результат будет выглядеть так:

[1] Набор команд для создания и включения политик, их синтаксис, а также список доступных функций зависят от платформы; пример иллюстрирует идею, но в точности не соответствует синтаксису ни одной из платформ.

```
> select * from musician;

ID NAME     INSTR_ID
-- -------- --------
1  мартышка 1
3  козёл    1
```

Фильтры работают при чтении данных из таблицы или коллекции и влияют в том числе и на результаты вычисления агрегатных функций:

```
> select count(*) from musician;

COUNT(*)
--------
2
```

Разумеется, применение фильтра на уровне строк влияет на производительность базы данных, особенно если для разграничения доступа используются сложные подзапросы. Главная проблема заключается в том, что фильтр вносит изменения непосредственно в текст запроса, и поэтому планы выполнения одного и того же запроса для разных пользователей могут отличаться. Это надо учитывать при разработке приложений: нагрузочное тестирование должно проводиться с включенными фильтрами и под всеми учётными записями, под которыми пользователи будут работать в промышленной среде.

Доступ на основе меток (label security)

Попытка формализации фильтров привела к созданию механизма доступа на основе меток. Как и в случае фильтров, терминология и детали реализации отличаются от платформы к платформе:

- Label security — Oracle;
- Label-based access control (LBAC) — Db2;
- Row Level Access Control (RLAC) — DataStax Enterprise[1];
- Mandatory Access Control (MAC) — PostgreSQL[2].

Для контроля доступа на основе меток каждой строке присваивается метка безопасности, а пользователю присваивается метка доступа. При доступе к данным метка каждой строки сравнивается с меткой пользователя, и строка попадает в результат выборки (или обновляется) только в том случае, если сравнение разрешает доступ.

---

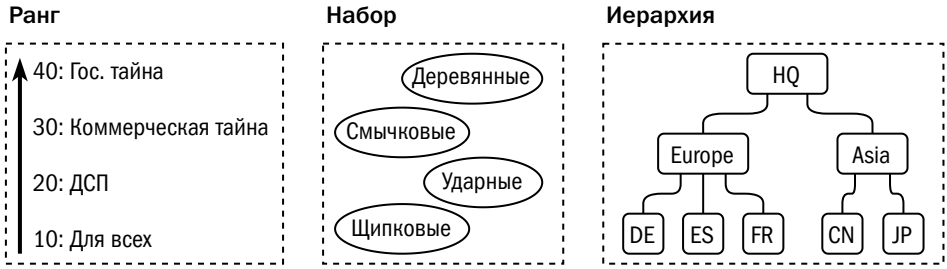
[1] Коммерческая СУБД на основе Apache Cassandra.

[2] Работает совместно с подсистемой SELinux и представляет собой скорее конструктор, чем законченное решение.

Метка может состоять из нескольких компонентов, каждый из которых имеет свой тип. От типа зависит, как именно происходит сравнение метки строки с меткой доступа пользователя. Компоненты бывают трёх типов:

1. **Ранг (Oracle: level, Db2: array)**. Компонент этого типа представляет собой число, обозначающее уровень секретности. Чем больше число, тем уже круг лиц, имеющих доступ к данным. Пользователь получает доступ к строке, если ранг его метки больше или равен рангу метки строки.
2. **Набор (Oracle: compartment, Db2: set, DataStax Enterprise: без названия)**. Компонент представляет собой набор значений. В Db2 и DataStax Enterprise пользователь получает доступ к строке, если в его метке есть хотя бы одно из значений метки строки (наборы пересекаются), а в Oracle — если в метке пользователя есть все значения из метки строки (метка строки является подмножеством метки пользователя).
3. **Иерархия (Oracle: group, Db2: tree)**. Компонент представляет собой значение из иерархического набора. Пользователь получает доступ к строке, если значение в его метке совпадает со значением в метке строки или находится выше по иерархии.

Определим три компонента меток безопасности, имеющих разный тип[1]:



Каждая строка в базе получает метку:

```
> select * from musician;

ID NAME     INSTR_ID LABEL
-- -------- -------- ------------------------------
1  мартышка 1        20 : Ударные, Смычковые : JP
2  осёл     2        40 : Щипковые : DE
3  козёл    1        30 : Ударные, Щипковые : Europe
4  мишка    3        10 : Ударные, Деревянные : CN
```

[1] Пусть вас не смущает отнесение скрипки к ударным инструментам: в конце концов, музыкант — Козёл.

Пусть теперь у нас есть несколько пользователей с разными метками доступа:

- `40 : Ударные, Щипковые : Europe`. Пользователь имеет доступ к строкам 2 и 3, но не видит строк 1 и 4 (нет доступа к элементам «Смычковые» и «Деревянные»);
- `40 : Ударные, Щипковые, Смычковые : Asia`. Пользователь имеет доступ к строке 1, но не видит строк 2 и 3 (нет доступа к регионам DE и Europe), а также 4 (нет доступа к элементу «Деревянные»);
- `20 : Ударные, Щипковые, Смычковые, Деревянные : HQ`. Пользователь имеет доступ к строкам 1 и 4, но не видит строк 2 и 3 (за счёт повышенного уровня секретности).

Физически метка доступа представляет собой одну или несколько дополнительных колонок в таблице (или полей в объекте). В DataStax Enterprise значение поля имеет прикладной смысл, а в Oracle и Db2 служит исключительно целям разграничения доступа. Управление метками осуществляется при помощи команд языка SQL/CQL (Db2, DSE) или процедур, входящих в стандартные пакеты (Oracle). У пользователя может быть несколько меток доступа, относящихся к разным таблицам или группам таблиц.

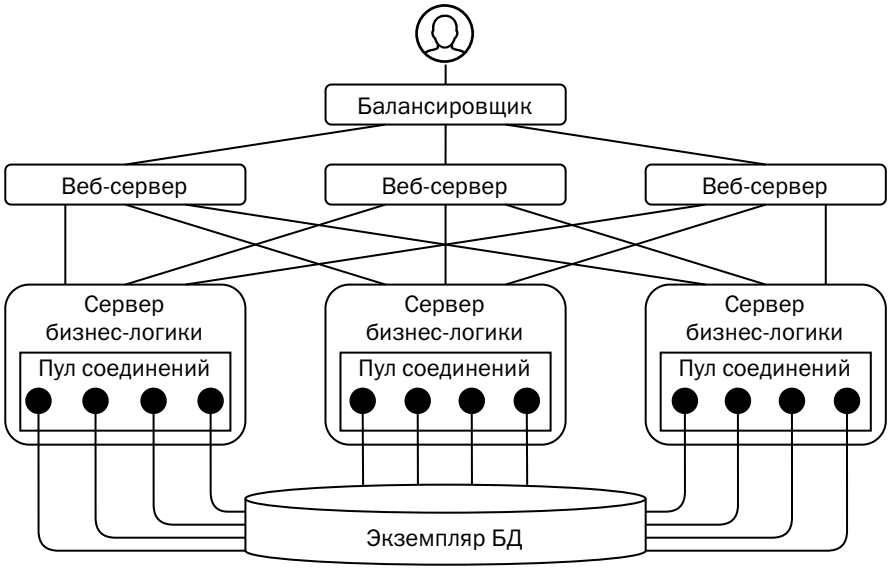
14.3. Модель доступа для транзакционных приложений

Сложные модели доступа, рассмотренные в предыдущих разделах, применимы, если каждый сотрудник работает с базой данных, используя персональную учётную запись. Такая конфигурация характерна для достаточно узкого класса систем, а именно для хранилищ данных, с которыми аналитик работает непосредственно, формируя специализированные запросы (ad hoc queries). Выполнение запросов занимает длительное время, и разделение сеанса между несколькими пользователями нецелесообразно. Количество пользователей у таких систем относительно невелико и исчисляется десятками, реже сотнями.

Для приложений, обрабатывающих транзакции, характерна другая конфигурация. Количество пользователей этих систем гораздо больше, чем количество сеансов, которое может поддержать платформа. При этом большая часть времени уходит на подготовку транзакции, а сама транзакция весьма коротка, что позволяет в одном сеансе обслуживать запросы множества пользователей. Правила

разграничения доступа в таких системах чаще формулируются в терминах процедур, которым соответствуют определённые экранные формы, чем в терминах таблиц и других объектов БД.

Архитектура системы обычно выглядит так:



Пользователь системы через балансировщик нагрузки попадает на случайный веб-сервер (front-end server), с которого, в свою очередь, попадает на случайный сервер, реализующий бизнес-логику. Сервер бизнес-логики, как правило, для каждой транзакции открывает соединение и закрывает его после завершения транзакции. Поскольку открытие соединения — дорогая операция, «закрытые» соединения на самом деле не закрываются, а сохраняются в пуле соединений (connection pool) и потом используются повторно.

Таким образом, невозможно связать пользователя системы с пользователем базы данных.

В связи с этим в транзакционных приложениях разграничение доступа осуществляется на уровне бизнес-логики и пользовательского интерфейса, а для доступа к базе данных создаётся лишь несколько учётных записей:

- персонализированные учётные записи администраторов баз данных (DBA): запуск и остановка экземпляра, изменение параметров;
- учётные записи администраторов резервного копирования (backup engineer, или backup administrator);

- персонализированные учётные записи администраторов приложения (application administrator): сбор статистики, обслуживание таблиц и индексов;
- технологическая учётная запись для системы мониторинга;
- учётная запись пользователя — владельца данных приложения;
- учётная запись для доступа приложения (все соединения на рисунке выше установлены с использованием этой учётной записи).

Есть одно исключение из этого правила. Oracle поставляет в составе своей платформы опцию Real Application Security. База данных, использующая эту опцию, рассматривает в качестве субъекта безопасности не пользователя базы данных, выполняющего запросы в сеансе, а пользователя приложения, которого приложение при помощи специализированного API ставит в соответствие каждому выполняемому запросу. Такой подход позволяет обеспечить сквозную проверку полномочий доступа и наилучшую безопасность. Однако он требует использования уникального интерфейса Oracle и не может быть перенесён на другие платформы.

Литература

- What is the meaning and difference between subject, user and principal?
  stackoverflow.com/questions/4989063/what-is-the-meaning-and-difference-between-subject-user-and-principal
- Jerome H. Saltzer, Michael D. Schröder. The Protection of Information in Computer Systems.
  www.cs.virginia.edu/~evans/cs551/saltzer/
- Pietro Colombo, Elena Ferrari. Fine-Grained Access Control Within NoSQL Document-Oriented Datastores.
  link.springer.com/article/10.1007/s41019-016-0015-z

Глава 15
Защита от внутренних угроз

Врага остеречься легче, чем друга.

Алкмеон Кротонский

В отличие от веб-серверов, серверы баз данных обычно находятся в доверенной среде, исключающей прямой доступ хакеров. В связи с этим средства защиты от незаконного доступа к базам данных достаточно примитивны. Однако существует опасность, что злоумышленник подкупом или методами социальной инженерии заставит сотрудника компании, имеющего законный доступ к базе данных, выдать её содержимое. Следовательно, данные должны быть защищены от пользователей, но при этом защита должна оказывать минимальное влияние на работу системы.

Большинство современных СУБД были спроектированы до того, как проблема защиты от пользователей встала в полный рост. Поэтому такая защита может использовать собственные механизмы, не связанные со стандартными механизмами платформ.

15.1. Принятие решения о защите

Классификация данных

Очевидно, что защита информации требует дополнительных усилий и затрат, поэтому, прежде чем принимать меры по защите, следует определить степень секретности защищаемых данных. Так же, как и при классификации информационных систем по критичности, мерой секретности данных будут последствия их обнародования.

Важно понимать, что критичность информационной системы и секретность данных, хранимых в БД этой системы, — это независимые характеристики.

Количество градаций секретности может быть разным, однако традиционная шкала насчитывает четыре уровня. Для обозначения уровней приведён один из возможных наборов терминов, общепринятой терминологии не существует:

1. **Общедоступные данные (public domain)**. Данные, относящиеся к этой категории, предназначены для публикации (пресс-релизы, официальные контакты организации, сводный баланс и т. д.) или берутся из открытых источников (ведомственные классификаторы — ФИАС, ОКВЭД и др., справочники биржевых цен или официальных курсов валют). Защищать такие данные от несанкционированного доступа бессмысленно, однако хочется ещё раз повторить, что для обеспечения бизнес-процессов может требоваться высокая доступность этих данных.
2. **Данные для внутреннего использования (confidential)**. К этой категории относятся данные, не являющиеся секретными, но не предназначенные для открытой публикации. Разглашение таких данных может повлечь за собой незначительные негативные последствия, включая дополнительные затраты на усиление мер защиты. К таким данным относятся, например, обезличенные персональные данные или IP-адреса серверов во внутренней сети предприятия. Как правило, достаточно разместить эти данные в доверенном контуре предприятия и настроить разграничение доступа.
3. **Секретные данные (secret)**. Публикация данных, относящихся к этой категории, может повлечь для предприятия существенные убытки, репутационный ущерб или претензии регуляторов. К секретным данным относится информация, составляющая коммерческую тайну, — остатки и обороты по счетам, критичные данные аутентификации и т. п., а также информация, доступ к которой регулируется законодательством, например персональные данные клиентов.
4. **Совершенно секретные данные (top secret)**. Данные, публикация которых может привести к остановке деятельности предприятия либо к значительным убыткам, таким как падение курса акций, паника, вызванная публикациями в ведущих СМИ, и т. д. К этой категории относятся, например, все сведения, составляющие государственную тайну. Большинство предприятий мелкого и среднего бизнеса не оперируют данными этой категории.

Если в организации нет человека или подразделения, отвечающего за классификацию данных, то, скорее всего, для обеспечения безопасности данных вполне достаточно помещения сервера БД в защищённый контур (локальная сеть без выхода в интернет) и настройки разграничения доступа средствами, описанными в предыдущей главе. Если же ваши данные классифицированы и среди них есть действительно секретные, тогда имеет смысл применять более изощрённые средства защиты.

Если организация работает с персональными данными клиентов или другой информацией, обработка которой регламентируется законом, то средства обработки и хранения такой информации, включая системы управления базами данных, должны быть сертифицированы. В России такой сертификацией занимается Федеральная служба по техническому и экспортному контролю (ФСТЭК).

Принципы защиты данных

Очевидно, что абсолютной защиты данных существовать не может, т. к. основная задача СУБД — всё же предоставлять данные авторизованным пользователям, а авторизованный пользователь может оказаться субъектом атаки. Таким образом, задача защиты — не предотвратить доступ к данным, а максимально быстро обнаружить неправомерное использование данных и минимизировать ущерб, нанесённый действиями злоумышленника.

Для достижения поставленной цели защита должна следовать нескольким принципам:

- **Эшелонированная защита.** Мы рассматриваем только защиту на уровне СУБД, подразумевая при этом, что используются и другие методы — физическая защита серверов, разграничение сетевого доступа, разграничение полномочий на уровне операционной системы и т. д.
- **Минимизация полномочий**. Субъект должен иметь ровно столько полномочий, сколько необходимо для выполнения служебных обязанностей, но не больше. Так, например, администратор БД не должен иметь полномочий на чтение данных, а администратор приложения не должен работать с данными под учётной записью их владельца.
- **Независимые потоки аудита**. Журналы операций, совершаемых через разные интерфейсы, анализируются совместно. Например, хранилище ключей сообщает о выдаче токена для доступа к информации, а приложение — об использовании этого токена. Обработчик журналов аудита проверяет, что токен был использован именно тем субъектом, которому он был предназначен.
- **Ограничение действия средств аутентификации**. Пароли учётных записей периодически меняются[1], а там, где возможно, используются одноразовые токены.

[1] Для учётных записей пользователя практика представляется спорной из-за известной проблемы «листочков на мониторе», но для технологических учётных записей это обязательная мера.

Модель угроз

Для определения средств защиты необходимо построить модель угроз (threat model). Если не вдаваться в формальные определения, то модель должна дать ответы на три вопроса:

1. **Кто** угрожает безопасности?
2. **Какова цель** атаки?
3. **Как** именно будет осуществлена атака?

Применительно к системам управления базами данных **целью** атаки считается чтение информации или внесение в неё изменений неавторизованными пользователями.

Субъектами атаки (ответ на вопрос «**кто?**») являются администраторы и владельцы специализированных учётных записей. Объектами атаки (ответ на вопрос «**как?**») могут быть либо стандартные интерфейсы доступа к данным (например, язык SQL), либо доступ к файлам БД в обход стандартных интерфейсов. В таблице ниже перечислены потенциальные субъекты атаки и способы, которыми они могут получить доступ к данным.

Роль	Права доступа
Администратор базы данных	Выполнение над системой управляющих действий, включая изменение параметров экземпляра БД, управление учётными записями пользователей, управление ролями
Администратор операционной системы	Полный доступ к файлам и процессам со стороны ОС
Администратор резервного копирования	Доступ на чтение к файлам БД и полный доступ к файлам резервных копий
Администратор безопасности	Управление параметрами системы, относящимися к безопасности, включая настройки аудита, парольные политики, параметры шифрования, полномочия на доступ к данным
Администратор приложения	Доступ на чтение и запись к данным приложения
Учётная запись мониторинга	Доступ к метрикам мониторинга экземпляра БД и ОС
Учётная запись владельца данных	Полный доступ к данным приложения
Учётная запись для работы приложения	Доступ на чтение и запись к данным приложения

Для защиты от атаки с использованием стандартных интерфейсов доступа к данным применяется разграничение полномочий и «правило второй руки», не позволяющее выполнить некоторые операции над платформой в одиночку. Важно, что для надёжной защиты необходимо не только наличие соответствующих механизмов, но и правильное их использование.

15.2. Средства защиты данных

Шифрование

Перехватить данные можно, прослушав сеть или получив доступ к дискам, на которых они хранятся. Следовательно, данные должны быть зашифрованы при передаче по сети и при хранении на диске.

Для защиты данных в файлах и каналах передачи данных применяется шифрование блочными симметричными алгоритмами, то есть алгоритмами, в которых ключ шифрования совпадает с ключом дешифрования. Примерами таких алгоритмов могут служить AES, ГОСТ 28147–89, IDEA, Threefish.

СУБД не имеют специфических средств шифрования сетевого трафика, поскольку с этой задачей прекрасно справляется шифрование на уровне транспортного протокола (Transport Layer Security, TLS), описанное в RFC 8446 и являющееся стандартом[1] для обеспечения безопасности любых соединений в интернете.

Многие современные платформы позволяют шифровать данные на диске. Опция шифрования так и называется — «прозрачное шифрование данных» (transparent data encryption). Иногда встречаются термины «native encryption» и «encryption at rest».

Шифрование данных на диске поддерживают:

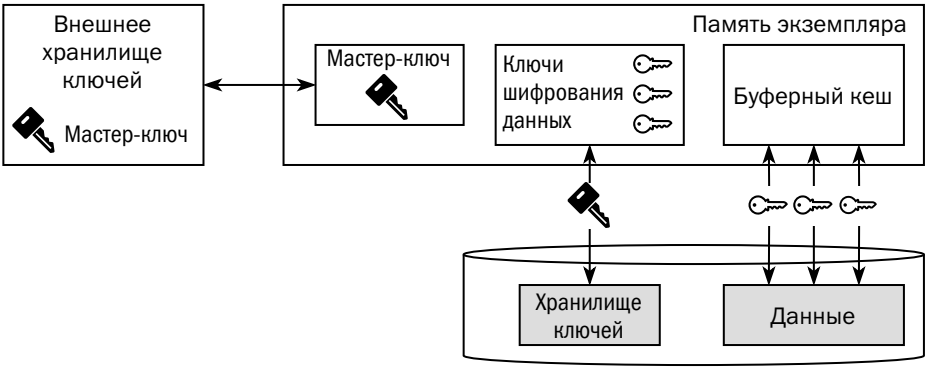
- транзакционные реляционные платформы — Oracle, Db2, Microsoft SQL Server, Pangolin[2], MySQL;
- аналитические платформы — Teradata, Vertica, SAP HANA;
- встраиваемые БД — SQLite, Berkeley DB;
- нереляционные платформы — MongoDB Enterprise, DataStax Enterprise.

---

[1] Конкретные реализации сетевых стеков могут не в полной мере соответствовать стандарту. Так, например, PostgreSQL на одном и том же порту принимает как защищённые, так и незащищённые соединения, и первый запрос после открытия соединения как раз определяет наличие защиты, что не соответствует стандарту TLS.

[2] СУБД на основе PostgreSQL, разработанная компанией Сбербанк-Технологии.

При включении шифрования администратор базы данных должен сгенерировать мастер-ключ. Затем СУБД генерирует ключи для шифрования данных, шифрует эти ключи мастер-ключом и сохраняет на диск. После этого данные могут быть зашифрованы при записи на диск и расшифрованы при чтении с использованием ключей шифрования данных. Использование нескольких ключей шифрования данных усложняет подбор ключа, если шифрованная копия всё же будет украдена, а также облегчает процесс ротации ключей.



Существует два подхода к шифрованию — шифрование отдельных колонок (полей) и полное шифрование блоков[1].

Шифрование отдельных колонок поддерживается в Oracle. Этот подход позволяет сократить объём вычислений, требуемых для шифрования и дешифрования, но при использовании современных процессоров с аппаратной реализацией криптографических функций это преимущество незначительно. Существенным недостатком такого подхода является то, что при шифровании теряется порядок сортировки значений и индексы по шифрованным полям невозможно использовать для сканирования по диапазону. В последних релизах Oracle не рекомендует использовать шифрование колонок.

Все остальные платформы шифруют блоки данных целиком[2]. Структура блока известна и может быть использована злоумышленником для подбора ключа, поэтому при шифровании к блокам данных добавляется «соль», основанная на положении этого блока в файле.

---

[1] Речь о страницах при использовании изменяемых структур или о внутренних единицах записи, обусловленных особенностями реализации неизменяемых структур.

[2] DataStax Enterprise не шифрует первичные ключи таблиц.

Платформы, использующие изменяемые структуры данных, могут поддерживать как шифрованные, так и нешифрованные данные в одной базе. Чтобы шифрование было эффективным, БД должна быть правильно настроена. Ниже приведён ряд необходимых правил, однако этот список не является исчерпывающим:

- если шифруется хотя бы одна таблица, журнал БД тоже должен быть зашифрован;
- все индексы на шифрованной таблице также должны быть зашифрованы;
- у администратора базы данных не должно быть прав на расшифровку данных (например, путём переноса таблицы в нешифрованное табличное пространство);
- ни у кого, кроме владельца, не должно быть полномочий на создание триггеров, материализованных представлений и других зависимых объектов на шифрованных таблицах.

Область действия ключа шифрования данных зависит от конкретной платформы. Ключи могут использоваться для таблицы, табличного пространства или базы данных целиком.

Ключи шифрования данных хранятся в самой БД в зашифрованном виде, а в открытом виде они появляются только в памяти экземпляра. Для шифрования ключей используется мастер-ключ, который периодически изменяется; при изменении мастер-ключа перешифровываются только ключи шифрования данных, занимающие небольшой объём, но не сами данные. Если злоумышленник получит доступ к мастер-ключу, то время, когда он сможет им воспользоваться, ограничено периодом ротации ключа.

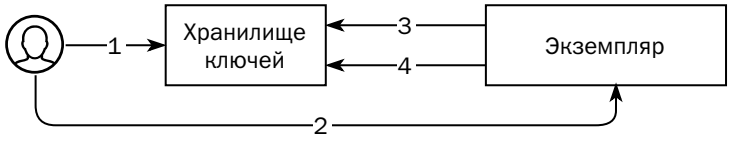
Важно помнить, что резервные копии шифрованных баз также содержат ключи шифрования, зашифрованные мастер-ключом, поэтому важно согласовывать регламент хранения резервных копий с регламентом ротации ключей.

Мастер-ключ может храниться в файле на сервере БД, однако такая конфигурация применяется главным образом на стендах разработки и тестирования. В промышленных базах данных используются специальные защищённые хранилища ключей (KMS, Key Management System):

- Oracle Key Vault — Oracle, MySQL;
- Windows Data Protection API — Microsoft SQL Server;
- HashiCorp Vault — Pangolin, MySQL;
- хранилища, поддерживающие стандартный протокол управления ключами (KMIP, Key Management Interoperability Protocol), включая Oracle Key Vault и HashiCorp Vault, — Db2, MySQL, MongoDB Enterprise, DataStax Enterprise;

- аппаратные хранилища ключей (HSM — Hardware Security Module, например, Gemalto, nCipher) — Oracle, Microsoft SQL Server, Db2.

Для получения ключей из защищённого хранилища используются сценарии, максимально затрудняющие незаконный доступ к ключам. Пример такого сценария приведён на рисунке ниже:



1. Администратор или скрипт запуска экземпляра обращается к хранилищу и получает одноразовый токен для доступа к секретной информации. Если в течение короткого времени после запроса токена не зафиксирована попытка запуска экземпляра, система аудита генерирует инцидент безопасности.
2. Полученный токен передаётся серверу, на котором запускается экземпляр.
3. Экземпляр обменивает этот токен на пароль к хранилищу секретов. Если до того, как сервер обратился к хранилищу, токен уже был использован злоумышленником, то пароль не будет выдан, информация о неудачном запросе попадёт в систему аудита, которая сгенерирует инцидент безопасности. У токена могут быть и дополнительные ограничения — время действия, IP-адрес, с которого должен прийти запрос, и др. Нарушение любого из этих ограничений также приводит к возникновению инцидента безопасности.
4. Имея пароль, экземпляр может обращаться к хранилищу секретов для получения мастер-ключа. Соединение с хранилищем секретов шифрованное, и злоумышленник не может получить пароль.

Настройка совмещения ролей

Для предотвращения несанкционированного доступа к данным важно настроить несколько ролей доступа к БД. Приведённые ниже рекомендации основаны на опыте обеспечения безопасности данных на крупнейших предприятиях, однако описанный в них подход не является единственно возможным:

- Для доступа приложения создаётся отдельная учётная запись, не совпадающая с учётной записью владельца данных.
- Для целей мониторинга создаётся отдельная учётная запись, имеющая доступ к системным представлениям, но не имеющая доступа к данным.

- Никто не знает паролей к технологическим учётным записям. Пароли генерируются роботами и сохраняются в хранилище секретов. Потребитель (приложение, инструмент обновления приложения или агент системы мониторинга) получает пароль при запуске с использованием одноразового токена.
- Администратор БД не имеет прав на чтение данных; попытки администратора выдать себе такие права или создать новую учётную запись регистрируются системой аудита и приводят к возникновению инцидента информационной безопасности.

Некоторые платформы предлагают механизм, исключающий выдачу администратором БД прав самому себе (Database Vault[1] в Oracle и «Защита от привилегированных пользователей» в Pangolin). Устройство этих механизмов отличается, но принципы работы очень похожи:

- Создаётся механизм ограничения доступа, параллельный ролевой модели.
- Создаётся роль администратора безопасности, участник которой может управлять защитой. На уровне кода платформы администратору БД запрещено получать роль администратора безопасности.
- Администратор безопасности может взять под защиту какой-либо объект — таблицу, представление, процедуру — и выдать полномочия, связанные с этим объектом, пользователям БД.
- Пользователь может получить доступ к защищённому объекту только в том случае, если это допускают одновременно и ролевая модель, и новый механизм.
- Администратор безопасности не получает никаких полномочий на доступ к объектам, кроме доступа к словарю данных, где хранится информация о защищённых объектах.

При использовании дополнительного механизма защиты несанкционированный доступ к данным невозможен без сговора администратора безопасности и администратора БД, что практически исключает попытки такого доступа.

Аудит

Журнал аудита (audit trail, или audit log) — запись событий безопасности, произошедших в системе, в хронологическом порядке. Аудит традиционно относится к средствам защиты: он не предотвращает несанкционированный доступ

[1] Не путать с Oracle Key Vault.

к данным, но позволяет быстро обнаружить факт незаконного доступа и уменьшить тяжесть последствий.

В информационных системах вообще и системах управления базами данных в частности существует два разных журнала — системный журнал и журнал аудита. При схожести технических средств, используемых в том и другом случае, эти два механизма решают разные задачи.

	Системный журнал	**Журнал аудита**
Тип событий	Изменения параметров системы, ошибки, нештатные ситуации	Операции, совершённые пользователями
Целевая аудитория	Системные администраторы	Администраторы безопасности
Настройка	Максимальный уровень событий (критические ошибки, ошибки, предупреждения, информация)	Фиксированный перечень событий и объектов
Дополнительные требования	Нет	Гарантированная запись событий; Неизменяемость журнала

В качестве примера можно рассмотреть событие переключения журнала транзакций БД на следующий файл. Это событие должно попасть в системный журнал, т. к. в случае проблем с производительностью информация о нём поможет администратору в локализации источника проблем; с точки зрения аудита это событие не интересно, поскольку не несёт никаких рисков утечки информации. Напротив, события создания пользователя базы данных и предоставления ему полномочий не интересны с точки зрения системного журнала (за исключением самых подробных уровней журналирования), но крайне важны с точки зрения безопасности, поэтому обязательно должны быть отражены в журнале аудита.

Все современные платформы имеют встроенные средства аудита. Журнал аудита чаще всего записывается в файл на локальном диске[1]. Многие платформы предоставляют возможности записи журнала аудита в таблицу базы данных (Oracle) или в журнал операционной системы (Microsoft SQL Server — журнал событий Windows или журнал безопасности Windows; PostgreSQL[2], Db2, MongoDB — syslog).

В журнал аудита в зависимости от настроек могут попадать любые события, включая доступ к данным на чтение.

---

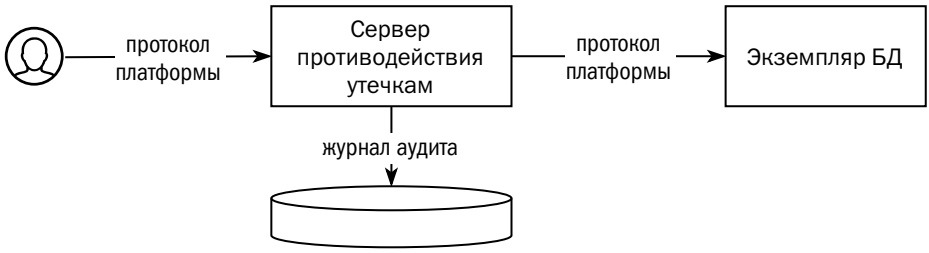
[1] Включая диски, смонтированные через NFS, CIFS или S3FS.

[2] Реализуется расширением PGAudit или подобным.

Индустриального стандарта на формат журнала аудита не существует — у каждой платформы формат свой. Данные журнала могут сохраняться в виде текста, JSON, XML или вообще в двоичном представлении.

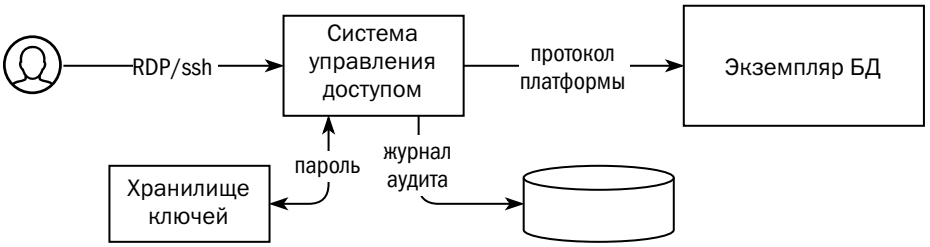
Сами по себе СУБД не предоставляют средств обработки журналов аудита, поэтому, как правило, содержимое этих журналов подаётся для анализа в системы класса SIEM (Security Information and Event Management). Помимо журналов аудита БД, эти системы в режиме, близком к реальному времени, анализируют поток событий из других источников — операционных систем и платформ запуска приложений, сетевого оборудования, систем авторизации и аутентификации, антивирусного ПО и др. В задачи SIEM входят обработка и анализ событий, поиск атак и других нарушений политик безопасности, оповещение администраторов об инцидентах информационной безопасности, а также автоматическое принятие решений по защите информации в некоторых случаях. Примерами таких систем могут служить Tivoli SIEM, Splunk, MaxPatrol, RUSIEM.

Существенным недостатком аудита на стороне СУБД является его влияние на производительность обработки данных, причём чем подробнее аудит, тем сильнее влияние. Для организации аудита без влияния на производительность могут быть использованы системы противодействия утечкам данных (DLP, Data Leak Prevention). Система ставится в разрыв между клиентским приложением и сервером БД в режиме прокси-сервера.



DLP-система анализирует трафик и при обнаружении интересных событий сохраняет их в журнал аудита. Помимо аудита, такие системы способны выполнять множество других функций, связанных с информационной безопасностью: шифрование данных на уровне колонок, защиту от привилегированных пользователей, самостоятельный анализ потока событий аудита, искажение и маскирование данных. К DLP относятся такие системы, как Гарда БД, Крипто БД (Аладдин), Database Security Service (Huawei), IBM Guardium, Imperva Database Security, Trellix (McAfee), Trustwave DbProtect и другие.

Объём данных аудита весьма велик, однако есть события, требующие особого внимания, — это любые действия, совершённые администратором БД. Для обеспечения дополнительной безопасности административных действий используются системы управления привилегированным доступом (PAM, Privileged Access Management).



Основная функция PAM-системы — разрыв аутентификации: администратор подключается к системе с использованием своей доменной учётной записи, а система устанавливает соединение с базой данных с использованием учётной записи БД. Пароль к учётной записи БД хранится в защищённом хранилище; он недоступен администратору и периодически изменяется.

Кроме разрыва аутентификации, к функциям PAM относятся:

- разрыв протокола взаимодействия, затрудняющий доступ роботов, для которых предназначены другие пути доступа к системе;
- полный аудит действий администратора, включая при необходимости видеофиксацию сеанса;
- более гранулярное разграничение доступа, чем может обеспечить СУБД;
- анализ действий администратора и реакция на потенциальные атаки вплоть до динамического понижения прав или блокировки доступа;
- интеграция с системами SIEM.

Ведущие производители систем управления привилегированным доступом — CyberArk, BeyondTrust, Centrify, Broadcom (CA Technologies).

Маскирование данных

Возможны случаи, когда пользователю системы необходимо показать не всё содержимое поля, а только его часть. Примером такого поля может служить номер банковской карты:

- для идентификации карты необходим её полный номер, состоящий из 16 цифр;

- клиенту для понимания, о какой из его карт идёт речь, достаточно показать последние четыре цифры номера;
- аналитику необходимы шесть первых цифр всех карт, т. к. в них закодированы банк-эмитент, платёжная система, тип карты и другая информация, не раскрывающая тайну конкретного клиента.

Задачу можно решить несколькими способами, например управлять отображением поля на уровне сервера бизнес-логики или выдать пользователям базы данных полномочия на доступ к специально созданным представлениям. Однако современные платформы предлагают ещё один механизм для решения этой задачи — **маскирование данных (data masking)**. Механизм доступен в Oracle, Db2 for z/OS, Microsoft SQL Server и PostgreSQL[1].

Для маскирования данных создаётся маска, представляющая собой функцию, применяемую к значению колонки, и политика, привязывающая маску к субъекту безопасности — пользователю базы данных. Синтаксис команд создания маски и список доступных функций зависят от конкретной платформы.

Предположим, мы создали маску, показывающую пользователю только первую букву английского названия музыкального инструмента — violin (скрипка) и viola (альт) начинаются с одной и той же буквы v:

```
> select m.name, i.name_eng as iname
  from musician m join instrument i on m.instr_id=i.id;

NAME     INAME
-------- -----
мартышка v***
осёл     v***
козёл    v***
мишка    c***
```

Маскирование эффективно защищает информацию от просмотра, но совершенно не годится как механизм разграничения доступа. Значения колонок заменяются на маски только в момент вывода, но во время исполнения запроса используются исходные значения, что может привести к раскрытию содержимого:

```
> select i.name_eng as iname, count(*)
  from musician m join instrument i on m.instr_id=i.id
  group by i.name_eng;

INAME COUNT(*)
----- --------
v***  2
v***  1
c***  1
```

[1] При использовании расширения postgresql_anonymizer, pg_datamask или аналогичного.

Литература

- Владимир Харчиков. Акторы сценариев утечки пользовательских данных.
- Владимир Харчиков. Сценарии утечки пользовательских данных.
- AppRole Pull Authentication.
  learn.hashicorp.com/tutorials/vault/approle
- Павел Лего. Обзор рынка систем защиты баз данных (Database Security).
  www.anti-malware.ru/analytics/Market_Analysis/Database-Security
- Денис Сарычев. Контроль привилегированных пользователей (PAM) — обзор мирового и российского рынка.
  www.anti-malware.ru/analytics/Market_Analysis/PAM-market-review

Послесловие

До конца всего осталось
Несколько минут,
И меня, какая жалость,
Видно, не убьют...
Наум Олев,
«Фортуна — лотерея»

Вот и закончилась книга.

Возможно, после того как вы её прочитали, вам покажется, что вы знаете о базах данных всё. Мне тоже так казалось, когда я начинал эту книгу писать. Нет, ну а что такого? Вот сейчас осчастливлю мир своим знанием, надо только проверить на всякий случай в документации... ой, а тут всё совсем не так!

Вот так за чтением источников и разнообразными экспериментами пролетело время. Как говорил мудрец, круг, представляющий собой модель наших знаний, вырос, но выросла и его граница с неведомым[1]. И теперь остаётся вслед за Сократом повторить: «Я знаю, что ничего не знаю». Но, подобно древнегреческому мудрецу, можно утешать себя тем, что многие не знают и этого.

Круг получился не такой уж маленький, и находиться внутри него может быть весьма комфортно. Но тот, кто не любит замкнутых пространств, волен выбрать точку на его границе и отправиться от неё в неведомое своей дорогой. А мне бы хотелось напоследок порассуждать о том, куда идёт индустрия баз данных и куда она может прийти в недалёком будущем.

Сегодняшняя ситуация чем-то напоминает положение 60-х годов XX века: современные платформы прекрасно решают задачи, которые ставит перед ними бизнес. Никакой революции не намечается, все изменения носят эволюционный или даже косметический характер.

Малые предприятия стремятся уйти от дорогих коммерческих платформ. Поскольку за время их эксплуатации были накоплены огромные объёмы хранимого кода, платформы, идущие на смену (например, Enterprise DB или Postgres Pro), учатся выполнять этот код, стремительно наращивая стандартные библиотеки.

[1] Несколько обнадёживает, что длина границы с неведомым растёт пропорционально радиусу, в то время как площадь круга знаний — пропорционально квадрату радиуса.

Эти библиотеки порой включают функции, совершенно не связанные с обработкой данных, — например, отправка электронных писем прямо из хранимой процедуры[1].

Крупные корпорации озабочены проблемами управляемости, отказоустойчивости и безопасности, и многие разработчики создают свои инструменты управления и мониторинга. В то же время бурно развиваются системы искусственного интеллекта, поэтому вполне можно ожидать, что роботы — администраторы баз данных (как, например, Oracle Autonomous DB) в ближайшее время станут привычным явлением.

Мошенники проявляют всё больше фантазии для получения доступа к данным компаний, и чтобы справляться с этим натиском, рынок средств безопасности растёт бешеными темпами. Полагаю, нас ждут новые захватывающие состязания снаряда и брони.

Уже который год со дня на день ожидается прорыв в надёжности и производительности распределённых систем. Такие платформы, как YugabyteDB, CockroachDB, Picodata, ScyllaDB, давно готовятся подвинуть с пьедестала традиционные монолитные системы, и в настоящее время надежды связаны с различными вариантами алгоритма детерминированных транзакций. Честно говоря, я не верю, что тут совершится революция. Но я лично знаю нескольких замечательных инженеров, занимающихся распределёнными системами и убеждённых в успехе, и искренне желаю им удачи.

Окинув взглядом историю развития систем управления базами данных, можно заметить, что она, как и любая другая история, развивается по спирали. А сделав такое наблюдение, можно попробовать угадать, что нас ждёт на новых витках.

...Давным-давно средства для работы с данными были встроены в языки программирования — например, MUMPS или dBase. Позже мы увидели, как базы данных выделились в самостоятельный класс программного обеспечения, и в какой-то момент их роль была низведена до простого хранилища значений, рассчитанных приложением.

Сегодня малый и средний бизнес отчётливо ощущают проблемы роста: потребность в автоматизации растёт, а найти квалифицированных программистов чем дальше, тем сложнее. Всё больше крепнет желание разрабатывать полноцен-

[1] Это не шутка. В составе стандартной библиотеки Oracle есть пакет `UTL_SMTP`, позволяющий работать с электронной почтой, и совсем недавно компания Postgres Professional по многочисленным просьбам пользователей добавила аналогичный пакет в свою СУБД.

ные приложения с использованием единственной технологии, чтобы не собирать команду специалистов по всем компонентам сложного стека, и вот уже почти десятилетие наблюдается сближение баз данных и сред разработки и исполнения бизнес-логики.

С одной стороны, компании-разработчики СУБД выпускают инструменты, максимально интегрированные с базами данных, — например, Oracle APEX[1] или Microsoft PowerApps[2]. Есть и независимые продукты класса «готовый бэкенд» (BaaS, Backend-as-a-Service), позволяющие разработчику не думать о создании структуры данных. В основном они представлены облачными сервисами, например Google Firebase[3] или backendless.com[4], но есть и платформы, которые можно установить на собственной инфраструктуре, например Parse[5].

С другой стороны, многие СУБД стремятся предоставить простые интерфейсы в стиле REST, а некоторые, например ArangoDB[6], сами содержат движок JavaScript. Не исключено, что в недалёком будущем мы увидим платформу, позволяющую разработать всё приложение целиком — от структуры данных до пользовательского интерфейса, как когда-то это могли Clipper, FoxPro и Microsoft Access®.

...Давным-давно установка и настройка баз данных были доступны лишь избранным. Когда реляционный подход начал набирать популярность, такие платформы, как ADABAS, обзавелись реляционным фасадом. Разработчиков ничуть не заботило, что инсталляция их продукта, и без того сложная, усложнилась ещё сильнее за счёт нового слоя, который надо отдельно развернуть и настроить. А потом пришли Oracle и Informix, установка которых заключалась в нажатии кнопки «next» в графическом инсталляторе, и Db2, которая была встроена прямо в операционную систему. И за счёт своей простоты эти платформы захватили рынок.

Сегодня многие клиенты предпочитают не разворачивать собственные базы данных, а пользоваться готовыми услугами в облаке. Облачные провайдеры, в отличие от мелких компаний, не связанных с информационными технологиями, могут позволить себе и сложную инфраструктуру, и лучших инженеров. Благодаря

---

1 apex.oracle.com.

2 powerapps.microsoft.com.

3 firebase.google.com.

4 backendless.com.

5 parseplatform.org.

6 arangodb.com.

этому появляются такие платформы, как Amazon Aurora[1], Google AlloyDB[2] или Microsoft SQL DB Hyperscale (на движке Socrates[3]), которые сочетают предсказуемость монолитных баз данных с возможностями горизонтального масштабирования.

Информация об архитектуре этих систем весьма скудна и отрывочна, а область обитания ограничена облаками, где их развивают и обслуживают специалисты высочайшей квалификации. Но шила, как известно, в мешке не утаишь, и не исключено, что принципы, заложенные в эти платформы, будут реализованы и в системах с открытым исходным кодом, а сложная инфраструктура станет такой же привычной, как обычная операционная система обычного компьютера.

...Давным-давно разработчики систем управления базами данных прибегали к невероятным ухищрениям, чтобы выжать все возможности из имеющихся в их распоряжении компьютеров. Потом начался экспоненциальный рост мощности аппаратуры, описанный законом Мура, и внезапно оказалось, что серверы избыточно производительны и часть их мощности можно потратить на всевозможные сервисные функции. Специализированные компьютеры ушли в небытие, уступив место обычному железу, так называемому «commodity».

Сегодня потребности в обработке данных вплотную подобрались к пределам возможностей оборудования, а закон Мура упёрся в законы физики. Инженерам предстоит изыскать резервы, которые позволят работать с данными ещё быстрее. Из традиционных процессоров и памяти уже выжато всё, что можно, а значит, нас может ждать возврат к специализированному оборудованию. Возможно, это будут ПЛИС и «умные» твердотельные диски, на которые когда-то делала ставку Netezza. А может быть, прорывом станут структуры данных, рассчитанные на работу с быстрой энергонезависимой памятью, ситуация с которой напоминает замкнутый круг: производители оборудования не наращивают производство, поскольку для энергонезависимой памяти нет эффективного программного обеспечения, а программисты не пишут код для экзотического оборудования. Надеюсь, что когда-нибудь этот круг будет разорван, и возможно, гипотеза о невозможности создать оптимальную структуру данных наконец-то будет опровергнута.

...Давным-давно проектированием баз данных занимались учёные и инженеры, воспитанные в традициях научного подхода. Для них реляционная модель бы-

---

1 pages.cs.wisc.edu/~yxy/cs764-f20/papers/aurora-sigmod-18.pdf.

2 www.dragonsegg.xyz/google-alloydb-architecture-deep-dive.

3 www.microsoft.com/en-us/research/publication/socrates-the-new-sql-server-in-the-cloud.

ла откровением, поскольку она позволяла разрабатывать приложения не «на глазок», а руководствуясь строгой методологией.

Сегодня стартаперы берутся осчастливить своими творениями мир, порой не закончив среднюю школу, и потому реляционная концепция для них слишком сложна. С этим связан рост популярности документоориентированных баз данных, где весь мир описывается документами в формате JSON. Даже взрывной рост популярности PostgreSQL его разработчики связывают не в последнюю очередь с возможностью хранения и обработки JSON.

К счастью, далеко не все пользователи баз данных — метросексуальные потребители смузи в коворкингах: разработчикам искусственного интеллекта тоже нужны инструменты надёжного хранения и удобной обработки данных. И вот уже появляется новая модель данных — векторная. Пока специализированные платформы, такие как Pinecone или Milvus, находятся в самом низу рейтинга db-engines.com, а расширение pgvector для PostgreSQL не входит в сотню самых популярных, но начало нового тренда уже налицо.

Конечно же, интересно будет прочесть этот прогноз через десять–пятнадцать лет. Наверняка мы увидим что-то другое или даже совсем третье. Но если через десять лет ко мне подойдёт молодой человек с горящими глазами и скажет: «Вот вы писали так, а на самом деле всё вот так...», то мне будет очень приятно. Это будет значить, что книга выполнила своё предназначение.

В добрый путь!

Предметно-именной указатель

N

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛОН ПРЕСС», «КТК Галактика»).
Адрес: г. Москва, пр. Андропова, 38, оф. 10;
тел.: (499) 782-38-89, электронная почта: books@alians-kniga.ru.
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: http://www.galaktika-dmk.com/.

Комаров Владимир Иванович

Путеводитель по базам данных

При поддержке Postgres Professional
https://postgrespro.ru

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Редактор *Рогов Е. В.*
Корректор *Синяева Г. И.*
Иллюстрация на обложке *Комарова О. В.*

Формат $70\times100^{1}/_{16}$. Печать цифровая.
Гарнитуры CharterITC, FranklinGothicITC (Паратайп) и Iosevka (Renzhi Li).
Усл. печ. л. 42,25. Тираж 200 экз.
Веб-сайт издательства: www.dmkpress.com