УДК 004.65
ББК 32.972.134
К63

Комаров В. И.

К63 Путеводитель по базам данных. — М.: ДМК-Пресс 2024. — 482 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

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 спарринга́м. 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 РАН **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 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 её директору **Олегу Бартунову** 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 к читателю 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:

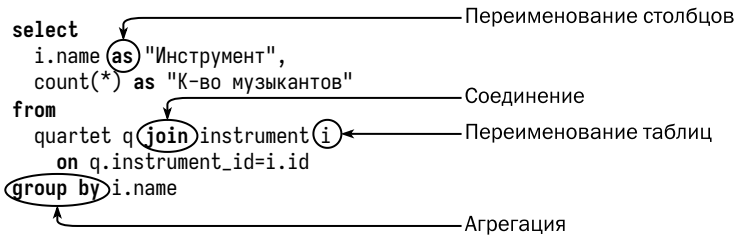
- переименование (rename) — изменение имён отношений и атрибутов (таблиц и колонок) в рамках текущей операции;
- проекция (projection) — выборка подмножества колонок[1];
- фильтрация (selection) — выборка подмножества строк, соответствующих заданному условию;
- соединение (join) — составление новых строк, 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 нет 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-куэль», 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, это 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 в базах данных предусмотрены специальные механизмы — последовательности

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

(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 на складе неотрицательно, или что в поле «пол» не записано ничего кроме «М»

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

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.

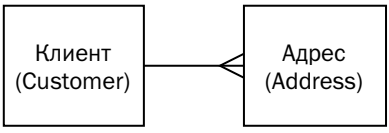
Диаграммы «сущность-связь»

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 между блоками. Диаграммы используются и для проектирования физической модели данных, и тогда блоки обозначают таблицы, а линии — внешние ключи.

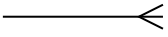
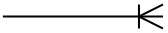
В большинстве случаев логическая модель совпадает с физической, то есть каждой сущности соответствует таблица, а каждой связи — внешний ключ. Однако существуют и исключения, например:

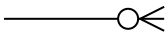
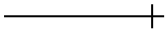
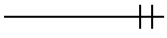
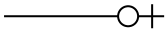
- в реляционной модели невозможно непосредственно создать связь «многие-ко-многим»;
- несколько сущностей (например, справочников) могут храниться в одной таблице;
- одна сущность может распадаться на несколько таблиц: часть атрибутов в одной таблице, часть — в другой;
- одна таблица может хранить текущее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



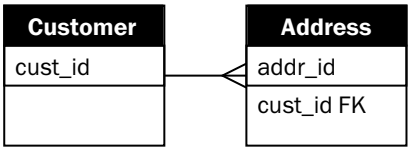
На диаграмме выше изображены две таблицы — клиент (customer) и адрес (address), причём таблица адресов ссылается на таблицу клиентов, то есть содержит внешний ключ. В данном случае логическая модель совпадает с физической.

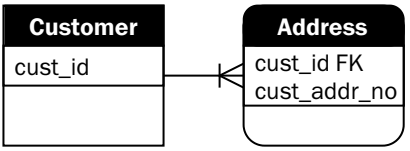
Существует несколько нотаций для диаграмм, однако наиболее распространённая — «вороньи лапки» (crow's foot). Для моделирования «на салфетке» достаточно линий, как на рисунке вверху — «лапка» цепляется к дочерней сущности, а линия без декораций — к родительской. Однако полная нотация включает в себя несколько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



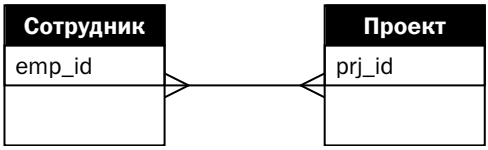


Пусть, например, в базе есть таблица «клиент» с суррогатным ключом. Для дочерней таблицы «адрес» мы можем также генерировать суррогатный ключ, а ключ клиента добавить обычным полем — внешним ключом. В этом случае диаграмма будет выглядеть как на рисунке слева.

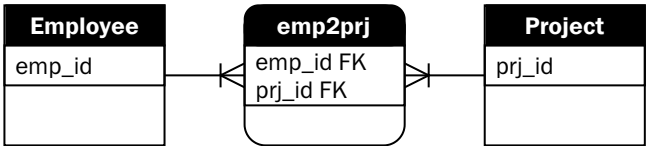
Другой способ идентификации адреса — идентификатор клиента и номер адреса для этого клиента, то есть внешний ключ является частью первичного ключа. В этом случае диаграмма будет выглядеть как на рисунке справа.

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": 00, "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": 54, "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	10000
42	Василий	кредит	917	автокредит	−75200
91	Анна	моя карта	800	текущий счёт	7000
91	Анна	на чёрный день	954	вклад	30000

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	10000
1002	42	кредит	917	автокредит	–75200
1003	91	моя карта	800	текущий счёт	7000
1004	91	на чёрный день	954	вклад	30000

Обратите внимание на два важных изменения:

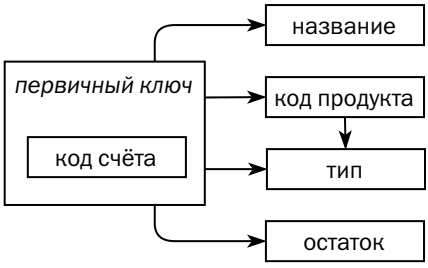
- в таблице «Счёт» сохранилось поле «Код клиента», и теперь оно является внешним ключом на таблицу «Клиент»;

- 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	10000
1002	42	кредит	917	−75200
1003	91	моя карта	800	7000
1004	91	на чёрный день	954	30000

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 потребовали 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 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 документо-ориентированных СУБД и семейств колонок.

Документо-ориентированные БД

Документо-ориентированная база данных — это хранилище «ключ — значение», где значение является не просто набором байт, 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 документо-ориентированные 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 документо-ориентированные 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 документо-ориентированной или ещё нет. Обычно документо-ориентированные 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 документо-ориентированные СУБД не умеют выполнять операций над несколькими наборами результатов — пересечение (intersection), вычитание (complement), объединение (union) и, конечно же, соединение (join).

Вопреки распространённому заблуждению, для эффективной работы с документо-ориентированной 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" }
  ]
}
```

Такой 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, документо-ориентированная БД при использовании такой структуры данных потребует два запроса: 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 документо-ориентированная 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 используется JSON-документ (Amazon Dynamo, 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 документо-ориентированных 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 хранит 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 в самих элементах (как, 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 документо-ориентированных платформ, использующих XML как основной формат хранения данных — Native XML Databases. К ним относятся MarkLogic,

Virtuozo, Oracle BerkeleyDB 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 web-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 байт» (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, позволяющий 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.

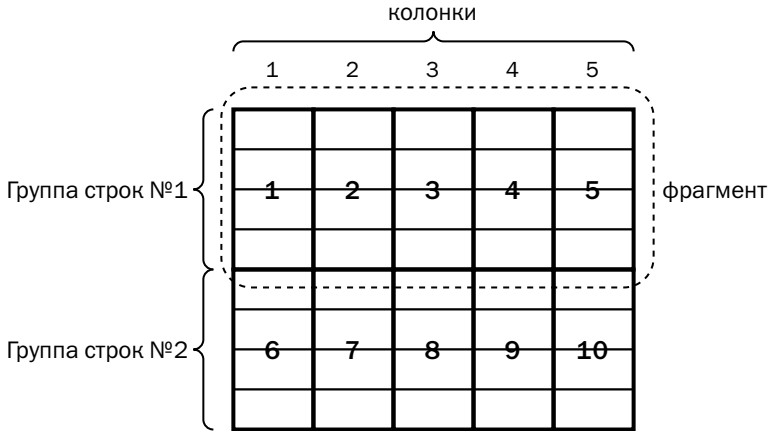
Представление данных в JSON, BSON или MsgPack хоть и компактно по сравнению с XML, всё же достаточно объёмно за счёт хранения схемы, то есть имён (а в BSON — и типов) атрибутов в каждом документе. Для экономии места и ускорения обработки применяются форматы, где схема данных хранится отдельно от самих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



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, это **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 документо-ориентированных 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`.

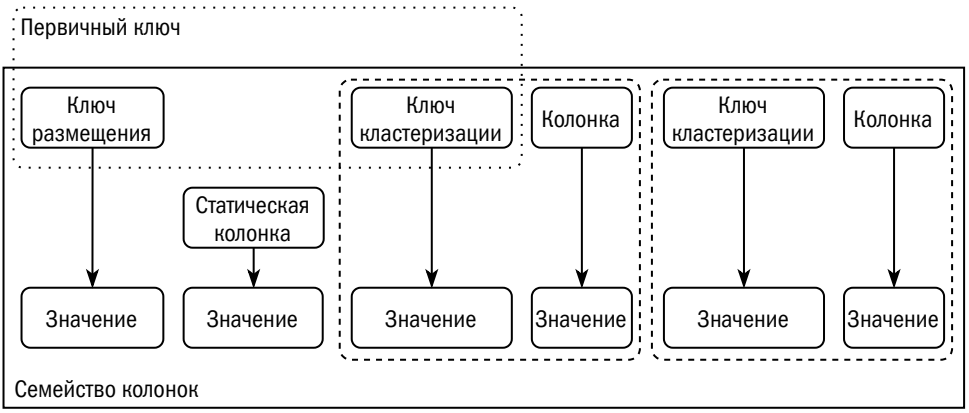
Если в HBase первичный ключ представляет собой набор байт, который приложение должно как-то интерпретировать, то в таких платформах как Cassandra и ScyllaDB, использующих CQL, первичный ключ — это типизированная колонка или набор колонок.

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

Первое поле (или несколько полей) составляют ключ размещения (partition 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 документо-ориентированных БД — возврат части атрибутов, поиск по значениям неключевых атрибутов и поддержка вторичных индексов. Так же, как и документо-ориентированные 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 временную 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, Eclipse Link 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 была 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.

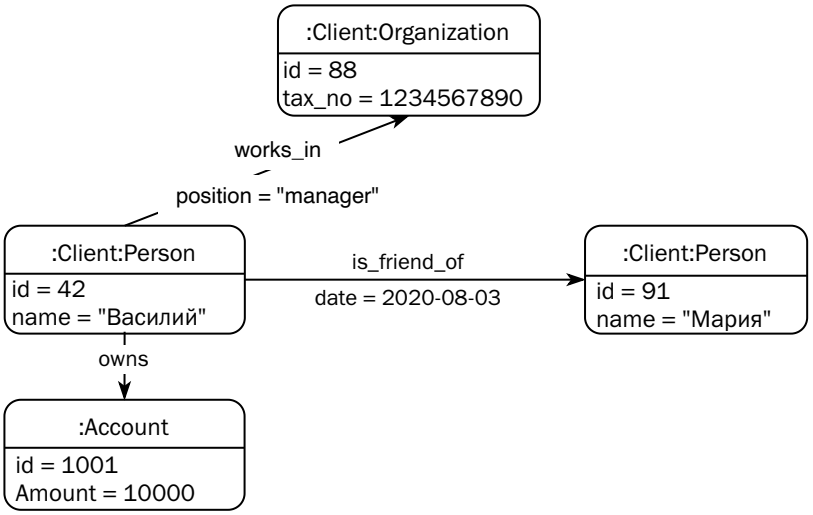
Графовые БД

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 выше 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 выше 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 обязаны существовать, а значение или наборы их 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 документо-ориентированной 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 Bhatia, «Форматы файлов в больших данных: краткий ликбез» (перевод 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