Сергей Тарасов

СУБД для программиста

Базы данных изнутри

СОЛОН-Пресс

2015

УДК 621.396.218
ББК 32.884.1
Т 19

Тарасов С. В. СУБД для программиста. Базы данных изнутри. — М.: СОЛОН-Пресс, 2015. — 320 с.: ил.

ISBN 978-2-7466-7383-0

EAN 9782746673830

Книга охватывает различные этапы разработки и сопутствующие им ситуации из практики программистов приложений, работающих с системами управления базами данных. Даются рекомендации по выбору решений как в проектировании (архитектуре), так и в программировании автоматизированных информационных систем уровня предприятия. Приводятся примеры для различных СУБД и моделей: Microsoft SQL Server, PostgreSQL, Firebird, Oracle, XML, NoSQL.

Для программистов, студентов и других специалистов в области информационных технологий, а также всех интересующихся темой разработки приложений баз данных.

ISBN 978-2-7466-7383-0
EAN 9782746673830

Посвящается моему отцу...

Цивилизованный человек отличается от дикаря главным образом благоразумием, или, если применить немного более широкий термин, предусмотрительностью. Цивилизованный человек готов ради будущих удовольствий перенести страдания в настоящем, даже если эти удовольствия довольно отдалены.

Б. Рассел, «История западной философии»

Содержание

Введение

Разработка приложений баз данных распространена не только в «корпоративном секторе» автоматизации производственных процессов предприятий и их отделов. Из классического определения «программы — это алгоритмы плюс данные» следует, что сколь верёвочке не виться, пройдя через цепочку служб, запрос в итоге обрабатывается системой управления базами данных (СУБД) или её неполнофункциональным аналогом. Активно развивающийся рынок мобильных устройств широко использует встраиваемые (embedded) СУБД, ранее применявшиеся в основном для управления оборудованием.

Если программист сознательно не ограничивает себя разработкой служб и человеко-машинных интерфейсов, взаимодействующих исключительно с другими службами, то вскоре возникает необходимость непосредственной работы с какой-либо СУБД, вероятнее всего реляционной.

Как правило, обладателям профильного образования преподаватели читали соответствующий теоретический курс в вузе [2], сопровождаемый практическими занятиями, пересекающийся по тематике с другими предметами. И, тем не менее, приступая к производственным задачам, вчерашний студент быстро ощутит отличие академических подходов от открывающегося взгляду пейзажа строек в недрах корпоративного софтостроения или в глубоком тылу веб-служб.

Тяжелее пришедшим в программирование из других областей деятельности. Окунувшись в реальность без багажа теории, соответствующего, как говорят американцы, бэкграунда, трудно сформировать в голове целостную картину, охватывающую важные детали происходящего, пропуская несущественные. Не хватает ни времени, ни мотивации читать достаточно скучные, толстые монографии вроде многократно переизданного Дейта [1], когда надо решать текущие задачи. Программисту становится не до вопросов философии кунг-фу, освоить бы побыстрее основные удары и блоки, чтобы получать поменьше оплеух от брыкающейся техники, исполненных значимости системных администраторов и недовольного начальства. Хорошо, если удастся выкроить часок-другой и потренироваться в сиквеле[1] на примерах из книжки Грабера [3]...

---

[1] Сиквелом на жаргоне программистов называется язык SQL (Structured Query Language). Первоначальная его версия так и называлась SEQUEL. Впрочем, если вы будете говорить просто «эс-ку-эль», это не повлияет на ваш профессионализм.

Справиться с растущим потоком информации, не пропуская её через фильтры теоретического багажа, становится всё труднее. Например, последние годы активно пропагандируют NoSQL, может быть там все будет проще, не надо задумываться о нормальных формах и тренировать мозги на непривычную множественную обработку? Или может быть лучше работать через проекцию таблиц на объекты и не использовать прямой доступ к базе данных?

Книга «Софтостроение изнутри» [13] посвятила немалое количество сюжетов теме «**как это не надо делать**» и прогрессирующей в среде программистов некомпетентности в области баз данных, приводящей к катастрофическим для проектов последствиям на более поздних стадиях. В отличие от предтечи, настоящее издание будет носить ровно противоположный характер, следуя принципу «**как это лучше сделать**». Опираясь на опыт работы в продуктовом софтостроении и в технической экспертизе СУБД-решений, автор постарается в рамках повествования помочь вам не утонуть в информационном потоке и разобраться в часто возникающих на практике проблемах, не отрывая их от теории.

Почему не блог, а книга? Действительно, в Сети можно найти немало интересных статей. Однако, во-первых, чтобы найти нужную информацию по правильно заданным ключевым словам, а, во-вторых, оценить достоверность найденной публикации, нужно уже иметь определённый уровень компетенции, который кроме как чтением книг и практикой не поднять. Поэтому рекомендую и начинающим, и программистам с небольшим (2-3 года) опытом не увлекаться малоосмысленным копированием кода со страниц в Сети и прочим натягиванием глобуса на Меркаторову проекцию.

Цель книги не в том, чтобы заменить чтение упомянутых выше монографий или других книг из прилагаемого списка литературы, но подготовить и подвести к нему осознанно, исходя из нужд решения практических задач. Попытаться выстроить мост между бескомпромиссным академическим гранитом и производственными зыбучими песками, бесследно засасывающих неосмотрительных путников своими половинчатыми решениями и постоянной текучкой.

Все-таки, нет ничего практичнее, чем хорошая теория.

Основные понятия

«...пока мы не знаем закона природы, он, существуя и действуя помимо, вне нашего познания, делает нас рабами «слепой необходимости». В. Ульянов (Ленин)

База данных и СУБД

Основополагающие термины достаточно подробно рассмотрены в любой из монографий [1,2,4,5]. Тем не менее, чтобы однозначно понимать суть в ходе чтения, нам нужно выработать общий словарь. С этой целью я буду по мере необходимости приводить минимальный набор определений, составляющий основу дальнейшего изложения.

База данных (БД) — структурированное поименованное хранилище информации.

Из самого первого определения должно быть понятно, что содержащий начисленные квартальные премии сотрудников файл формата CSV[2] является хоть и очень простой, но базой данных, а текстовый файл с научным описанием мышей-полёвок ей не является.

Система управления базами данных (СУБД) — специализированное программное обеспечение, обеспечивающее доступ к базе данных как к совокупности её структурных единиц.

Из второго определения также должно быть понятно, что открыв упомянутый CSV-файл в текстовом редакторе, мы хоть и видим базу данных, но не работаем с ней посредством СУБД. Если же мы откроем файл в приложении LibreOffice Calc, то данная программа превращается хоть и в очень простую, но СУБД, позволяющую нам работать со структурированным текстом, как с таблицей на уровне колонок, строк и значений, а также посчитать итоги, средние и крайние величины. Положив

[2] CSV (от англ. Comma-Separated Values — значения, разделённые запятыми) — текстовый формат, предназначенный для представления табличных данных. Каждая строка файла — это одна строка таблицы. Значения отдельных колонок разделяются разделительным символом (запятой).

файл в разделяемый по локальной сети каталог, получаем примитивную многопользовательскую СУБД на основе все того же приложения.

Исторически, в устройстве СУБД выделяли три уровня, предложенных ещё в 1975 году в отчёте ANSI/X3/SPARC [1,2,5,7]:

- *внешний уровень* наиболее близок к приложениям и пользователям, он связан со специфичными для них способами представления данных;
- *логический уровень*, также называемый концептуальным, для описаний данных, не зависящих от физической реализации;
- *внутренний уровень*, называемый также физическим, наиболее близок к физическому хранилищу данных, он связан со способами хранения на физических устройствах.

Внутренний уровень не всегда связан с файловой системой. Наиболее развитые современные СУБД представляют собой по сути специализированную операционную систему[3], способную управлять физическими устройствами хранения, кешем данных, процессами и потоками, оперативной памятью, асинхронным запуском и внутренним планировщиком задач минуя собственно операционную систему компьютера.

Вернёмся к примеру с CSV-файлом и приложением LibreOffice Calc, наглядно показывающему, что даже в примитивной СУБД все три упомянутых уровня можно чётко выделить:

- внутренний уровень представлен собственно файлом формата CVS со спецификацией его полного имени в файловой системе, кодовой страницы и символа разделителя;
- логический уровень представлен объектом «Электронная таблица» (spreadsheet), позволяющим описать данные в терминах листов, колонок и строк независимо от того, работаем ли мы с CSV-файлом или с файлом в другом формате, возможно даже двоичном;

[3] См. например информацию по SQLOS в составе Microsoft SQL Server

- внешний уровень не только позволяет пользователям проводить над данными специфичные операции путём создания встроенных программ на бейсик-подобном языке, но и разграничить доступ к ним на уровне видимости, чтения и записи.

Тем не менее, LibreOffice Calc является не СУБД, а приложением для работы с электронными таблицами. Сама по себе всякая электронная таблица уже является БД, но далеко не всякая БД является электронной таблицей. Поскольку целью книги не является обзор способов организации простейших БД и примитивных СУБД, то на этом примере мы пока остановимся и перейдём к более насущным вопросам.

Типы приложений: транзакционная и аналитическая обработка

Начиная разговор о транзакционной обработке, нельзя обойтись без соответствующего определения.

> *Транзакцией* в СУБД называется совокупность операций над данными, являющаяся неделимой (атомарной).

Исторически, транзакции пришли в СУБД из области финансов и бухгалтерии, где нельзя снять деньги с одного счета, не зачислив их на другой, а сама операция, проводка, так и называется — transaction. На этом хрестоматийном примере часто объясняют сам принцип транзакционной обработки. Другой пример, тоже родом откуда-то из 1950-60-х годов, из области пассажирских перевозок. Когда необходимо заказать билет для путешествия транзитом, транзакция состоит, как минимум, из двух операций: резервирования билета до пункта пересадки и резервирования билета до места назначения. В случае если одна из операций невозможна, отменяется вся транзакция целиком.

Любое приложение, реализующее аналогичные вышеописанным прикладные функции, требующие использования механизма транзакций, относится к типу *транзакционных*. Отличительные особенности таких приложений следующие.

- Обработка идёт **в режиме реального или приближенного к реальному времени**. Время отклика системы при запросе оператора не превышает единиц секунд.
- Запросы представляют собой **интенсивный поток коротких операций** по вставке, изменению и удалению **небольшого числа записей** в БД. Эти операции могут быть как одиночными транзакциями, так и объединяться в более крупные транзакции.

В реляционной СУБД любой оператор SQL является одиночной транзакцией по умолчанию. Некоторые реализации, например Firebird, по умолчанию открывают новую транзакцию при выполнении каждого оператора SQL, оставляя её «висеть» до выдачи команды завершения. Это так называемый режим неявных транзакций (implicit transaction). Для управления соответствующим поведением, СУБД обычно имеет опцию включения и отключения неявных транзакций, например, `set implicit_transaction on|off` в SQL Server.

Основными аббревиатурами, касающимися транзакций с которыми вам придётся сталкиваться будут OLTP (On-Line Transaction Processing) — собственно интерактивная транзакционная обработка, и ACID (Atomicity-Consistency-Isolation-Durability) — принципы неделимости, целостности, изолированности и надёжности. О них мы поговорим в дальнейшем, рассмотрев на примерах принципы изоляции.

В противоположность транзакционной, аналитическая обработка данных имеет другие отличительные признаки.

- Данные находятся **в режиме чтения**, за исключением моментов их обновления.
- Выборки представляют собой **одиночные тяжёлые запросы**: поиски и расчёты по множеству произвольных критериев могут охватывать значительную часть данных в базе.
- **Время отклика системы не регламентировано**, нередко пользователь имеет возможность прервать слишком долго

выполняющийся запрос, чтобы упростить его или разделить расчёты на более короткие этапы.

- Размеры базы данных, как правило, на порядок и больше превышают таковые для транзакционной.

Аналогичной аббревиатурой для **интерактивной** аналитической обработки является OLAP (On-Line Analytical Processing). Соответственно, интерактивное приложение, работающее с СУБД в режиме OLAP, относится к *аналитическим*.

Важное здесь слово «интерактивная». Действительно, в отличие от транзакционной, аналитическая обработка может вестись и в пакетном режиме: пользователь формулирует задачу в понятных приложению терминах, ставит обработку в очередь и, например, утром следующего дня получает результаты. Таковой, например, является заблаговременная ночная генерация регулярных отчётов, рассылаемых по утрам или с другой заданной периодичностью группам пользователей (см. например Microsoft SQL Server Reporting Services).

Необходимо понимать, почему возникло разделение на транзакционную и аналитическую обработку, ведь со стороны внешнего уровня архитектуры СУБД они не слишком отличаются. В обоих случаях запросы, да, немного разные, ну и что?

Основная проблема заключается в том, что СУБД, ориентированные на транзакционную обработку, менее эффективны при работе с аналитическими запросами и наоборот. Универсальных архитектур и их реализаций, одинаково легко справляющихся как с большим потоком мелких запросов и транзакций, так и с гораздо меньшим числом массивных тяжёлых выборок по большому диапазону, не существует. Пока, по крайней мере. Ещё сложнее реализовать оба типа обработки одновременно в одной и той же БД. Поэтому возникла и уже долгое время[4] существует упомянутая

[4] Первые тиражируемые системы аналитической обработки данных относятся к 1970 году (Express), но окончательно, как самостоятельное направление, OLAP оформилось только в начале 1990-х годов.

выше специализация как среди СУБД, так и среди программистов, администраторов и консультантов.

Клиент-серверные и встроенные СУБД

Технология *клиент-сервер* является основой большинства реализаций современных программных систем. Речь идёт не только о паре «Приложение конечного пользователя — СУБД». Каждое звено многозвенной системы [13] с длинными цепочками прохождения сообщений от пользовательского интерфейса через службы и серверы приложений к СУБД также работает в режиме «клиент-сервер». Отличительные признаки технологии следующие:

- два или более автономных процесса[5], исполняющихся на одном или нескольких вычислительных устройствах (компьютерах);
- процессы взаимодействуют посредством передачи сообщений согласно протоколу;
- протокол минимально состоит из запроса и ответа, запрашивающий процесс называется *клиентом*, отвечающий на запрос — *сервером*.

Роли клиента и сервера у процессов могут меняться. Например, сервер может время от времени опрашивать своих клиентов о состоянии. В производственной практике *серверным приложением* или просто *сервером* обычно называют тот процесс, основная задача которого состоит в обработке поступающих от клиентов запросов. В такой терминологии есть некоторый риск спутать приложение-сервер с компьютером-сервером, исполняющим роль физического устройства для эксплуатации тех самых серверных приложений. Поэтому однозначно понять смысл термина «сервер» можно только в контексте его употребления.

Из сказанного следует, что СУБД, работающая как самостоятельный процесс, также является сервером, СУБД-сервером. Однако, некоторые СУБД позволяют напрямую пристыковать себя к процессу-приложению,

[5] Имеется в виду понятие процесса не только в рамках многозадачной операционной системы, но и любого программного модуля на устройстве, например, на платёжном терминале для приёма банковских карт или на кассовом аппарате.

например, посредством динамических библиотек. В этом случае СУБД исполняется в том же процессе (in-process), что и приложение и называется *встроенной* или встраиваемой.

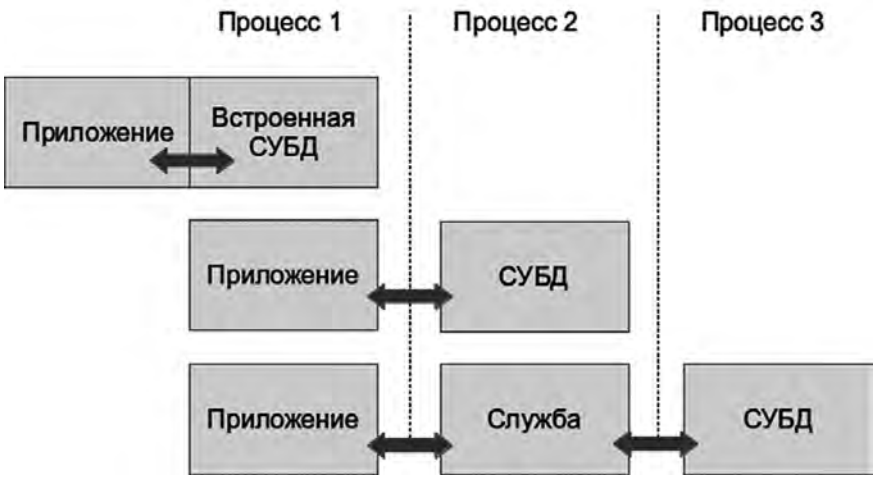

Рис. 1. Взаимодействие приложения с СУБД в системах с разным числом звеньев

Встроенные СУБД по сравнению с клиент-серверными имеют ряд преимуществ и недостатков. К преимуществам встроенных СУБД можно отнести следующие особенности.

1. Простота разработки приложений. Программисту не требуется установка и настройка СУБД на своём рабочем месте. Программа имеет полный контроль над логикой работы с БД.

2. Простота развёртывания приложений. В большинстве случаев достаточно скопировать файлы динамически подгружаемых библиотек в директорий приложения, при этом пользователю даже не нужно иметь права администратора на своём компьютере или другом устройстве, например, на планшете.

3. Простота обслуживания локальной базы данных. Как правило, приложение со встроенной СУБД эксплуатируется в однопользовательском режиме, а размер базы данных относительно невелик, поэтому необходимость в дополнительной настройке прав доступа и оптимизации быстродействия практически отсутствует. Регулярные операции обслуживания базы данных (резервное копирование, индексация, очистка, дефрагментация и т.п.) могут

быть автоматизированы и реализованы непосредственно приложением, например, при его запуске.

4. Возможность работы в однозадачной операционной системе. Хотя времена господства MS DOS на персональных компьютерах давно прошли, разработчики встраиваемых систем для различных устройств и оборудования смогут по достоинству оценить эту возможность.

5. Как правило, более высокое быстродействие на операциях вставки, удаления и считывания одиночных записей. Это связано не только с однопользовательским режимом работы, но и с отсутствием затрат на упаковку данных приложением с последующей их передачей по сети СУБД-серверу. По сути, приложение напрямую работает с локально расположенными файлами через слой программных интерфейсов (API) встроенной СУБД.

Следует отметить, что возможно также использование встроенной СУБД в многопользовательском режиме для работы с разделяемыми по сети файлами данных. Такая технология получила название «файл-сервер» и была весьма популярна до широкого распространения клиент-серверных СУБД. Интересующихся я отсылаю к системам разработки под общим названием xBase. Сюда относятся такие продукты, как dBase, Paradox, FoxPro или Clipper (современная открытая кросс-платформенная реализация Clipper называется Harbour). Тем не менее, как минимум одна популярная встраиваемая СУБД до сих пор нередко используется в таком качестве, это Microsoft Access и его «мотор» MS Jet, являющийся компонентом операционной системы Windows.

Кроме преимуществ, встроенные СУБД обладают рядом ограничений и недостатков, вытекающих непосредственно из технологии.

1. Более высокий риск потерь и повреждения данных. Как мы помним, встроенная СУБД находится в том же процессе операционной системы, что и приложение. Соответственно, аварийное завершение работы приложения в момент записи данных может повредить их. Транзакции также контролируются непосредственно приложением,

поэтому при аварийном останове некому будет откатить незавершённые модификации базы данных и она останется в несогласованном состоянии.

2. Ориентация на автономные однопользовательские клиентские и многопоточные серверные приложения. Хотя с помощью разделения файлов в сети можно организовать доступ к одной БД из нескольких приложений, находящихся на разных устройствах, такое решение будет ограничено небольшим числом пользователей и малыми размерами базы данных. Не касаясь серьёзных проблем обслуживания и обеспечения безопасности такой БД, на практике в сети передачи данных 100 мегабит/сек проблемы быстродействия начинаются уже при десятке пользователей, работающих с файлами размером нескольких сотен мегабайт данных.
3. Невозможность распределения вычислительной нагрузки. Встроенная СУБД работает на одном устройстве, а вычисления производит непосредственно приложение. Клиент-серверная СУБД может быть развёрнута в кластере из многих устройств, при этом она способна сама производить вычисления, возвращая приложению-клиенту результат.
4. Низкий уровень безопасности. Файлы базы данных должны быть целиком доступны пользователю как минимум на чтение. Таким образом, невозможно предотвратить копирование этих файлов злоумышленниками.
5. Как правило, функционал встроенной СУБД урезан по сравнению с клиент-серверной версией того же продукта. Например, встроенная версия Microsoft SQL Server Compact несовместима с клиент-серверными версиями и сравнима с Microsoft Access.

Конечно, перечисленных критериев недостаточно для полноценного выбора технологии и типа СУБД, но, надеюсь, у читателя появится базис для систематизации поиска согласно требованиям к разрабатываемой программной системе.

На практике иногда я встречал отношение к вопросам выбора СУБД на уровне высказывания: «Не будем ломать голову, возьмём Oracle: он, как танк, везде пройдёт и нам уж всяко подойдёт». Действительно, если вы не хотите поднапрячь голову сейчас, но согласны, что спустя некоторое время, возможно, придётся на этой же голове рвать волосы, то можно, не напрягаясь, попросить сделать за вас работу продавцов-консультантов продуктов «Большой тройки»[6].

Из ряда многочисленных встроенных СУБД, опробованных непосредственно на практике или в тестовом режиме, таких как xBase, Microsoft Access, Microsoft SQL Server Compact, SQLite, Oracle Lite) наиболее интересным современным решением мне представляется Firebird по следующим причинам.

1. Кросс-платформенная СУБД, доступны версии для Windows, Linux и Mac.
2. Свободно распространяемая СУБД с открытым исходным кодом, допускающая использование в закрытых коммерческих продуктах.
3. Всего один основной DLL-файл для развёртывания (могут также понадобиться файлы ресурсов), одновременно работающий и как встроенная СУБД, и как клиентская библиотека для связи с СУБД-сервером Firebird. Соответствующий режим выбирается указанием или пропуском имени сервера в строке соединения.
4. Практически полностью поддержана функциональность своей клиент-серверной версии, включая поддержку встроенных типов данных, видов, триггеров, хранимых процедур и системы безопасности. Это значит, что разработанное программистом приложение будет иметь минимум специфики при работе со встроенной или клиент-серверной версией СУБД с возможностью переключения изменением всего одного параметра соединения.
5. Версионность данных, благодаря которой читающие транзакции не блокируют пишущие транзакции и наоборот. Разработчики отчётов

[6] На данный момент и уже в течение многих лет Big-3 поставщиков СУБД на мировом рынке составляют корпорации Oracle, IBM и Microsoft.

оперативного контура информационных систем оценят эту возможность, когда требуется выдача согласованной информации на текущий момент времени.

Конечно, Firebird не лишён недостатков, особенно с точки зрения администратора баз данных, но для автономных приложений эта СУБД предоставляет разработчику заманчивые возможности. В дополнение к теме я приведу выдержки из своего небольшого обзора состояния Firebird, написанного в 2013 году с позиций как администратора БД, которому необходимо обеспечивать промышленную эксплуатацию баз данных под Firebird, так и инженера, постоянно решающего вопросы поддержки работоспособности продукта среди многих клиентов, эксплуатирующих разные версии СУБД.

Сноска. Firebird 2.5: состояние

Firebird достаточно лёгок в администрировании: для относительно небольших баз данных и приложений СУБД практически не требует вмешательства администратора БД. Для разработчиков автономных приложений это важный фактор, позволяющий минимизировать затраты на поддержку приложений у клиентов, не предоставляющих удалённый доступ к своей сети или у которых не имеется своего администратора БД в штате службы ИТ, если она вообще есть.

Однако, если эксплуатируемых баз данных, приложений и пользователей в хозяйстве становится все больше, то прежняя лёгкость начинает превращаться в легковесность.

Особенности реализации требуют, чтобы операция создания внешнего ключа между двумя таблицами проводилась только при наличии одного единственного соединения к БД. Сами же таблицы создать можно и в обычном соединении, имея соответствующие права. Просуществовало это ограничение до версии 2.1, но у многих клиентов в эксплуатации находится ещё версия 1.5.

Штатных средств соединения в однопользовательском режиме нет. Поддержка предлагает утилитой `gfix` отключить базу данных в

эксплуатации (!), потом включить её в режиме sysdba, что, впрочем, тоже не гарантирует единственного соединения, если более одного сотрудника имеют доступ с уровнем sysdba.

Редакций Firebird по-прежнему несколько, к SuperServer и Classic добавилась SuperClassic. Исторически, Interbase, на основе открытого кода которого возник Firebird, использовал classic-архитектуру: на каждое соединение СУБД стартует отдельный серверный процесс. Такой подход обеспечивает достаточно высокую надёжность (если один процесс аварийно завершается, то на работу остальных это не влияет) и параллелизм средствами операционной системы, но отсутствует разделяемый кэш данных в оперативной памяти. В SuperServer серверный процесс один, каждое соединение работает в своём потоке с разделяемым кэшем. Но SuperServer не умеет распараллеливать запросы: пока один запрос выполняется со 100% загрузкой одного ядра (одного процессора), остальные ждут. Ситуацию призвана исправить появившаяся редакция SuperClassic.

По большому счёту, «суперклассик» - это все тот же «классик», упакованный в один процесс: потоки внутри по-прежнему максимально изолированы и не разделяют кэш, поэтому риск сбоя или отказа СУБД-сервера снижается. Но один поток не соответствует соединению, имеется их пул, использующий потоки по мере необходимости, поэтому «суперклассик» будет несколько производительнее и рачительнее к оперативной памяти. Администратору также удобнее наблюдать и поддерживать в работоспособном состоянии один процесс, чем множество.

Собственно, кэш данных у Firebird весьма условный, СУБД использует кэш файлов операционной системы, поэтому сбросить его для проверки производительности «холодных» запросов[7] невозможно даже перезапустив СУБД. К примеру, в SQL Server для этого есть специальная команда.

Средства администрирования, прежде всего аудита и мониторинга из комплекта достаточно примитивны. Например, трассировщик запросов,

[7] Запрос к СУБД, требующий преимущественно операции с долговременной памятью (дисковой памятью)

образцом которого вполне может служить MS SQL Server Profiler, являет собой утилиту командной строки, запускаемую со своим конфигурационным файлом. Последующую расшифровку полученных результатов надо проводить вручную при помощи регулярных выражений. Есть коммерческие утилиты, предоставляющие более дружественный интерфейс. В версии 2.5 появились возможности накопления статистики в системных таблицах мониторинга вида MON$XXX.

Средством «полной очистки» БД от так называемого мусора до сих пор является процедура резервного копирования и последующего за ним восстановления при помощи утилиты gbak. Однако, подобная процедура означает отключение пользователей от СУБД на всё время восстановления, делая проблематичной работу в режиме 24x7. Файл базы данных понемногу растёт даже при нулевом балансе добавленных и удалённых записей с постоянной фиксацией транзакций, данные дефрагментируются, что может снизить производительность. Ощущается нехватка аналога команды VACUUM, имеющейся в СУБД PostgreSQL.

Firebird не блокирует используемые файлы. Таким образом файл БД можно скопировать на сервере прямо во время многопользовательской работы. Как оказалось, некоторые клиенты именно так и поступают, называя подобное непотребство «резервным копированием». Проблема заключается в том, что во время копирования нет никакой гарантии нахождения файла в целостном состоянии, как и отсутствия незавершённых транзакций. Наверное, все-таки лучше было бы предусмотреть блокирование, как дополнительную защиту «от дурака».

Наличие перечисленных проблем не позволяет поднять усреднённую планку использования выше чем уровень «СУБД рабочих групп и небольших предприятий», означающий для систем транзакционной обработки десятки пользователей при объёмах БД в десятки гигабайт. Но, вполне возможно, что и такого уровня будет достаточно для ваших приложений.

С другой стороны, с точки зрения программиста, Firebird представляет собой удобный СУБД-сервер: простой в установке, нетребовательный к

ресурсам вычислительного устройства, переносимый между различными операционными системами, встраиваемый в приложение. Собственный язык программирования позволяет реализовать логику в хранимых процедурах и триггерах. Имеется большое число средств проектирования, разработки и отладки: от свободно распространяемых с открытым кодом до коммерческих. Стандартные средства сетевого доступа к СУБД имеются практически для всех платформ разработки от массово-корпоративной Явы до более экзотических вроде Руби.

Основные модели данных: иерархическая, сетевая, реляционная

Для реализаций универсальных СУБД в течение многих лет и по настоящий момент доминирует реляционная модель. Несмотря на то, что вы, возможно, и не столкнётесь с СУБД иных типов, нелишним будет знать о существовании других миров. Например, чтобы понимать, соответствуют ли реалиям заявления продавцов о «новых подходах и концепциях».

Иерархическая модель

Исторически, первой моделью данных, то есть способом их организации, структурирования, доступа и манипуляции, была *иерархическая модель*. Сейчас уже трудно найти тому причину, вероятнее всего сыграла свою роль тесная связь традиционного программирования с метафорой деревьев — упрощённого типа графов. С другой стороны, человеческому мозгу, справляющемуся со сложностями окружающего мира путём выделения иерархий, работать с ними оказывается проще, чем с более абстрактными множествами.

Продукт IMS (Information Management System) фирмы IBM, считающийся первой промышленной СУБД, реализует именно иерархическую модель. Разработанная в 1966 году, эта СУБД до сих пор эксплуатируется на новейших мэйнфреймах серии Z, обеспечивая высокую производительность обработки порядка сотни тысяч транзакций в секунду.

Современной массово доступной каждому программисту реализацией иерархической модели данных является XML, точнее, разнообразные

«движки» и API для манипуляции со структурами, созданными на основе этой технологии с обязательным применением схем определения данных.

Последний пункт важен: без использования XML-схемы (XML schema) или описания типов DTD (Data Type Definition) документ XML относится к неполно структурированным моделям данных, рассматриваемым в следующей главе.

Если взглянуть на структуру XML-документа, то иерархия элементов данных становится видна, что называется, невооружённым глазом.

```
<?xml version="1.0"?>
<sales>
  <orders>
    <order>
      <num>S01</num>
      <date>2014-02-20</date>
      <client>
        <name>Пирожки ООО</name>
        <address>ул. Благодатная 235</address>
        <phone>322-223-322</phone>
      </client>
      <items>
        <product>
          <name>Мука</name>
          <quantity>50</quantity>
          <units>кг</units>
          <price>45</price>
        </product>
        <product>
          <name>Дрожжи</name>
          <quantity>300</quantity>
          <units>г</units>
          <price>80</price>
        </product>
      </items>
    </order>
    <order>
      <num>S02</num>
      <date>2014-02-21</date>
      <client>
        <name>Пирожки ООО</name>
        <address>ул. Благодатная 235</address>
        <phone>322-223-322</phone>
```

```
      </client>
      <items>
        <product>
          <name>Мука</name>
          <quantity>70</quantity>
          <units>кг</units>
          <price>40</price>
        </product>
        <product>
          <name>Дрожжи</name>
          <quantity>500</quantity>
          <units>г</units>
          <price>75</price>
        </product>
      </items>
    </order>
  </orders>
</sales>
```

В приведённом примере БД описания продаж имеет иерархическую структуру. Все описываемые сущности являются **вложенными** в сущность более высокого уровня организации, что вызывает определённые неудобства хранения: приходится дублировать информацию.

Преимуществом такой структуры является возможность выполнения поисковых запросов без соединений. Например, чтобы выбрать все заказы клиента «Пирожки ООО», в которых он покупал муку в количестве более 50 кг, язык XPath[8] позволяет сделать в программе следующее:

```
var nodes = xmlDoc.SelectNodes(
  "/sales/orders/order[client/name='Пирожки ООО' and
items/product[name = 'Мука']/quantity > 50]");

foreach(node in nodes) {
  ...
}
```

Однако, с точки зрения расширяемости и поддержки целостности такая структура малопригодна. В рамках реляционной модели мы бы констатировали, что она денормализована, о чем будет сказано позже в разделе, посвящённом проектированию. Например, при добавлении оплат

[8] XPath (от англ. XML Path Language) — язык запросов к элементам XML-документа.

потребуется создавать новый тег описаний <payment> и включать их в состав заказов (orders). Однако, один платёж может покрывать более одного заказа и наоборот. Если же у клиента изменился адрес или телефон, то необходимо сделать соответствующие изменения во всех тегах <client>.

Частично, эта проблема решается введением ссылок на элементы XML-документа.

```
<?xml version="1.0"?>
<sales>
  <clients>
    <client>
      <id>1</id>
      <name>Пирожки ООО</name>
      <address>ул. Благодатная 235</address>
      <phone>322-223-322</phone>
    </client>
  </clients>
  <orders>
    <order>
      <num>S01</num>
      <date>2014-02-20</date>
      <id_client>1</id_client>
      <items>
        <item>
          <id_product>1</id_product>
          <quantity>50</quantity>
          <units>кг</units>
          <price>45</price>
        </item>
        <item>
          <id_product>2</id_product>
          <quantity>300</quantity>
          <units>г</units>
          <price>80</price>
        </item>
      </items>
    </order>
    <order>
      <num>S02</num>
      <date>2014-02-21</date>
      <id_client>1</id_client>
      <items>
        <item>
```

```
          <id_product>1</id_product>
          <quantity>70</quantity>
          <units>кг</units>
          <price>40</price>
        </item>
        <item>
          <id_product>2</id_product>
          <quantity>500</quantity>
          <units>г</units>
          <price>75</price>
        </item>
      </items>
    </order>
  </orders>
  <products>
    <product>
      <id>1</id>
      <name>Мука</name>
    </product>
    <product>
      <id>2</id>
      <name>Дрожжи</name>
    </product>
  </products>
</sales>
```

Структура становится гибче, мы исключаем дублирование. Но теперь наш первоначальный запрос значительно усложнится, так как приходится делать соединения между элементами.

```
var nodes = xmlDoc.SelectNodes(
  "/sales/orders/order[id_client = /sales/clients/client[name
= 'Пирожки ООО']/id and items/item[id_product =
/sales/products/product[name = 'Мука']/id and quantity >
50]]");

foreach(node in nodes) {
  ...
}
```

На XML-документах небольшого размера с сотнями элементов подобные запросы выполняются быстро, но с ростом объёма данных неизбежно возникает необходимость их индексировать. В СУБД, поддерживающих XML хотя бы на уровне типа данных колонки таблицы, имеется

возможность такой индексации. Например, в MS SQL Server вначале строится так называемый первичный XML-индекс, затем несколько вторичных, для которых нужно указывать их специализацию: PATH для запросов типа проверки существования элементов, PROPERTY — для извлечения значений элементов и атрибутов или VALUE — для сквозного поиска элементов по значению. Без достаточного понимания таких особенностей физической организации данных хранилище XML начнёт испытывать проблемы с производительностью уже на небольших объёмах.

Кроме XML и поддерживающих его СУБД существуют и другие реализации. Наиболее известная отечественная разработка — СУБД ИНЕС [16], являвшаяся фактически стандартом на советских ЭВМ серии ЕС. К сожалению, эволюция системы прервалась вместе с прекращением выпуска своих ЭВМ, до наших дней этот продукт не дожил.

К иерархическим также можно отнести СУБД на базе подхода MUMPS[9] (в СССР назывался «ДИАМС»), лежащий в основе линейки продуктов InterSystems: от выпущенной в 1970-х годах СУБД ISM (InterSystem M), через OpenM периода 1980-х к СУБД Cache с конца 1990-х и по наше время, реализующей современные подходы к организации иерархий [22].

Так, например, в MUMPS можно обращаться к значениям полей:

```
SET ^Автомобиль("Корпус","Дверь","Цвет")="Синий"
```

Здесь сущность «Автомобиль» представлена в виде дерева, одним из верхних узлов которого является элемент «Корпус», в состав которого, в свою очередь, входит «Дверь», имеющая пока ещё листовой узел «Цвет».

Иерархию несложно динамически расширять как в ширину:

```
SET ^Автомобиль("Корпус","Крышка капота")=2
```

так и в глубину:

```
SET ^Автомобиль
      ("Корпус","Дверь","Цвет","Отражение")="Матовый"
```

[9] MUMPS — Massachusetts General Hospital Utility Multi-Programming System, позднее Multi-User Multi-Programming System, разработка датирована 1966 годом.

Общий принцип распознавания лежащей в основе СУБД иерархической модели можно сформулировать так.

> Если при использовании входного языка и/или API наиболее низкого уровня из доступных программисту для доступа к данным (значениям узлов, переменных, полей и т. д.) **требуется указывать некоторый путь**, то лежащая в основе СУБД модель базируется на иерархиях.

Иерархическая модель: плюсы и минусы

К преимуществам иерархической модели можно отнести относительную простоту восприятия логической структуры базы данных человеком. Система обеспечивает высокое быстродействие при транзакционной обработке, когда номенклатура типов запросов фиксирована.

При переходе к сложным многосвязным структурам и произвольным запросам начинают всплывать недостатки модели. Прежде всего, это медленный доступ к данным нижних уровней иерархии, что нетрудно заметить на примерах запросов с XML и в особенностях реализации индексов СУБД, поддерживающих XML в качестве встроенного типа. В тех же примерах видна и чёткая ориентация структур на определённые типы запросов, что исключает универсальность использования одной и той же БД разными типами приложений.

С теоретической точки зрения, используемая иерархическими СУБД модель графов ограничена деревьями, что сужает область её применения. Если структура связей сложна, то попытка втиснуть эту сложность в прокрустово ложе иерархий рождает на свет громоздкие описания, трудные для понимания специалистами и пользователями.

Сетевая модель

Сетевая модель, также относящаяся к теоретико-графовым, явилась развитием подходов, реализованных в модели иерархической. Прежде всего, развитие касается связей между записями, имеющих двунаправленный характер. Сетевую модель можно представить в виде графа с узлами в виде записей, и рёбрами, отображающими наборы. Направление и характер связи в сетевых БД не являются очевидными, как в

иерархических БД, поэтому характеристики и направление связей должны указываться явно при описании БД.

В 1975 году конференция CODASYL (Conference of Data System Languages) стандартизовала базовые понятия и формальный язык описания. Широко известной системой, основанной на сетевой модели данных, являлась СУБД IDMS (Integrated Database Management System), использовавшаяся на мэйнфреймах IBM. В настоящее время IDMS принадлежит компании Computer Associates, имеющей неформальный статус «лавки старьёвщика» в мире софтостроения: как правило, все купленные компанией продукты берутся на сопровождение, но не развиваются, доживая до своего естественного конца.

Несмотря на некоторые интересные особенности и преимущества, до наших дней дожило только небольшое число СУБД, реализующих сетевую модель, например американская Raima (бывшая dbVista) и отечественная КроносПро. На их примере также можно проследить эволюцию программных продуктов класса сетевых СУБД.

СУБД Raima изначально использовалась, как встроенная с достаточно низкоуровневым API для языка Си. Постепенно, в систему был добавлен интерфейс доступа на SQL, а сама СУБД получила возможность работы в режиме клиент-сервер. По-прежнему Raima используется для транзакционных приложений как лёгкое (lightweight) кросс-платформенное решение.

Напротив, развитие СУБД Кронос пошло в сторону аналитической обработки. Это та область, где преимущества сетевой модели могут быть использованы полнее, а недостатки обойдены более просто. Для объяснений нам все же придётся кратко познакомиться с основными понятиями сетевой модели.

> Термин *запись* соответствует аналогичному понятию структурного типа в традиционных языках программирования: record в Паскаль-подобных или struct в наследниках Си.

Набор данных служит для связывания двух типов записей отношением «один-ко-многим».

Связи на основе наборов являются по сути физическими ссылками. Это означает, что можно установить связи между двумя любыми экземплярами записей, если у нас имеются их указатели. Не требуется, как в реляционной модели, думать о первичных и внешних ключах уже на стадии логического проектирования. С помощью двух взаимно-обратных наборов реализуется связь «многие-ко-многим».

В СУБД Кронос также реализовано объединение наборов, позволяющее одной записи ссылаться на записи двух и более типов. Такая возможность очень удобна при моделировании. Например, если имеется тип записи «Адрес», который используется записями «Лица» и «Организации», то объединённый набор «Относится к лицу или организации» отобразит все возможные связи.

Реализация связей между записями на базе физических ссылок является более низкоуровневым механизмом, чем ссылочная целостность на основе ключей в реляционной модели, поскольку связи между конкретными экземплярами необходимо задавать явным образом. Но по этой же причине операции соединения имеют более высокое быстродействие.

Поясню на упрощённом примере. Связь между двумя таблицами в реляционной модели осуществляется по значениям, соответственно, первичного и внешних ключей. Для выбранной строки подчинённой таблицы берётся значение внешнего ключа, которое затем ищется среди значений первичного ключа главной таблицы. Если таблица отсортирована в физическом порядке следования значений первичного ключа (так называемый *кластерный ключ*, подробнее см. «Физическая организация памяти»), то строка данных находится в той же области памяти, что и найденное значение. Если нет, то происходит дополнительный поиск связанной со значением первичного ключа строки по её идентификатору (row id).

Рис. 2. Организация связей между типами записей в сетевой модели

В случае сетевой модели запись одного типа имеет одну или несколько ссылок на записи другого типа; по этой ссылке непосредственно извлекается связанная запись.

Однако, если связи между записями меняются часто, то возрастают накладные расходы на удаление прежних и создание новых ссылок в наборе, что может явиться недостатком на пути достижения требуемого быстродействия.

Действительно, в реляционной модели для модификации связи достаточно изменить значение внешнего ключа или просто обнулить его для удаления связи. Последующая проверка ссылочной целостности

сводится лишь к поиску соответствующего значения первичного ключа, но и её можно принудительно отключать, например, в случае массированной загрузки данных.

Другим недостатком сетевой СУБД является жёсткость задаваемых структур, и, соответственно, более высокая трудоёмкость при реструктурировании схемы БД. Добавление новых типов записей и полей не представляет собой большой проблемы, но изменение связей и разукрупнение записей требуют множества низкоуровневых операций, связанных с физической реорганизацией хранилища.

Было отмечено, что одной из особенностей аналитической обработки является нахождение базы данных в режиме чтения, изменения связей между записями редки и происходят, как правило, только в моменты обновления информации. Поэтому использование сетевой модели в аналитических приложениях может нивелировать влияние указанных недостатков.

Сетевая модель: плюсы и минусы

К преимуществам сетевой модели можно отнести следующие особенности:

- **стандартизация**. Стандарт CODASYL определяет базовые понятия модели и формальный язык описания.
- **быстродействие** сетевых БД данных сравнимо с таковым для иерархических БД.
- Полное использование теоретико-графовых моделей предоставляет **высокий уровень абстракции** описания предметных областей, не ограниченных иерархиями.
- **гибкость доступа к данным** через любую последовательность связанных записей, а не через их иерархию.

О недостатках уже было упомянуто:

- **жёсткость** задаваемых структур и сложность реструктуризации схем БД.

- **сложная** структура управления памятью в транзакционных приложениях с частыми изменениями связей.

Реляционная модель

Теоретические основы реляционной модели многократно и полно изложены в различных монографиях, книгах и статьях [1,2,4,11], поэтому в рамках данной главы ограничимся лишь минимальными описаниями. В дальнейшем мы будем расширять их, опираясь на практическую нужду.

Как и положено доминирующей на протяжении десятков лет массово внедрённой парадигме, реляционная модель имеет свой основополагающий миф. Речь, конечно, идёт об известной работе Эдгара Кодда «Реляционная модель данных для больших совместно используемых банков данных»[17], опубликованной в CACM[10] в 1970 году. «Да будет свет!» — сказал Кодд, и появился свет, и начали рождаться реляционные СУБД.

На самом деле, Кодд опирался на существовавшую и проработанную в рамках математической логики ещё в 19 веке теорию отношений [2,12]. В рамках этой теории было показано, что множество отношений замкнуто относительно некоторых специальных операций, то есть образует вместе с этими операциями абстрактную алгебру. В упрощённой формулировке это означает, что **все возможные результаты операций над элементами множества отношений также являются отношениями**.

Кроме того, Кодд начал свои публикации раньше, в 1969 году, исследовательским отчётом «Derivability, Redundency, and Consistency of Relations Stored in Large Data Banks» в среде ограниченного круга подписчиков. В дальнейшем концепция эволюционировала, известный эксперт в области СУБД Майкл Стоунбрейкер отмечал [18], что «можно видеть четыре разных версии» модели:

- версия 1, та самая упомянутая, составляющая основу отраслевого мифа, опубликована в статье в CACM в 1970 г.;

[10] CACM - Communications of the ACM, ежемесячный журнал сообщества Association for Computing Machinery (ACM), основанный в 1957 году.

- версия 2, определена в статье по поводу Тьюринговской премии в 1981 г.;
- версия 3, доопределена «дюжиной Кодда» — двенадцатью правилами и оценочной системой в 1985 году [19,20];
- версия 4, определена в опубликованной в 1990 году книге Кодда [21].

Из сказанного Стоунбрейкером становится понятным, что Кодд, как и положено учёному, в течение десятилетий продолжал исследования, пересматривая свои подходы и концепции. И полнота реализации теории в промышленных СУБД волновала её автора в меньшей степени, потому что «дюжина Кодда» до сих пор является актуальной для оценки соответствия, а последние стандарты SQL полностью не реализованы ни в одном продукте.

Таким образом, начиная изучать предмет данной главы, необходимо принять к сведению, что **существуют два мира с условными названиями «реляционная модель» и «промышленные РСУБД», и эти два мира пересекаются, но не полностью**, создавая, в частности, проблемы переносимости приложений между СУБД разных производителей.

Реляционная модель, как и другая, более известная программистам — объектная, относится к логическим. Определённые в рамках модели структуры, операции над ними и задаваемые ограничения не зависят от способов реализации физической организации данных и управления ими. Тем не менее, каждая СУБД имеет свои особенности физической организации, поэтому одна и та же схема данных в рамках реляционной модели может иметь специфические параметры и опции, оптимизирующие работу с данными. Например, соответствующая понятию «отношение» таблица может быть организована в виде кластера, сегмента памяти (кучи), материализованной проекции, секционированной таблицы. К такого рода неоднозначности мы вернёмся в разделе, посвящённом проектированию.

Что же представляет собой отношение с более формальной точки зрения? Обратимся к первоисточнику [17].

Термин *отношение* используется в его общепринятом математическом смысле. Для заданных множеств S_1, S_2, ..., S_n (не обязательно различных) **R** является отношением на этих n множествах, если представляет собой множество кортежей степени n, у каждого из которых первый элемент взят из множества S1, второй — из множества S2 и т.д. Мы будем называть S_j j-тым доменом R. Говорят, что такое отношение R имеет степень n. Отношения степени 1 часто называют унарными, степени 2 — бинарными, степени 3 — тернарными и степени n — n-арными.

Для лучшего понимания столь лаконичного определения воспользуемся его графической интерпретацией.

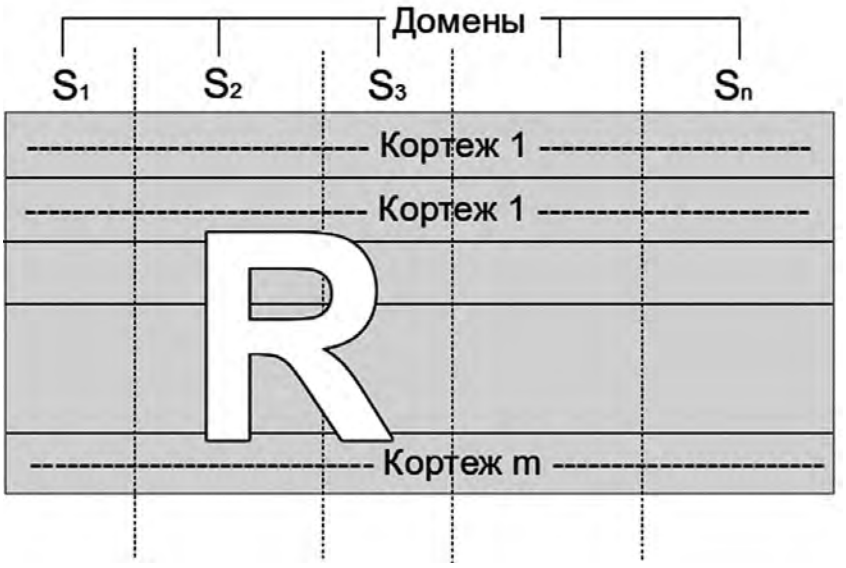

Рис. 3. Отношение, как множество кортежей, заданных на доменах

Геометрический смысл данного выше определения больше всего напоминает таблицу с зафиксированным типом данных для каждой колонки. По этой причине реляционные СУБД предлагают своим пользователям оперировать не определениями реляционной алгебры, а их адаптированными аналогами (таблицы, строки, столбцы и т. п.). Хотя эти аналоги имеют более низкий уровень абстракции, они легче представляются в головах среднестатистических специалистов, занятых в основном насущными производственными проблемами, а не задачами из курса теории множеств.

Реляционная модель: плюсы и минусы

К преимуществам реляционной модели можно отнести следующие особенности:

- **теоретическая основа.** Формально определяет базовые понятия модели, язык описания и операции над отношениями;
- **стандартизация**. Стандарты SQL-NN (SQL-89, SQL-92, SQL-99 и т. д.), имеющие несколько уровней полноты реализации, позволяют создавать приложения, переносимые между СУБД разных поставщиков;
- полное **разделение** доступа к данным от способа их физической организации;
- **универсальность**. Информационное моделирование сущностей реального мира в виде набора связанных таблиц является достаточно хорошим подходом в большинстве случаев;
- **простота** манипуляции данными с точки зрения конечного пользователя;
- **SQL** — развитый стандартизованный декларативный язык 4-го поколения.

Недостатки:

- в общем случае, **более низкое быстродействие** по сравнению с сетевыми и иерархическими СУБД или другими подходами, обеспечивающими доступ к данным непосредственно на уровне их физической организации, например, индексированные файлы;
- **неполнота реализации** стандартов SQL-NN, а также специфические языковые и процедурные расширения СУБД разных поставщиков, осложняющие переносимость приложений (так называемый vendor lock);
- необходимость учёта некоторых **особенностей модели на концептуальном уровне** (ключи — идентификаторы сущностей), отсутствующая, например, в сетевой модели.

Другие подходы и модели данных

Модель «Сущность-атрибут-значение» (EAV)

В англоязычной терминологии модель САЗ (Сущность-Атрибут-Значение) носит название EAV (Entity-Atribute-Value). В русскоязычном сообществе разработчиков приложений баз данных подход обсуждался в 1990-х годах как «вертикальное хранение атрибутов», в противовес реляционной модели, где атрибуты располагаются горизонтально.

Суть подхода состоит в максимальном упрощении структуры хранения данных, что обеспечивает высокую гибкость изменения логической схемы данных. Вертикальное хранилище атрибутов представляет собой похожую на таблицу структуру, состоящую из трёх столбцов:

- столбец «Сущность». Содержит уникальный идентификатор сущности. Также может представлять собой пару (идентификатор, временная метка) для отражения хронологии событий и фактов;
- столбец «Атрибут». Название свойства сущности, как правило — ссылка (короткий идентификатор) на таблицу атрибутов в метаданных;
- столбец «Значение». Непосредственно значение атрибута.

Использование реляционной терминологии «таблица» и «столбец» для описания модели САЗ вызвана, во-первых, стереотипом наиболее понятным для большинства разработчиков, во-вторых, тем фактом, что вертикальное хранение в большинстве известных автору случаев было реализовано средствами реляционной СУБД.

На самом деле не столь важно, сгруппированы ли тройки САЗ в длинную таблицу, или же речь идёт о записях, объединённых в двунаправленный список. Все они суть способы представления **разреженной матрицы САЗ**, столбцы которой представляют собой все определённые в системе атрибуты, строки — все имеющиеся сущности, а на пересечении строки и столбца находится соответствующее значение, если оно имеется.

Товар №795	Наименование	Мука пшеничная в/с
Товар №795	Вес упаковки	50
Клиент №123	Наименование	"Пирожки" ООО
Доставка №24	Дата создания	2014-06-15
Доставка №24	Статус	В пути
Продажа №867	Дата создания	2014-06-12
...	...	...

Рис. 4. Представление САЗ в виде таблицы

	Наименование	Вес упаковки	Дата создания	Статус
Товар №795	Мука пшеничная в/с	50		
Клиент №123	"Пирожки" ООО			
Доставка №24			2014-06-15	В пути
Продажа №867			2014-06-12	

Рис. 5. Матрица САЗ является разреженной

Немного углубясь в историю, можно обнаружить гораздо более древний и широко известный в классическом программировании подход ассоциативных массивов, состоящих из пар «ключ-значение». Этот путь прямиком приводит нас в 1960-е годы к языку Лисп. Базы данных, использующие такой поход, применялись ещё до эпохи массового распространения реляционных СУБД. Только и разница, что теперь вместо простого ключа используется составной. Записывали раньше в массив пару («Товар№795/Наименование», «Мука пшеничная в/с»), теперь же ключом ассоциативного списка становится пара атрибутов, а в программе оперируем уже тройками.

Развивая подход, получаем современные многомерные кубы, где ключом является набор величин, а значением — некоторая агрегированная величина. К кубам мы ещё вернёмся в следующих главах.

Обратимся к практической стороне использования модели. САЗ даёт возможность менять логическую схему данных, что называется, «на лету». Действительно, достаточно добавить в метаданные описание нового атрибута и связать его с типами сущностей, как все приложения могут считывать и записывать значения данного атрибута, не принимая во внимание структурные изменения. Для сравнения, в реляционной БД добавление колонки в таблицу вызовет изменение структуры соответствующих наборов данных на стороне приложения по горизонтали, в противном случае колонка останется невидимой.

Интересную возможность представляют запросы по значениям атрибутов вне учёта сущностей. Например, следующий простой запрос выведет нам все сущности, у которых, во-первых, имеется атрибут «вес», а, во-вторых, этот вес находится в заданных рамках.

```
SELECT *
FROM eav
WHERE attribite = 'Вес' AND value BETWEEN 10 AND 50
```

В рамках корпоративной БД, в результатах такого запроса можно будет обнаружить не только упакованные товары, но и разнообразные весовые коэффициенты, отгрузки, приёмки, характеристики сотрудников, параметры оборудования, рецептуры технологий и многое другое. Неплохой инструмент для исследования семантики.

Однако, такие запросы без учёта контекста, например, типа сущности, достаточно редки. Для отображения связей приходится вводить специальные атрибуты, ссылочная целостность по значениям которых без типизации сущностей не реализуется в принципе, а при наличии типизации реализуется с высокими накладными расходами.

Расширенная модель САЗ носит название EAV/CR (EAV with classes and relationships), то есть САЗ с классами (типами) и связями. В дальнейшем будем называть её РСАЗ.

РСАЗ явным образом поддерживает для каждой сущности понятие класса (типа). Это означает, что запросы теперь можно ограничивать только сущностями заданного типа, чтобы избежать приведённой выше неоднозначности выборки по значениям атрибутов.

Связи между сущностями также поддерживаются в явном виде, кроме того структура, задающая связи, подлежит унификации.

Ограничения целостности могут быть описаны на уровне метаданных, но способ их реализации не оговорён. Например, в реляционной БД таким механизмом могут являться триггеры таблиц метаданных.

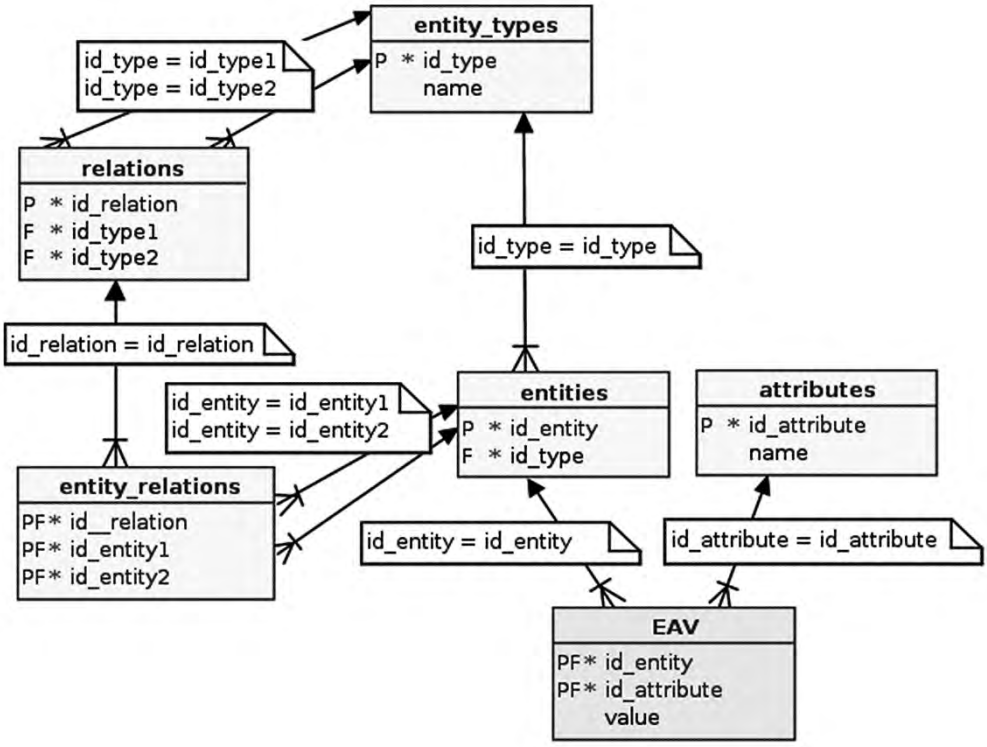

Рис. 6. Пример реализации расширенной модели САЗ

Запросы, относящиеся к категории CRUD[11], также не представляют особенных трудностей, за исключением необходимости транспонировать результат в табличную форму. В рамках реализующего САЗ реляционного

[11] CRUD — от англ. Create-Retreive-Update-Delete, типы множественных запросов на создание, считывание, модификацию или удаление единичного экземпляра сущности.

подхода такие макрозапросы состоят из большого числа мелких подзапросов в соответствии с числом выводимых колонок-атрибутов.

```
SELECT
  (SELECT value
   FROM eav
   WHERE entity = 'Товар795'
         AND attribute = 'Наименование'
  ) AS "Наименование",
  (SELECT value
   FROM eav
   WHERE entity = 'Товар795'
         AND attribute = 'Вес упаковки'
  ) AS "Вес упаковки"
```

Для сравнения, запрос в рамках реляционной БД выглядит куда более лаконичным.

```
SELECT name AS "Наименование",
       weight AS "Вес упаковки"
FROM goods
WHERE name = 'Товар795'
```

Наибольшую сложность представляют собой запросы ad hoc[12], запросы с агрегацией и запросы с большим числом связей между сущностями. Текст становится неудобочитаемым, а эффективность выполнения реляционной СУБД вызывает сомнения без дополнительной оптимизации физического хранения и прямых подсказок оптимизатору со стороны программиста.

Следующий запрос выводит все номера доставленных заказов, в которых имеется товар «Мука пшеничная в/с», и его вес превышает 100 кг.

```
SELECT num AS "Номер заказа",
       SUM(amount) AS "Количество"
FROM
  (SELECT orders.value AS num, order_items2.value AS amount
   FROM eav orders
        INNER JOIN entities e1
          ON orders.id_entity = e1.id_entity
        INNER JOIN entity_types et1 ON e1.id_type =
et1.id_type
        INNER JOIN attributes a1
          ON a1.id_attribute = orders.id_attribute
```

[12] «по месту», в контексте СУБД — произвольные запросы пользователей

```
        INNER JOIN eav orders2
          ON orders.id_entity = orders2.id_entity
        INNER JOIN attributes a2
          ON a2.id_attribute = orders2.id_attribute
        INNER JOIN entity_relations er
          ON er.id_entity1 = orders.id_entity
        INNER JOIN eav order_items
          ON er.id_entity2 = order_items.id_entity
        INNER JOIN entities e3
          ON order_items.id_entity = e3.id_entity
        INNER JOIN entity_types et3 ON e3.id_type =
et3.id_type
        INNER JOIN attributes a3
          ON a3.id_attribute = order_items.id_attribute
        INNER JOIN eav order_items2
          ON order_items2.id_entity = order_items.id_entity
        INNER JOIN attributes a4
          ON a4.id_attribute = order_items2.id_attribute
   WHERE et1.name = 'Заказ' AND
         a1.name = 'Номер' AND
         a2.name = 'Статус' AND
         a2.value = 'Доставлен'
         et3.name = 'Предмет заказа' AND
         a3.name = 'Название' AND
         a3.value = 'Мука пшеничная в/с'
         a4.name = 'Вес' AND
         a4.value > 100 AND
  ) AS selected_orders
GROUP BY num
```

Несмотря на относительную простоту первоначального запроса, при реализации РСАЗ в рамках реляционной БД необходимо определить целых 12 соединений!

Снова приведём для сравнения запрос к таблицам реляционной СУБД.

```
SELECT o.num AS "Номер",
       SUM(oi.amount) AS "Количество"
FROM orders o
     INNER JOIN order_items oi ON o.id = oi.id_order
WHERE o.state = 'Доставлен'
      AND oi.name = 'Мука'
      AND oi.amount > 100
GROUP BY o.num
```

В структуре запроса к РСАЗ можно выделить шаблонные конструкции, необходимые для фильтрации по типу сущности и названию атрибута. Если ввести поверх SQL надстройку над языком в виде макроопределений или даже просто определить соответствующие виды, то число соединений можно сократить, увеличив наглядность кода.

```
CREATE VIEW eav_t
AS
SELECT eav.id_entity,
       e.id_type,
       et.name AS type_name,
       eav.id_attribute,
       a.name AS attr_name,
       eav.value
FROM eav
     INNER JOIN entities e ON eav.id_entity = e.id_entity
     INNER JOIN entity_types et1 ON e.id_type = et.id_type
     INNER JOIN attributes a ON a.id_attribute =
eav.id_attribute
```

Теперь наш запрос примет более удобочитаемый вид.

```
SELECT num AS "Номер заказа",
       SUM(amount) AS "Количество"
FROM
  (SELECT orders.value AS num, order_items2.value AS amount
   FROM eav_t orders
        INNER JOIN eav_t orders2
          ON orders.id_entity = orders2.id_entity
        INNER JOIN entity_relations er
          ON er.id_entity1 = orders.id_entity
        INNER JOIN eav_t order_items
          ON er.id_entity2 = order_items.id_entity
        INNER JOIN eav_t order_items2
          ON order_items2.id_entity = order_items.id_entity
   WHERE orders.type_name = 'Заказ' AND
         orders.attr_name = 'Номер' AND
         orders2.attr_name = 'Статус' AND
         orders2.value = 'Доставлен'
         order_items.type_name = 'Предмет заказа' AND
         order_items.attr_name = 'Название' AND
         order_items.value = 'Мука пшеничная в/с'
         order_items2.attr_name = 'Вес' AND
         order_items2.value > 100 AND
  ) AS selected_orders
```

```
GROUP BY num
```

Следует понимать, что несмотря на некоторое упрощение текста запроса и явное задание четырёх соединений вместо двенадцати, оптимизатор реляционной СУБД все равно будет вынужден оперировать двенадцатью соединениями.

Другой важный аспект — «волшебный» тип колонки «Значение». Все наши предыдущие примеры неявно предполагали некий аналог универсального типа Variant, который может хранить разнообразные величины.

На практике реляционные СУБД такой тип не поддерживают, нет его и в стандарте SQL. Поэтому используются различные собственные реализации «универсального» типа данных для колонки «Значение»:

- строковый или бинарный тип с конвертацией в другие типы данных. Недостатком является более низкое быстродействие по сравнению с хранением в формате встроенных типов СУБД;
- несколько колонок разного типа, например «Значение-строка», «Значение-целое», «Значение-дата» и т. д. Конвертация данных не требуется, но манипуляции с чтением-записью значений в зависимости от типа атрибута усложняют запросы, необходимо наращивать язык с помощью тех же видов или хранимых процедур. Другой недостаток — разрежённость хранения, ведь если определить 5 колонок для значений разного типа, то только одно значение в каждой строке будет использовано. Некоторые СУБД, например, SQL Server 2008 и выше предлагают оптимизированное хранение для таких разреженных колонок (подробнее см. sparse columns);
- отдельные таблицы, содержащие значения только заданного типа, связанные с основной eav по короткому уникальному идентификатору. Недостаток — необходимость дополнительных соединений, в том числе внешних (LEFT JOIN вместо INNER JOIN);
- смешанные подходы, например, создание отдельных таблиц только для хранения длинных текстовых и бинарных значений атрибутов.

Модель САЗ: плюсы и минусы

Преимуществами модели САЗ являются:

- гибкость схемы данных и возможность её изменения в динамике;
- высокая степень унификации и повторного использования структур хранения;
- возможность выполнения контекстно-независимых запросов, например, не принимающих в расчёт тип сущностей;
- возможность реализации на основе реляционной СУБД;
- относительно простая базовая концепция.

К недостаткам следует отнести:

- сложность манипуляций данными вне CRUD-запросов;
- как следствие, необходимость вводить собственный язык манипуляции данными;
- в случае реализации на основе реляционной СУБД, более низкая производительность за счёт большего числа соединений в запросах;
- как следствие, необходимость оптимизации физического хранения и прямых подсказок оптимизатору со стороны программиста, что снижает масштабируемость по мере роста объёма данных.

Несмотря на свою гибкость, модель САЗ имеет ряд недостатков, а рекомендовать её реализацию на основе реляционной СУБД можно лишь в отдельных случаях. Также модель не слишком хорошо подходит в ситуации схем с большим количеством связей между сущностями и жёсткими ограничениями ссылочной целостности.

В наиболее дружественном для программиста и пользователя варианте, это должна быть полноценная СУБД, реализующая модель САЗ на физическом уровне и предоставляющая разработчику готовый входной язык определения структур и манипуляции данными, а не надстройка над реляционной СУБД.

Неполно структурированные модели данных

К неполно структурированным моделям данных (НСМД, в англоязычной терминологии semi-structured data models) относятся те способы их организации, в которых **схема и данные не разделены**. Обратите внимание, схема не «зашита» в данные, подобно встроенной DTD в XML-файле, а отсутствует в явном виде, при этом данные остаются структурированными.

> *В русскоязычной среде нередко используется термин-калька «полуструктурированные данные», не отражающий, на мой взгляд, суть. Степень структурности можно оценить, она варьируется, поэтому данные не могут быть всегда структурированы только наполовину. Другой термин «слабоструктурированные» также не соответствует смыслу. Данные могут быть структурированы «сильно», но при этом все-таки неполно.*
>
> *Тем не менее, если вы встречаете упоминание о полу- или слабо структурированных данных, то, скорее всего, речь пойдёт о НСМД.*
>
> *В рамках книги будет использован термин «неполно структурированные данные» (НСД).*

Например, в реляционной БД известно, что в определённой таблице в каждой строке всегда имеется столько значений, сколько определено колонок. Сами значения в каждом поле строки соответствуют типам колонок. Мы не можем добавить дополнительное значение в отдельные строки, не определив новых колонок для всей таблицы.

В случаях, когда исходные данные неполно структурированы, а возможности по их анализу и классификации ограничены, такой подход может оказаться препятствием на пути первичного сбора информации, вынуждая переходить, например, к модели САЗ без всяких ограничений со стороны метаданных.

Другая область применения — обмен информацией в гетерогенной среде. Объект в одной системе может быть упакован (сериализован) в какой-либо

из форматов НСМД, затем передан другой системе, которая на базе полученного описания и данных создаст соответствующий объект в своей среде. Согласование схем в данном случае не требуется, хотя для эффективности обработки и лучшей переносимости такие схемы часто формализуют и даже стандартизируют на отраслевом уровне.

Поскольку в НСМД данные остаются структурированными, они могут быть проанализированы и интерпретированы непосредственно в ходе обработки. Таковым, например, является XML, не имеющий ограничений XML-схемы или DTD. Любой разборщик XML позволяет работать с таким нестрогим документом посредством навигации по иерархии тегов и спискам атрибутов.

Например, если интерпретатор умеет выделять в документе информацию по заказам внутри иерархии `<order>/<number>` независимо от её уровня вложения, то он справится с задачей в обоих приведённых ниже случаях.

```
<!-- Случай 1. Заказы списком -->
<orders>
  <order>
    <number>123</number>
    <items />
  </order>
</orders>
...
<!-- Случай 2. Заказы в составе продаж -->
<sales>
  <sale>
    <order>
      <number>123</number>
    </order>
  </sale>
</sales>
```

Другим широко распространенным подходом НСМД является JSON (JavaScript Object Notation). По сравнению с XML «без схем», JSON представляет собой менее удобный для восприятия человеком формат. Однако, он более прост для обработки анализаторами и может быть более экономичным по объёму за счёт отсутствия замыкающих тегов.

Как и XML, JSON также может иметь схему данных, содержащую информацию о типах элементов и порядке их следования в документе. Ниже — пример сохранения данных в JSON-документе.

```
{
    "номер_заказа": "123-45",
    "дата": "2014-02-15",
    "состояние": "выполнен",
    "адрес_доставки": {
        "улица": "Ивановский пр. 5",
        "город": "Крыжополь",
        "индекс": "333222"
    },
    "телефоны": [
        { "тип": "домашний", "номер": "921-1112233" },
        { "тип": "мобильный",  "номер": "921-1114455" }
    ]
}
```

Подходы НСМД тесно связаны с обширной темой самодокументированных данных и понятием «семантический веб» (semantic web). Согласно этой концепции, информация передаётся между системами в НСМД, при этом структура считается самодостаточной для распознавания и интерпретации смысловых и контекстных связей в документе без предварительного согласования схем данных для обмена.

Документ-ориентированная модель и NoSQL

Документ-ориентированная модель (ДОМ, не путайте с DOM — Document Object Model, объектной моделью документа, используемой при разборе XML) носит такое название, потому что основным её элементом является документ. В отсутствии строгого определения, это понятие соответствует иерархическим XML, JSON-документам и подобным структурам.

Документы организованы в линейные списки — коллекции, которые могут содержать другие коллекции. Таким образом, ДОМ представляет собой синтез иерархической и неполно-структурированной моделей данных.

В отличие от реляционной модели, где каждая строка таблицы имеет строго оговорённое число полей, определяемых заданными колонками и их типами, или от канонической иерархической модели, где каждый сегмент

содержит список записей одного типа, документы списка в ДОМ могут иметь структуру с общей и различающимися частями.

Эта возможность позволяет относительно просто расширять структуру хранимой в БД информации. Для аналогичных расширений в реляционной модели приходится создавать таблицы расширений, связанные отношением «один-к-одному» с главной таблицей. Такое расширение в случае иерархии таблиц является выделением подтипов (sub-typing). В случае множественного подчинения речь идёт об агрегации.

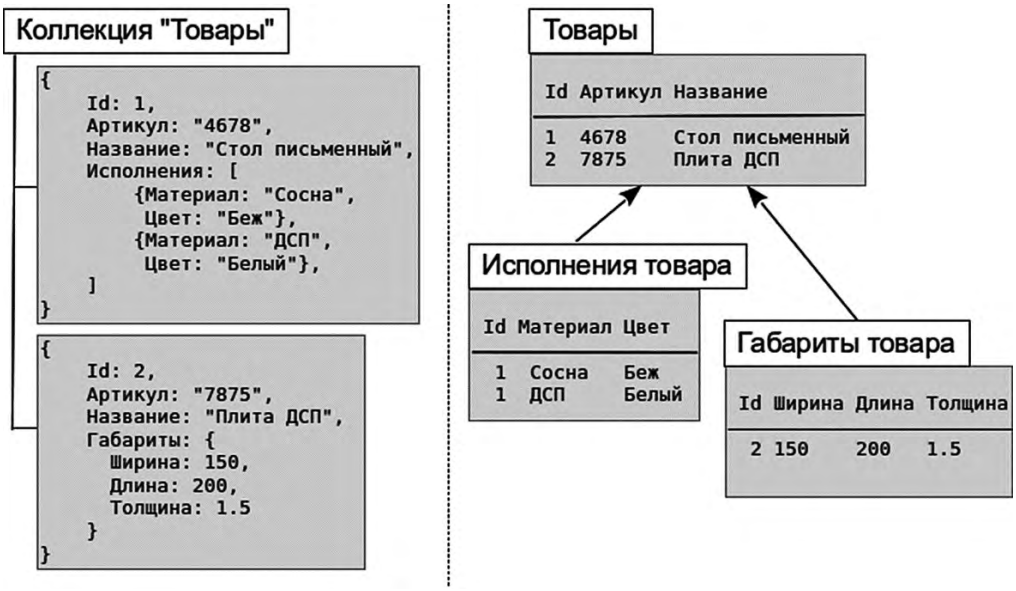

Рис. 7. Расширение структур в документ-ориентированной и реляционной моделях

СУБД, реализующие документ-ориентированную модель, получили название NoSQL, явно обозначающий их отличие от реляционных подходов. В дальнейшем термин NoSQL «лёгким движением руки» превратился в аббревиатуру, которая расшифровывалась уже как Not Only SQL (не только SQL), став обозначением для семейства нереляционных СУБД, созданных преимущественно в последнее десятилетие и в большинстве своём под свободными лицензиями с открытым исходным кодом: Cassandra, CouchDB, MongoDB и многие другие.

Из главы про модель САЗ (нерасширенную) на примере запроса по сущностям, имеющим атрибут «Вес», мы знаем, что отсутствие схемы данных может иметь побочные последствия.

Аналогичным образом этот принцип действует и в ДОМ. **Нет схемы — нет однозначной интерпретации элементов данных**. Чтобы избавиться от подобных побочных эффектов и уметь отличать вес упакованного товара от веса гружёного контейнера, необходимо вводить схемы данных. И тогда ДОМ превращается в обычную иерархическую модель.

Слабым звеном NoSQL является, как это следует из названия, отсутствие стандартизованного декларативного языка запросов, коим является SQL. Иллюстрация ниже показывает, как переписать SQL-запрос на последовательность команд входного языка MongoDB,

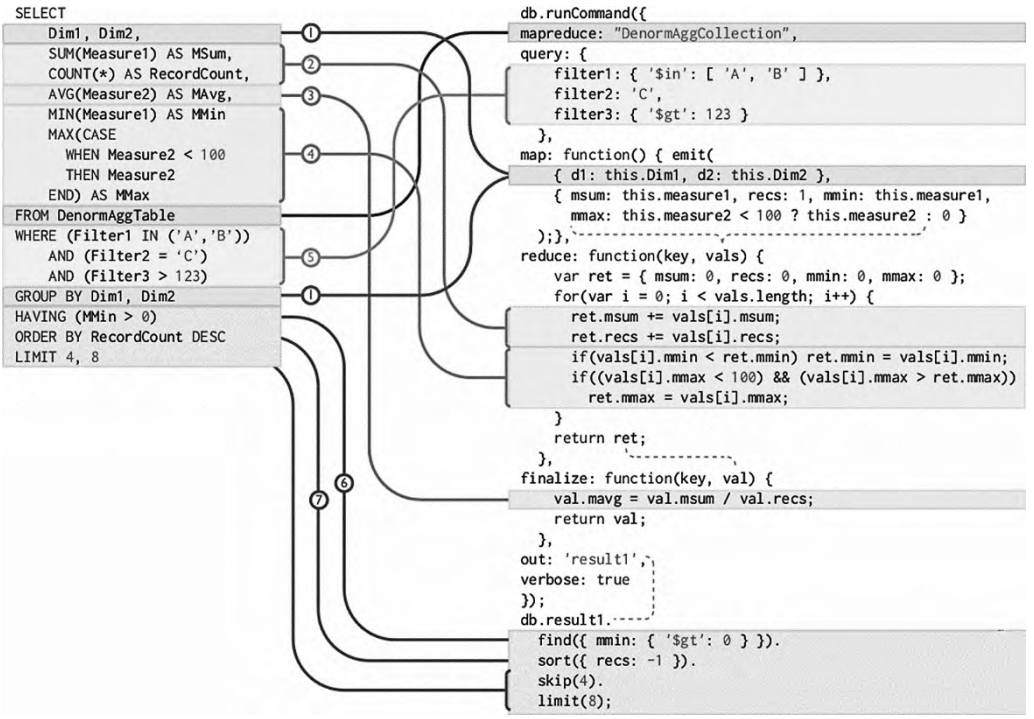

Рис. 8. Сравнение текстов запросов MySQL и MongoDB (источник: http://rickosborne.org)

Пояснения к трансформации:

1. Функции агрегации в MongoDB программируются вручную с использованием механизма map-reduce. Колонки измерений вынесены в map.

2. Непосредственная реализация функций агрегации над величинами (мерами).
3. Агрегации, зависящие от числа обработанных записей, должны ждать завершения основной функции.
4. Внутри можно использовать обычную процедурную логику, например, условные переходы.
5. Выражения фильтров записываются в близком к ORM стиле.
6. Фильтры уровня агрегации должны быть применены к результату map-reduce.
7. Восходящая сортировка кодируется единицей «1», нисходящая — «1».

Данная иллюстрация хорошо показывает, чем декларативный язык 4 поколения SQL, отличается от императивного, основанного на языке 3 поколения типа Ява-скрипт.

Для меня же интерпретация данной картинки представила интерес совсем по другой причине: по замыслу авторов подобное переписывание запроса представляет собой более привычный и простой (!) для программиста код...

Коллеги, осваивайте SQL. Всякий раз когда вам кажется, что SQL-запросы слишком трудны, обращайте свой взор на приведённую выше иллюстрацию и оценивайте, во сколько раз код NoSQL будет длиннее.

Ещё одну типичную ситуацию выпукло изобразили редакторы веб-сайта http://www.commitstrip.com. С их любезного разрешения ниже публикуется переведённая мной на русский язык история.

В целом говорить о каких-то новых возможностях NoSQL не приходится. Существует класс задач сбора, хранения и первичной обработки неполно структурированной информации, где продукты данного типа могут быть рассмотрены в качестве альтернативы реляционным СУБД, не имеющих соответствующих расширений для работы с XML или JSON. В остальных же случаях действует простое правило:

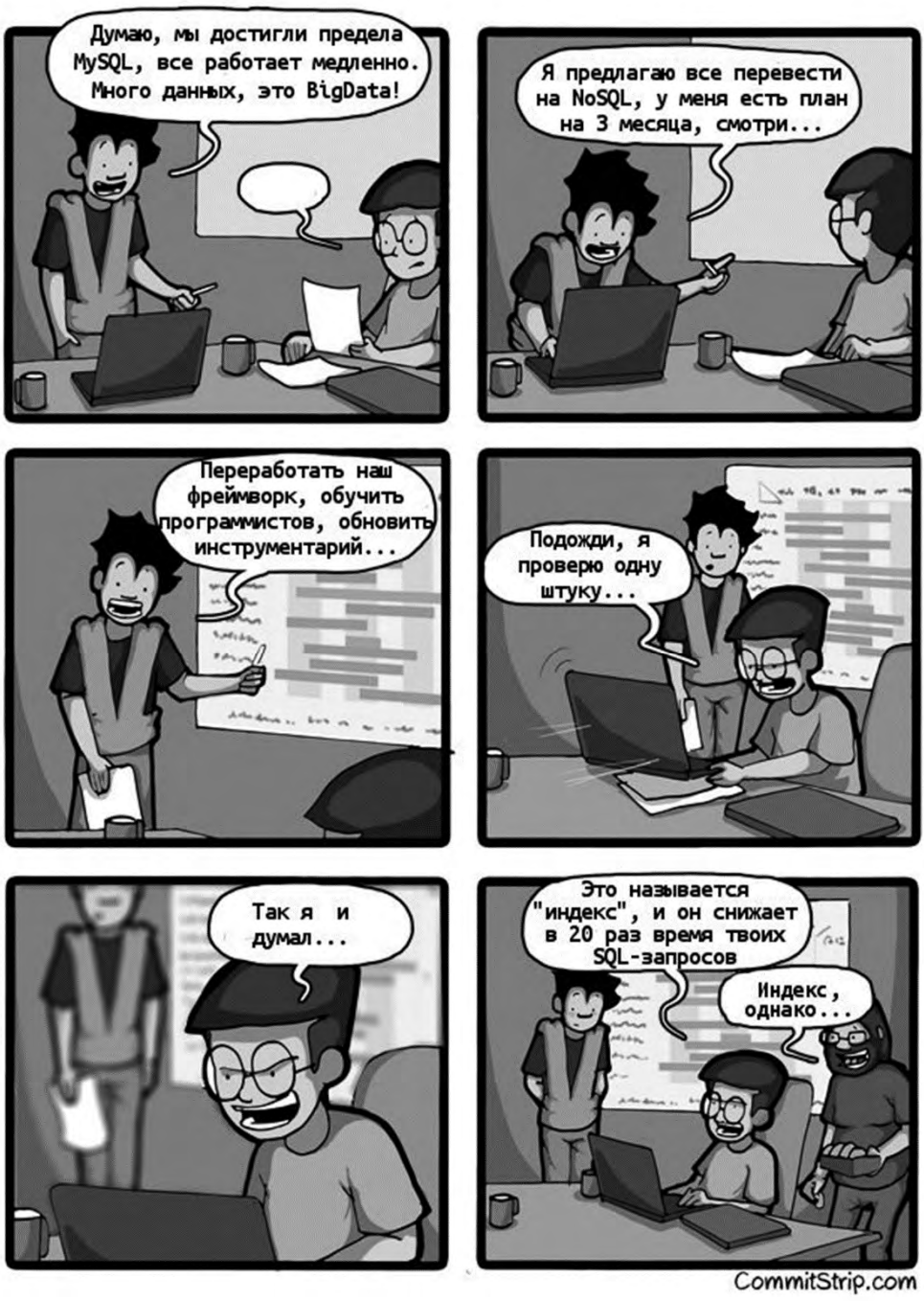

Рис. 9. Некомпетентность в SQL — не причина миграции на NoSQL.

Некомпетентность в SQL и реляционных СУБД не может служить серьёзным доводом в выборе NoSQL СУБД.

Многомерные модели данных

Центральным понятием модели является **многомерный куб** или **гиперкуб**. С привычным всем стереометрическим кубом сходство можно найти лишь при числе измерений, равном трём. Искать наглядные аналоги гиперкуба в геометрии трудно, а трёх измерений хватает разве что для самых простых аналитических выборок. Положим, например, что при оценке сбыта нужны всего 3 измерения: покупатель, период, товар. Значением узла куба по этим измерениям может быть сумма продаж или количество товара, если он отгружается в сравнимых единицах.

Рис. 10. Данные в ячейках гиперкуба трёхмерной модели

Гиперкуб по сути является многомерным разреженным массивом. Отличия от понятия массива, используемого в программировании незначительны:

- индексы измерений поименованы: вместо целых чисел 1, 2,...N используется их заданное соответствие. Например, 1 — «Мука в/с», 2 — «Дрожжи», 3 — «Сахар» и т. д. — это обычная кодификация;

- для доступа к значениям гиперкуба не обязательно специфицировать все индексы измерений.

Последний пункт требует пояснений. Чтобы получить значение элемента N-мерного массива, необходимо задать индексы по всем измерениям. Следующее выражение возвращает сумму продаж муки клиенту «ООО Пирожки» в марте 2014 года.

```
Сумма := Сбыт[Индекс("2014-02"), Индекс("ООО Пирожки"),
Индекс("Мука")];
```

что эквивалентно в нашем примере

```
Сумма := Сбыт[2, 2, 1];
```

В многомерной модели это необязательно. Если индекс не специфицирован, то вычисляется заданная функция агрегации по всем значениям данного измерения. Например, следующее выражение вернёт сумму продаж любых товаров клиенту «ООО Пирожки» за март 2014 года.

```
Сумма := Сбыт[Индекс("2014-02"), Индекс("ООО Пирожки"), ?];
```

Немаловажно, что в реализации многомерных моделей агрегаты предварительно вычисляются на этапе заполнения гиперкуба информацией, поэтому скорость выполнения следующих запросов будет примерно одинакова и составляет, как правило, миллисекунды.

```
Сумма := Сбыт[Индекс("2014-02"), Индекс("ООО Пирожки"),
Индекс("Мука")];
Сумма := Сбыт[?, Индекс("ООО Пирожки"), Индекс("Мука в/с")];
Сумма := Сбыт[?, Индекс("ООО Пирожки"), ?];
```

В статьях нередко смешивают понятия интерактивной аналитической обработки OLAP и гиперкубов, называя последние OLAP-кубами. Такое смешение вызвано давними маркетинговыми причинами: СУБД компании Microsoft, реализующая многомерную модель, вначале носила название Microsoft OLAP.

Значение термина OLAP гораздо шире гиперкубов, поскольку сюда включены все методы интерактивной аналитической обработки данных: хранилища данных (data warehouse), так называемые витрины данных (data mart), добыча данных (data mining) и т. п. Часть перечисленных понятий может быть одновременно объектом пакетной обработки.

Применение многомерных моделей в транзакционной среде, чтобы одна и та же СУБД служила для обработки транзакций и аналитики, не представляет практического интереса по следующим причинам:

- максимальная детализация фактов и событий, необходимая в транзакционной обработке, увеличивает количество измерений, узлов и, соответственно, число предварительно вычисляемых агрегатов, что, в свою очередь, приводит к экспоненциальному росту физического размера БД;
- вставка, обновление и удаление записей приводит к пересчёту значений узлов и агрегатов, что резко увеличивает время и толщину транзакции.

Также следует упомянуть, что многомерная модель может быть реализована средствами реляционной СУБД путём надстраивания метаязыка и исполнительной среды для оперирования многомерными структурами. В этом случае гиперкуб представляется своими двумерными разрезами в виде иерархии таблиц согласно возможным сочетаниям из N измерений по k, где k лежит в диапазоне от 1 до N. Например, рассмотренный выше трёхмерный гиперкуб может храниться в следующей схеме.

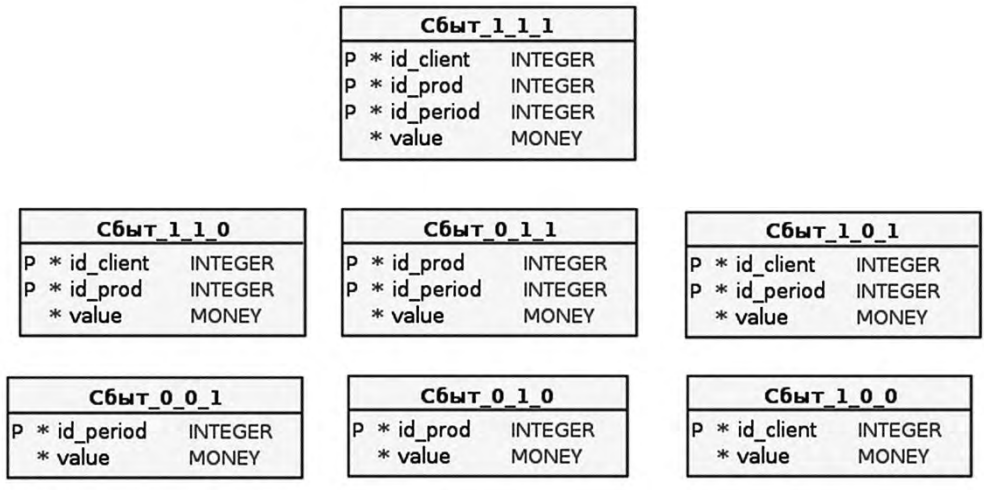

Рис. 11. Реляционная схема хранения трёхмерного гиперкуба

Подобная реализация также имеет название ROLAP (Relational OLAP) и поддерживается большинством систем аналитической обработки.

В общем случае, число необходимых таблиц M_{TAB} для хранения N-мерного гиперкуба определяется как сумма всех возможных сочетаний из N по k, где k меняется от 0 до N.

$$M_{TAB} = \sum_{k=0}^{N} C_N^k = 2^N$$

В нашем примере при количестве измерений N=3 таблиц должно было быть $2^3=8$, но для простоты исключена таблица с единственной строкой, не содержащая колонок-измерений. Соответствующую величину можно высчитать по наименее детализированной таблице (с одной колонкой-измерением).

Так как в общем случае для N измерений требуется 2^N таблиц, их число растёт экспоненциально, поэтому на практике возможности ROLAP ограничиваются их небольшим ($N < 10$) количеством.

О применимости NoSQL

Как уже было сказано, NoSQL-решения и, в частности, основанные на ДОМ, имеют вполне конкретную область применения: первичный сбор и индексация информации, чью структуру мы не в состоянии классифицировать настолько полно, чтобы спроектировать соответствующую схему базы данных.

Если цель дальнейшей классификации и аналитической обработки собранной информации не ставится, то такую БД можно непосредственно эксплуатировать, используя, например, анализ семантических связей в «кипящей каше» из постоянно поступающих в систему документов. Собственно, исследовательские центры поисковых машин Интернета занимаются в том числе и такими вопросами.

Современные РСУБД, обладающие функциями хранения и обработки XML-документов, предоставляют проектировщику возможность гибридных решений, например, в условиях высокой нагрузки. Пример

одного из таких решений приведён в главе «Неполно структурированные данные и высокая нагрузка» раздела о проектировании.

С другой стороны, есть класс задач, где исходная информация хорошо структурирована, но использование реляционной модели не является оптимальным. Речь идёт прежде всего о сущностях со сложной и многоуровневой иерархией вложения. Если хранить такие сущности в реляционной СУБД, то каждый тип должен быть в нормализованном виде отражён на десятки таблиц. Соответственно, обработка таких данных будет вызывать проблемы с производительностью, код запросов будет сложен и труден в сопровождении.

В таком случае также имеет смысл обратить внимание на нереляционные решения, но вовсе не на ДОМ-СУБД, а на поддерживающие многомерные и сетевые модели данных.

Приведём и другой тип труднореализуемой в рамках реляционной СУБД задачи. Пусть имеется множество однотипных объектов и связанных с каждым состояния, являющиеся длинным списком значений. Для простоты будем считать их двоичными: Состояние_1 = 0, Состояние_2 = 1, ... Состояние_N = 0. Число N может быть большим: сотни и тысячи состояний.

Упрощённый пример этого типа задач из практики: объектом является пользователи, а в качестве состояний выступают их ответы на длинную анкету из вопросов «да/нет». Другой вариант — пользователи и признаки посещения/непосещения страницы веб-сайта; в этом варианте можно сэкономить на объёме и хранить только посещения. Ещё пример — дискретные состояния системы в виде показания датчиков на заданные моменты времени.

В рамках реляционной модели построить собственно структуру для хранения данных не представляет труда.

Приведённая схема напоминает уже рассмотренную для хранения модели САЗ. Отличие пока количественное: в САЗ мы оперируем сущностями с единицами и десятками атрибутов, тогда как в данном примере — с сотнями и тысячами.

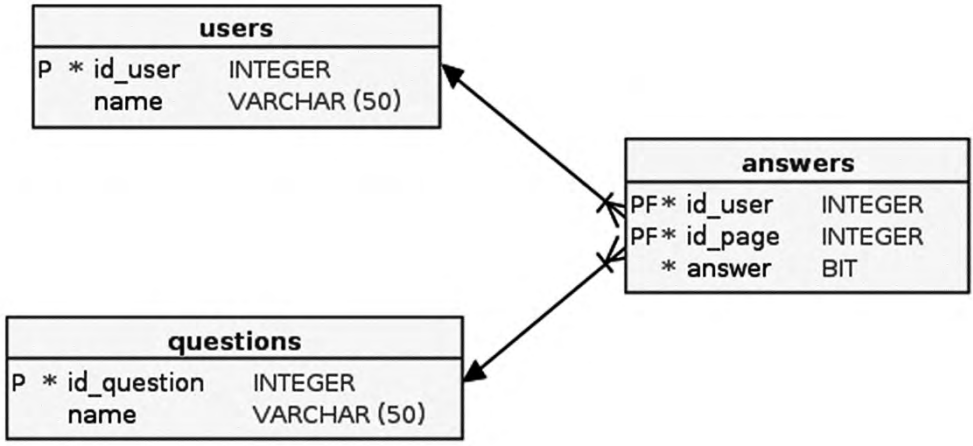

Рис. 12. Реляционная схема хранения множества бинарных состояний объекта

Количественное отличие быстро переходит в качественное, стоит лишь начать производить из БД простейшие выборки аналитического характера, например, «определить количество анкетированных, у которых ответ на Вопрос_1 - да, на Вопрос_3 — нет, на Вопрос_N — да и т.д». Число N будет однозначно определять число соединений в запросе. Для простоты считаем, что `id_question` соответствует его номеру.

```
SELECT COUNT(1)
FROM
  users u
  INNER JOIN answers a1 ON u.id_user = a1.id_user
    AND a1.id_question = 1 AND a1.answer = 1
  INNER JOIN answers a2 ON u.id_user = a2.id_user
    AND a2.id_question = 2 AND a2.answer = 0
  ...
  INNER JOIN answers aN ON u.id_user = aN.id_user
    AND aN.id_question = N AND aN.answer = 1
```

Если N достаточно велико, то время выполнения такого запроса становится неприемлемым для интерактивной аналитической обработки, занимая долгие минуты. На практике, проблемы начинаются уже при N > 10..15, если таблица ответов содержит даже десятки миллионов строк[13], а план выполнения запроса представляет собой скопление вложенных циклов со внутренними хеш-слияниями промежуточных выборок.

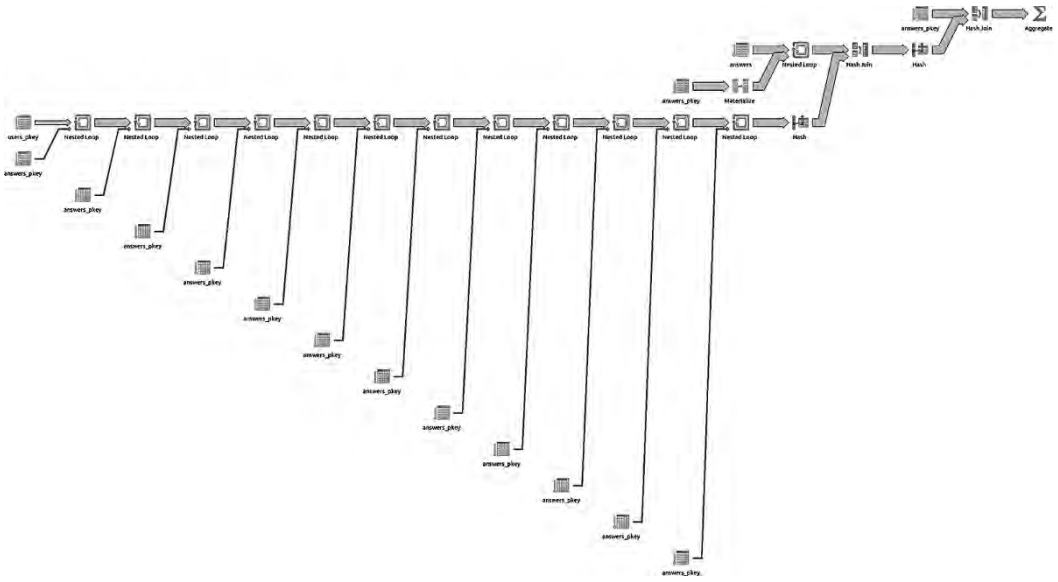

Рис. 13. План выполнения запроса в СУБД PostgreSQL при N = 15

Разумеется, средствами тонкой настройки и покупкой более мощной аппаратуры можно несколько увеличить производительность (более подробно об этом в следующих главах), но увеличив N добавлением ещё нескольких INNER JOIN мы, в лучшем случае, возвращаемся на исходные позиции.

Зная, что начиная с некоторого порога N время выполнения первого запроса будет расти линейно, можно переписать его в другом виде, без соединений, но с необходимостью всякий раз производить полное сканирование длинной таблицы ответов. Сканирование и промежуточное агрегирование на основе хеш-функций также негативно влияет на

[13] Разумеется, приведённые значения N приблизительны, точные оценки зависят от многих факторов, прежде всего от производительности аппаратуры и конкретной СУБД. Для их получения необходимо производить нагрузочные тесты.

производительность, но время выполнения становится предсказуемым, хотя в случае N > 20..30 может вновь стать неприемлемым.

```
SELECT COUNT(1)
FROM
  (SELECT id_user, COUNT(1)
   FROM answers
   WHERE id_question = 1 AND answer = 1
         OR id_question = 2 AND answer = 0
         ...
         OR id_question = N AND answer = 1
   GROUP BY id_user
   HAVING COUNT(1) = N -- соответствует числу вопросов в
запросе
  ) t
```

Соответствующий план будет примерно следующим, независимо от N.

Рис. 14. План выполнения запроса в СУБД PostgreSQL

При N порядка сотен, программист начинает сталкиваться с техническими ограничениями СУБД на число соединений в одном запросе и на общую длину текста запроса.

Как можно увидеть, даже в случае очень простых запросов проявляются трудности их реализации. Стоит несколько усложнить условия выборки, например «все пользователи, ответившие на вопросы NNN заданным образом, но по группе вопросов MMM совпадение не превышает 50%», как технические проблемы начинают расти в геометрической прогрессии.

Вариант решения с изменением начальной схемы путём денормализации и транспонирования таблицы ответов в структуру со многими колонками «Вопрос_1»,... «Вопрос_N» также не является возможным по техническим причинам: большинство реляционных СУБД имеет ограничения на число колонок в таблице или на общую длину строки (размер в байтах). Также будет невозможно построить индекс по всей совокупности колонок.

В таких ситуациях необходимо задумываться о нереляционных подходах к организации данных и, прежде всего, о СУБД, реализующие многомерные модели.

Множественная и навигационная обработка, менеджеры записей

Как было отмечено выше, в отличие от иерархической и сетевой моделей реляционная относится не к графовым, а к теоретико-множественным. Это означает, что все операции происходят не с единичными экземплярами записей и ссылками, а со множествами записей в соответствии с выбранными критериями и ограничениями. В частности, понятие «текущая запись» в реляционной модели отсутствует[14], и программисту надо отходить от традиционной последовательной обработки данных, перестраивая мозги на работу со множествами.

Сравните два фрагмента программ, производящих одинаковый результат. При множественной обработке:

```
-- установить цену для всех товаров с идентификатором 7615
-- в заказе № 123-45
UPDATE order_items
SET price = 10.55
WHERE id_product = 7615 AND
      id_order IN (SELECT id FROM orders WHERE number = '123-
45')
```

И при последовательной обработке:

```
SELECT orders    // текущая область данных - "заказы"
SET ORDER TO id  // упорядочиваем по индексу id
SEEK '123-45'    // находим первую запись по значению
IF FOUND() THEN  // если нашли
  SET @id_order = current.id
  SELECT order_itmes // переходим в область данных "элементы
заказов"
  SET ORDER TO id_order
  SEEK @id_order
  BEGIN TRANSACTION
  WHILE current.id_order = @id_order
```

[14] Многие реляционные СУБД поддерживают в своих процедурных расширениях так называемые курсоры, позволяющие перемещаться по выбранному множеству записей.

```
  DO
    IF current.id_product = 4615 THEN
      current.price = 10.55
    ENDIF
    NEXT
  END
  COMMIT TRANSACTION
ENDIF
```

Код последовательной обработки в общем случае намного менее лаконичен и нагляден, чем обработки множественной, что имеет простое объяснение: в первом примере используется **декларативный язык**.

Основное отличие декларативного языка в том, что программист описывает его в терминах «что делать», а «как делать» — решает исполняющая код система. В противоположность декларативному, императивный язык позволяет программисту определять непосредственно «как делать». У обоих подходов есть свои преимущества и недостатки, проявляющиеся в разных случаях применения технологии. В случае СУБД декларативный язык является несомненным преимуществом, по многим причинам. Приведу некоторые:

- максимальная независимость от физической организации данных;
- динамический выбор оптимальной стратегии (плана) выполнения запросов к СУБД в зависимости от состояния (например, от объёма опрашиваемых данных);
- динамическая параллелизация этапов выполнения запроса к СУБД;
- более краткий код, как уже отмечалось.

Проблему выбора оптимального плана выполнения запроса в императивном варианте обработки можно проиллюстрировать простым примером. Предположим, нужно подсчитать количество заказов, сделанных данным клиентом и содержащих определённый товар. Код может быть следующим:

```
SET @id_client = 4587 // идентификатор клиента
SET @id_prod = 493    // идентификатор товара
SET @orders_count = 0
SELECT orders
```

```
SET ORDER TO id_client
SEEK @id_client
IF FOUND()THEN
  WHILE current.id_client = @id_client
  DO
    SET @id_order = current.id
    SELECT order_items // проверяем товар в позициях заказа
    SET ORDER TO id_order
    SEEK @id_order
    IF FOUND() THEN
      WHILE current.id_order = @id_order
      DO
        IF current.id_prod = @id_prod THEN
          @orders_count = @orders_count + 1
          BREAK
        ENDIF
      END
    ENDIF
    SELECT orders  // возвращаемся к циклу по заказам
    NEXT
  END
ENDIF
PRINT @orders_count
```

Суть проблемы состоит в том, что мы заранее не знаем, каких записей в БД больше: клиентов или товаров. У некоторых предприятий может быть огромный каталог товаров с десятками тысяч позиций, при этом имеется лишь небольшой (десятки единиц) список оптовых клиентов. В других — наоборот, небольшая номенклатура товаров сбывается сотням тысяч клиентов по всему миру.

В нашем примере подсчёт идёт со стороны цикла по заказам, сделанных данным клиентом, затем для каждого заказа мы проверяем наличие в его позициях интересующего нас товара. Если клиентов в БД много меньше, чем товаров, то такая обработка будет оптимальной.

А если наоборот? Тогда оптимальная по времени обработка должна идти со стороны позиций заказа.

```
SET @id_client = 4587 // идентификатор клиента
SET @id_prod = 493    // идентификатор товара
SET @orders_count = 0
SELECT order_items
```

```
SET ORDER TO id_prod
SEEK @id_prod
IF FOUND() THEN
  WHILE current.id_prod = @id_prod
  DO
    SET @id_order = current.id_order
    SELECT orders // проверяем клиента сделавшего заказ
    SET ORDER TO id
    SEEK @id_order
    IF FOUND() THEN
      IF current.id_client = @id_client THEN
        @orders_count = @orders_count + 1
        BREAK
      ENDIF
    ENDIF
    SELECT order_items  // возвращаемся к циклу по позициям
заказам
    NEXT
  END
ENDIF
PRINT @orders_count
```

Декларативный подход решает данную проблему.

```
SELECT count(*)
FROM orders o
     INNER JOIN order_items oi ON o.id = oi.id_order
WHERE o.id_client = 4587 AND oi.id_prod = 493
```

Интерпретирующая запрос подсистема СУБД, так называемый оптимизатор, учтёт статистическую информацию о распределении (плотности) значений идентификаторов клиентов и товаров соответственно в колонках `id_client` и `id_prod` или их индексах, после чего будет выбран оптимальный вариант выполнения этого запроса.

Разница в размере кода и, особенно, в его универсальности значительна. Но чем приходится платить?

Прежде всего, программисту в общих чертах надо понимать, как работает реляционная СУБД внутри. Если в нашем примере в базе данных отсутствуют индексы по колонкам `orders.id_client`, `order_items.id_order` и `order_items.id_prod`, то в общем случае планом выполнения будет прямое сканирование таблиц. Но даже если

индексы построены, следует оценить их избирательность, чтобы удалить бесполезные.

А что произойдёт, если во время расчёта другой пользователь добавит или удалит строки? К такого рода вопросам мы обратимся в главе, посвящённой изоляции транзакций.

Значит ли сказанное выше, что СУБД с навигационными подходами являются неполноценными? Разумеется, нет. Даже если при работе с данными используется понятие «текущая запись», это ещё не значит, что СУБД относится к категории *менеджеров записей*.

Основное отличие менеджеров записей от полноценных СУБД — **отсутствие общей схемы базы данных**. Менеджеры записей могут быть даже транзакционными, как Btrieve, но по своей логике они гораздо ближе к системам на основе плоских файлов, чем к СУБД, реализующим одну из перечисленных моделей данных.

Отсутствие общей схемы данных вовсе не означает отсутствие схем, как таковых. Скорее всего вы столкнётесь со множеством схем (структур) для каждого типа хранимых сущностей. Например, в случае СУБД, основанных на индексно-последовательном доступе, каждый файл является аналогом реляционной таблицы с колонками и строками. Однако, в системе менеджера записей невозможно определить ссылочную целостность, являющуюся частью логики хранения данных. Эти функции должно взять на себя непосредственно приложение.

> В общем случае, если кроме навигационного подхода к обработке данных в приложении приходится реализовывать логику слоя хранения данных, значит СУБД относится к категории менеджеров записей.

Объектная модель и объектно-реляционная проекция

Сведения об объектной модели намеренно вынесены из рамок главы «Другие подходы и модели». Причина проста: объектной модели данных не существует, как не существует, например, функциональной или событийной модели данных. Соответствующие модели, точнее говоря, подходы к

разработке приложений, разумеется, существуют, но без уточняющего слова «данных», важного для нас в рамках сюжета книги.

Тем не менее, доминирующая на сегодняшний день в программировании объектная модель и соответствующие подходы оперируют какими-то данными. Причём данные эти инкапсулированы, то есть встроены и сокрыты в объектах. Как ими управлять: извлекать согласно критериям запросов, хранить, модифицировать и т. д.?

В канонической процедурной программе данные отделены от кода, потому могут быть организованы в соответствии с более абстрактной теорией. Полистав классические монографии-учебники Дональда Кнута [24] или Никлауса Вирта [25] можно найти немало математики, делающей синтез и оптимизацию структур данных осмысленным занятием для применяемых алгоритмов.

С объектами сложнее. Математической базы за объектно-ориентированным подходом (ООП) не стоит, это детище инженерных экспериментов 1960-х годов (см. например язык Симула-67). Стандартизация объектной модели проводится «задним числом» консорциумом OMG, что не мешает объектным средам иметь свои особые понятия, не имеющие полных аналогов ни в одной версии UML. С начала массового распространения ООП прошло более 30 лет, но стандартизация внутренней реализации объекта, позволяющая оперировать ими в разных средах — недостижимая на практике утопия до сих пор. Все убожество интероперабельности современных систем живёт на шлюзах, ежесекундно перепаковывающих тучи рассекающих сетевое пространство слепков извлечённых из объектов данных в виде рассмотренных выше XML, JSON и иже с ними.

Поэтому в объектной модели с точки зрения данных нас должна интересовать сугубо структурная часть, являющаяся основой для процедур упаковки (serialization) и восстановления (deserialization). Действительно, если «оголить» класс, убрав из него все функции (методы) и рассматривать только свойства, привязанные к внутренним переменным класса (полям), считая их детерминированными, то есть, не меняющими своё значение

после его считывания, то получается знакомая всем программистам структура. Информационный скелет класса.

> *Небольшое отступление. Не следует забывать принцип инкапсуляции в ООП, ведь совершенно необязательно, чтобы «скелет» соответствовал наружному виду, фасаду. Но в большинстве случаев структурное сходство объекта с тем, что мы наблюдаем извне (интерфейс) имеется. Интересно также, что данный постулат является одной из основ научного мировоззрения, состоящего из следующих допущений:*
>
> *- объекты внешнего мира действительно существуют;*
>
> *- объекты внешнего мира имеют по крайней мере структурное сходство с нашим восприятием этих объектов;*
>
> *- корректны заключения, сделанные на основе принципов математической логики.*

Наиболее близкой к такой «оголённой» объектной модели, к скелету, будет *сетевая модель данных*: структуры и связи между ними. Но как вы знаете, сетевые модели данных в софтостроительной отрасли не являются распространёнными, доминирует реляционная модель. Отсюда возникает задача отображения двух доминирующих подходов, объектного и реляционного, друг на друга.

Реализации отображения данных реляционной СУБД на объекты приложения называется *Объектно-Реляционной Проекцией* (ОРП), соответственно, *Object Relational Mapping* (ORM) в англоязычных источниках.

Первым в аббревиатуре ОРП идёт слово «объектно», что означает условный примат схемы объектной модели, по которой генерируется схема реляционной БД. Но поскольку обе модели являются логическими и имеют сравнимый уровень абстракции, то возможен и обратный механизм: генерация классов объектов по реляционной схеме. В этом случае проектор называется уже РОП (ROM — Relational-Object Mapping). На практике чаще

под словом ORM (ОРП) имеют в виду оба подхода, не уточняя, являются они объектно- или датацентричными.

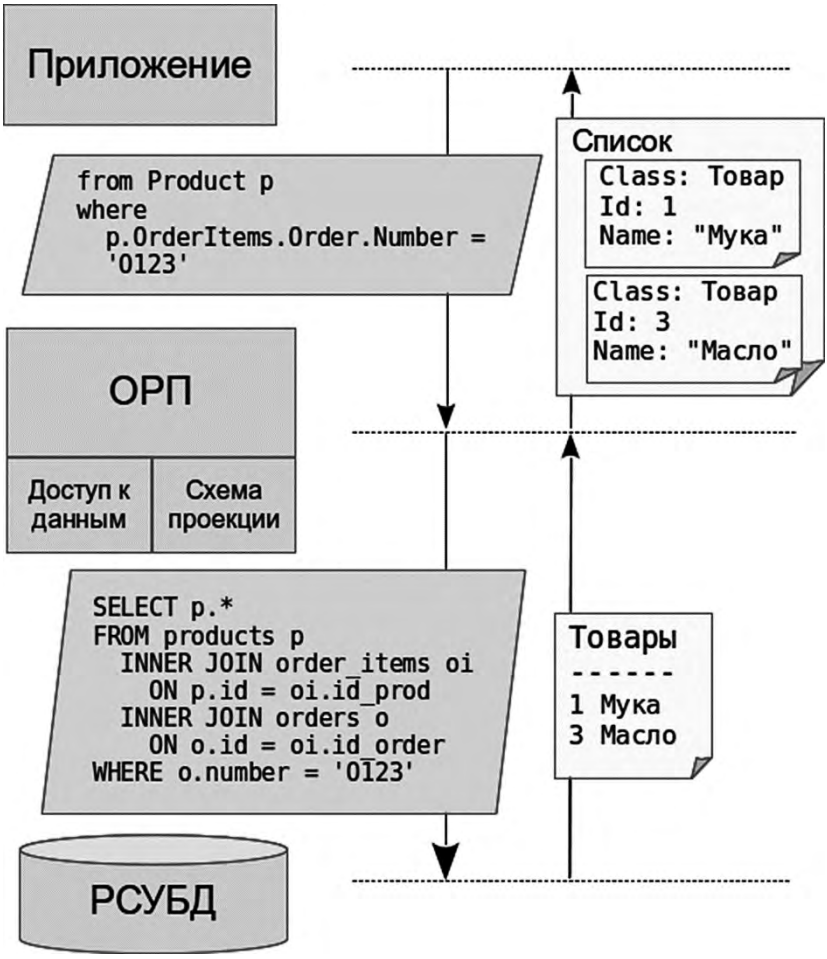

Рис. 15. Прохождение запроса к СУБД через ОРП

На первый взгляд может показаться, что запрос к ОРП короче SQL. Это не вполне верно, потому что язык запросов ОРП представляет собой некоторую надстройку, сравнивать которую нужно с соответствующим аналогом. На уровне СУБД для наиболее часто используемых запросов создаются **виды** (view), который выполняют в том числе ту же функцию — сокращение кода запросов в прикладных программах за счёт сокрытия соединений. Другая важная функция видов — абстрагирование от структуры таблиц, позволяющее менять схему данных не нарушая работу приложения.

Так, если создать в СУБД следующий вид

```
CREATE VIEW v_product_orders AS
SELECT p.*
FROM products p
  INNER JOIN order_items oi ON p.id = oi.id_prod
  INNER JOIN orders o ON o.id = oi.id_order
```

то в прикладной программе можно будет писать даже короче, чем в примере с ОРП

```
SELECT *
FROM v_product_orders
WHERE number = 'O123'
```

С другой стороны, более короткий код запроса к ОРП на рисунке взялся не по волшебству. Под ним лежит проекция — схема отображения таблиц на классы. Иначе запрос не будет даже просто интерпретирован, не говоря уже о выполнении. Проекция представляет собой достаточно громоздкое описание, например, в виде отдельного XML документа. При ручном управлении удобнее использовать один документ на класс, если же проекция генерируется из модели более высокого уровня [13], то проще иметь единый документ на все классы. Вот пример простой проекции с одной связью.

```
<class name="Domain.Sales.Order" table="orders"
schema="sales">
  <id name="Id" access="property" column="id_order">
    <generator class="native">
    <param name="sequence">sales.saq_id_order</param>
    </generator>
  </id>
  <set name="OrderItems" table="sales.order_items"
inverse="true" lazy="true">
    <key column="id_order" />
    <one-to-many class="Domain.Sales.OrderItem" />
  </set>
  <property name="Number" access="property">
    <column name="number" not-null="true" />
  </property>
  <property name="Created" access="property">
    <column name="created" not-null="true" />
  </property>
  <property name="Completed" access="property" type="YesNo">
```

```
    <column name="completed" not-null="true" />
  </property>
</class>
```

Среды с развитыми механизмами метаданных, такие как .NET, позволяют также определять проекцию непосредственно в коде классов, например, в виде атрибутов.

```
[Serializable]
[Class(Schema = "sales", Table = "order_itemss")]
public class OrderItem : NhBase<ProgramRequirements>
{
  [Id(Name = "Id", Column = "id_order_item"), Generator(1,
Class = "native")]
  public virtual int Id { get; set; }

  [ManyToOne(Column = "id_order", OuterJoin =
OuterJoinStrategy.False)]
  public virtual Order Order { get; set; }

  [ManyToOne(Column = "id_prod", OuterJoin =
OuterJoinStrategy.False)]
  public virtual Product Product { get; set; }

  [Property(Column = "price")]
  public virtual decimal Price { get; set; }

  [Property(Column = "qty")]
  public virtual decimal Quantity { get; set; }
}
```

Данный подход может показаться более простым по сравнению с описанием проекции в XML, но у него есть и недостатки, прежде всего, невозможность использования заданной непосредственно в коде программы схемы отображения для приложений другой среды. XML в этом случае представляет собой стандартную информационную развязку, обеспечивающую уменьшение платформенных зависимостей.

С точки зрения проектировщика, разработка проекции является дополнительным этапом впридачу к синтезу диаграммы классов. И этот этап содержит в себе немало подводных камней.

Прежде всего, нет однозначного способа проекции объектной модели на реляционную и обратно. Одна и та же диаграмма классов может быть отображена на структуру таблиц как минимум тремя способами.

1. Хранение всех атрибутов (полей) в одной таблице.
2. Группировка общих атрибутов в одной таблице и разнесение уникальных атрибутов подклассов по связанным таблицам.
3. Представление каждого класса в виде отдельной таблицы с дублированием общих атрибутов.

Первый метод имеет весьма опосредованное отношение к реляционной модели и может рассматриваться разве что как временное решение или средство оптимизации в частном случае. Остальные методы используются автоматизированными инструментами проектирования базы данных, например ErWin или PowerDesigner, и обозначаются как «полный/неполный подтип» (complete/incomplete sub-typing) со схемой «атрибуты подкласса в новой таблице» (inherit only primary attributes, метод 2) и «наследуемые атрибуты добавляются к таблице подкласса» (inherit all attributes, метод 3).

В ОРП, поддерживающих выбор метода генерации схемы БД, например, в JDO, обозначения несколько иные. Первый метод называется «плоским отображением» (flat mapping), остальные методы реализуются с помощью либо «вертикального отображения» (метод 2, vertical mapping), либо «смешанного отображения» (методы 2 и 3, mixed mapping).

С точки зрения реляционной модели, **только метод 2 приводит к созданию нормализованной схемы данных**. Однако, если иерархия классов глубока, то каждый запрос будет сопровождаться соответствующим числом соединений таблиц, что неминуемо скажется на производительности. Поэтому в качестве компромисса возможно комбинирование с методами 1 и 3.

Обратное проектирование, то есть восстановление диаграммы классов из реляционной схемы данных, является операцией с потерей части семантики. Например, реляционная модель поддерживает единственное понятие связей — отношение «один-ко-многим», тогда как в объектной

модели можно непосредственно определить связи «один-к-одному» и «многие-ко-многим». Таким образом, при восстановлении структур из реляционной схемы данных в диаграмме классов окажется «лишний» элемент, реализующий связи «многие-ко-многим».

Следующей проблемой, уже технологического характера, является поддержание схемы проекции объектной модели на реляционную в актуальном состоянии. Приложение развивается в соответствии с изменениями требований к системе. Для простой операции добавления атрибута кроме собственно изменения таблицы и экранных форм потребуются дополнительные изменения на уровне ОРП: меняется описание класса и схемы отображения. Изменение связей и реструктурирование с разбиением таблиц вызовет цепные изменения во многих классах.

Перечисленные случаи и сопутствующие им операции в той или иной степени могут быть поддержаны автоматизированным инструментарием проектировщика, снижающим долю ручного труда в процессе внесения изменений. Один из подходов — разработка, управляемая моделями [13].

Как видите, возникает немало сложностей уже на первых этапах проектирования и разработки. Какие же выгоды даст нам выбранный подход в реализации кроме сокращения кода произвольных запросов при условии отсутствия созданных для них видов?

Основное преимущество проявляется в том, что слой ОРП позволяет программисту оперировать в приложении не SQL-запросами и табличными результатами, а объектами и их списками (коллекциями), выбираемыми по заданным критериям. Это делает программный код более единообразным, нежели изобилующий текстовыми константами SQL-запросов типичный код клиент-серверного приложения, работающего непосредственно с СУБД. То есть вместо одних операторов

```
Query1.Open("SELECT * FROM task_queue WHERE id_task IN (2, 3,
15) AND id_task_origin = 10");
foreach (DataRow row in Query1.Rows)
{
  ...
}
```

программист пишет другие:

```
IList<TaskQueue> queues = session.CreateCriteria()
  .Add(Expression.In("Task.Id", someTasks.ToArray()))
  .Add(Expression.Eq("TaskOrigin.Id", 10))
  .List<TaskQueue>();
foreach (TaskQueue queue in queues)
{
  ...
}
```

Во втором фрагменте, несколько более громоздком, в отличие от SQL-запроса в виде текстовой строки, часть кода проходит проверку компилятором, выявляя возможные ошибки на более ранней стадии. Работу по отображению объектно-ориентированного определения запроса в SQL целевой СУБД берет на себя слой ОРП.

В рамках простых запросов такой подход даёт более строгую типизацию результатов и простоту манипуляции одиночными объектами вместо строк набора данных. Также более простой становится навигация по связям.

```
String cityName = order.Customer.Address.City.Name;
```

Но перемещаясь по связям в самой программе, надо отдавать себе отчёт, что каждое обращение к свойству объекта, возвращающее другой объект, может вызвать запрос к СУБД для инициализации этого объекта. Извлечённые объекты сохраняются в кэше ОРП, поэтому возможна ситуация, когда большая часть БД окажется в оперативной памяти приложения. Возникает проблема синхронизации этого кэша с изменениями, совершаемыми другими приложениями.

Вообще, при использовании слоя ОРП отдавать отчёт приходится во многом. Представление, что генерируемый SQL является аналогом трансляции языка высокого уровня в ассемблер не просто глубоко ошибочно, но быстрыми темпами ведёт к созданию трудносопровождаемых систем с врождёнными проблемами производительности.

Сложности нарастают с переходом к обработке данных за пределами CRUD-логики. Использовать в запросах непосредственно SQL остаётся технически возможным, но тогда теряется основной смысл ОРП. Чтобы избежать работы с SQL, программисту приходится извлекать списки

объектов и обрабатывать их средствами приложения, тогда как в рамках СУБД аналогичная обработка выполнялась бы на порядок быстрее.

SQL — высокоуровневый декларативный специализированный язык четвёртого поколения, в отличие от того же Java или C#, по-прежнему относящихся к третьему поколению языков императивных. Единственный оператор SQL на три десятка строк, выполняющий нечто посложнее выборки по ключу, потребует для достижения того же результата в разы, если не на порядок, больше строк на C#.

Подведём некоторые итоги. Применение ОРП является, скорее, вынужденным, чем необходимым шагом ввиду разницы подходов в проектировании и разработке приложений и баз данных. Также использование ОРП навязывает домен-ориентированный подход к проектированию системы, в центре которой находятся классы бизнес-объектов.

Для простых транзакционных приложений, оперирующих в CRUD-логике, использование ОРП даёт ряд преимуществ: лучшая типизация, более простой код, лучшая переносимость между СУБД. Однако, любые выходы за рамки CRUD приводят программиста к следующим альтернативам:

- продолжать использовать нестандартный и весьма ограниченный собственный язык запросов ОРП типа HQL;
- использовать SQL, динамически отображая результат запросов на специально создаваемые для этих целей классы;
- извлекать списки объектов более простыми запросами и обрабатывать их в приложении.

Все перечисленные способы имеют кроме преимуществ и серьёзные недостатки, анализировать которые следует заранее. Например, HQL мало пригоден для запросов аналитического характера и отчётности, прямое использование SQL рушит абстракцию и приводит к мысли о её изначальной ненужности, обработка объектов средствами приложения требует высокой квалификации программистов, как минимум понимающих

зависимости типа O(n). В случаях, когда обработка нужна для предоставления пользователю табличных форм, например, отчётности, с чем прекрасно справляется созданный для этих целей SQL, ОРП является лишним звеном в системе, лишь увеличивающим время отклика.

Главное, что, на мой взгляд, нужно понимать разработчику системы:

- современные реализации ОРП, такие как Hibernate, являются системами типа «вещь в себе»: они не поддерживают стандартов, привязаны к средам и фреймворкам, содержат множество параметров настройки, имеют изъяны в дизайне, ставшие особенностями (feature), исправлять которые уже никто не будет;
- сложность наиболее развитых ОРП стала сравнима с СУБД. И не мудрено, ведь подобные реализации ОРП по сути — попытки создать встраиваемую в приложение объектную СУБД, используя реляционную лишь в качестве «интеллектуальной файловой системы»;
- использование слоя домена затруднено или даже невозможно из приложений других сред (например, домен-сборка C#.NET из PHP- или C++-приложения), поэтому бизнес-правила могут быть обойдены или многократно реализованы заново. Для решения этой задачи требуется полноценная многоуровневая архитектура с сервером приложений;
- создать систему с высокой производительностью, используя ОРП, невозможно без хорошего знания реляционных СУБД и привлечения в проект соответствующих экспертов.

SQL как универсальный входной язык

Общепринято мнение, что SQL безраздельно принадлежит вотчине реляционных баз данных. Действительно, первые версии языка, тогда ещё называвшегося SEQUEL, были разработаны компанией IBM в рамках одной из пионерских реализации реляционной СУБД System R в 1974 году. С тех пор SQL неизменно является основным входным языком для всех реляционных СУБД.

Однако, возможности языка SQL несколько шире благодаря своей декларативной природе. Реализовать интерпретатор SQL можно поверх самых разнообразных структур, не обязательно реляционных. Разумеется, это может наложить некоторые ограничения на конструкции и отход от стандарта, но суть технологии при этом останется неизменной.

Другой подход — отображение и последующая реализация реляционной модели на базе иерархий или сетевой модели (графа). Ведь любая таблица может быть представлена в виде дерева с несколькими уровнями иерархии.

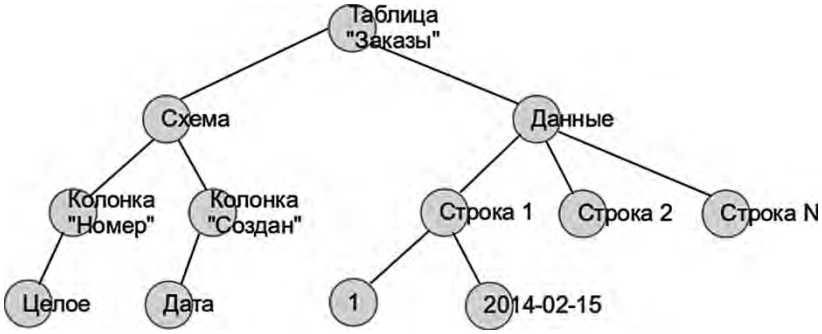

Рис. 16. Пример представления таблицы графом без циклов

Так уже упомянутые СУБД Cache, иерархическая в своей основе, и сетевая Raima реализуют собственное подмножество SQL для доступа к данным. В Cache реляционная модель реализуется поверх иерархической [22], поэтому работа на уровне SQL предоставляет программисту привычные операторы полноценного определения и манипуляции данными в терминах таблиц.

Аналогичным образом реализована реляционная модель поверх сетевой и в Raima. Подмножество SQL поддерживает нестандартный оператор натурального соединения (natural join), позволяющий связывать таблицы в запросе на основе заданной схемы, используя связи «по умолчанию» [23].

Например, следующий запрос с использованием специфичного для Raima расширения SQL

```
SELECT
  orders.number,
  products.name_prod,
  order_items.amount
FROM
  orders
```

```
  NATURAL JOIN order_items
  NATURAL JOIN products
```

будет эквивалентен стандартному SQL-запросу

```
SELECT
  orders.number,
  products.name_prod,
  order_items.amount
FROM
  orders
  INNER JOIN order_items ON orders.id_order =
order_items.id_order
  INNER JOIN products ON products.id_prod =
order_items.id_prod
```

В рассмотренных случаях у программиста имеется выбор: оставаться лишь в рамках реляционной модели или работать на уровне «родной» для данной СУБД модели данных, прежде всего с целью повышения скорости обработки и выполнения запросов. Оборотной стороной такой универсальности может явиться более низкое быстродействие и ограниченные функциональные возможности по сравнению с СУБД, непосредственно реализующей реляционную модель.

В свете перечисленных возможностей термин NoSQL является бессмысленным в любой своей интерпретации, будь то «не-SQL» или «не только (Not only) SQL»: нереляционные модели, используемые NoSQL-СУБД вполне могут реализовать SQL в качестве входного декларативного языка высокого уровня вместо низкоуровневых манипуляций и навигационного подхода.

Проектирование

«Отличие государственного деятеля от политика в том, что политик ориентируется на следующие выборы, а государственный деятель — на следующее поколение». У. Черчилль

Терминология уровней

В начале книги была упомянута трехуровневая архитектура СУБД ANSI/X3/SPARC. Долгое время она являлась своего рода образцом для проектирования информационной системы в целом, состоящей из СУБД и приложений. Если для 1975 года была характерна ситуация, когда большинство разработчиков занималось вопросами реализации физического уровня хранения, то уже к середине-концу 1980-х появилось достаточное число программных продуктов класса универсальных СУБД, реализующих ту или иную модель данных логического уровня. Соответственно, границы физического уровня заметно сдвинулись, оставляя лишь место для тонкой настройки и потянув за собой и логические рамки. Особенности организации внешнего уровня также стало возможным описывать на логическом, прежде всего с помощью схем (пространств имён), видов (view) и хранимых процедур самой СУБД, а также различными инструментами слоя сервера приложений .

Похожая история случилась и с другим термином. В 1970-х годах для обозначения роли проектирующего и обслуживающего структуры баз данных сотрудника было введено специальное понятие — *администратор баз данных* [5]. Но в настоящее время уже возникло достаточно чуткое разделение на администраторов-проектировщиков и администраторов по эксплуатации. Например, учебные курсы и сертификации Microsoft по СУБД имеют два основных сюжета: реализация и эксплуатация. Компетенции этих специалистов пересекаются, но сложность моделируемых предметных областей, с одной стороны, и обросших тысячами параметров тонкой настройки и мониторинга на конкретных аппаратных платформах с другой, вынуждают к такому разделению труда. Впрочем, руководство некоторых предприятий зачастую уверено, что администратор БД — это тот работник, что делает резервные копии, раздаёт

права доступа, а остальное время генерирует трафик в социальных сетях[15]. Но даже подобный взгляд является прогрессом, ведь в РФ образца 1990х такого сотрудника называли программистом...

Возвращаясь к терминологии проектирования, очень часто в обсуждениях встречается путаница между понятиями *моделей данных* и *моделей баз данных*, являющихся на самом деле *схемами баз данных*. Избежать неоднозначности помогут два простых определения.

Модель данных — это инструмент описания

Схема базы данных — это результат использования инструмента

Взяв в качестве инструмента уже рассмотренные выше реляционную, сетевую или иерархическую модели данных вы получите на выходе соответствующую (или пока ещё не вполне соответствующую) функциональным требованиям схему базы данных.

Определившись с разницей между моделями и схемами данных, можно перейти к разделению на следующие уровни абстракции баз данных: *концептуальный*, *логический* и *физический* [6]. Они соответствуют взглядам на одну и ту же базу данных со стороны разных участников проекта: функционального специалиста, проектировщика БД, администратора БД, программиста, пользователя.

- *Концептуальный уровень* соответствует описаниям сущностей (объектов) предметной области, связям между ними. Используются модели понятийного уровня, семантические модели. Характерная особенность полученных описаний — независимость от используемых моделей данных.
- *Логический уровень* описывает схему базы данных в терминах выбранной модели данных. Скорее всего это будет реляционная модель, но если вы столкнётесь с иным походом, то моделирование на логическом уровне также будет специфичным.

[15] На заре развития Интернет во времена распространения сетей FIDO, узлы которых часто содержали администраторы сетей и БД, ходила такая шутка: «Настоящий фидошник умеет делать 2 вещи: читать почту и пить пиво»

- *Физический уровень* соответствует описанию схемы данных в конкретной СУБД. Одной логической схеме БД может соответствовать несколько физических схем для разных СУБД, реализующих, тем не менее, одну и ту же модель

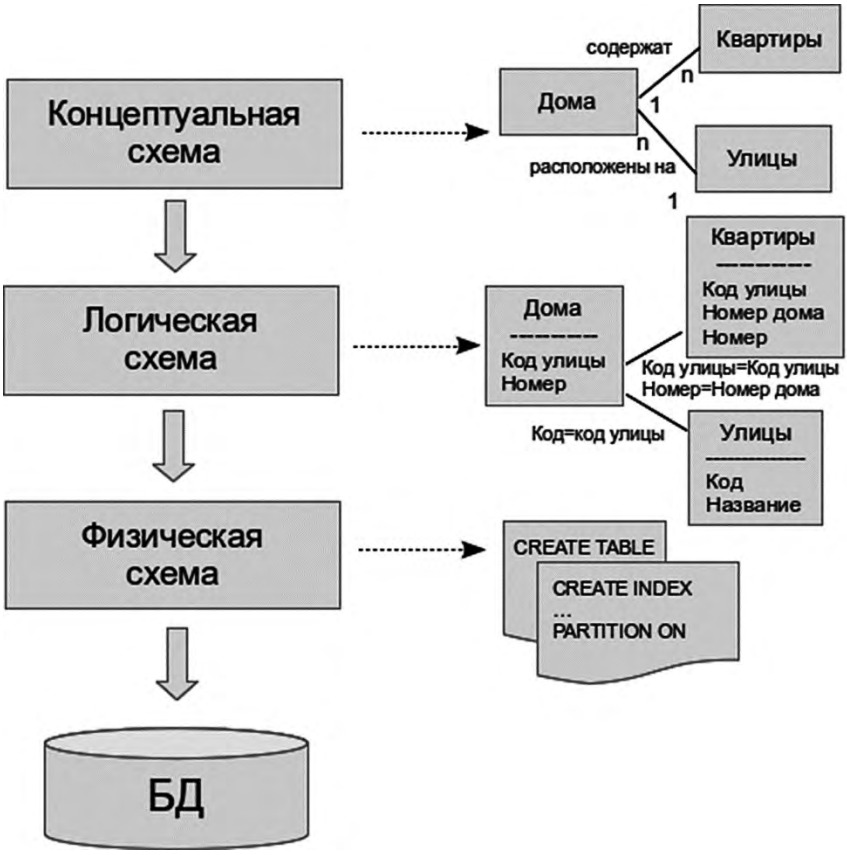

Рис. 17. Три уровня абстракции базы банных

Концептуальный уровень представляет собой наибольший интерес для функционального специалиста-проектировщика. Для разработчика информационной системы важнее результат моделирования на концептуальном уровне, который будет использован в логическом проектировании. Если концептуальные модели отсутствуют, проектировщику придётся работать «стандартным» методом прочтения линейных текстов многостраничных функциональных спецификаций, выстраивая по ним логическую схему БД.

Специфика выбранной модели данных, проявляющаяся уже на логическом уровне, вполне объясняет не слишком нравящееся части программистского сообщества «засилье» реляционных баз данных. Однако, описывая структуры в терминах отношений, ключей, связей, проектировщик может быть уверен, что его результат будет однозначно понят не только непосредственными коллегами, но и другими специалистами. Перейдя же, например, на сетевую терминологию наборов, записей и агрегатов данных, можно встретить не только непонимание изложенной сути, но и неумение программистов приложений оптимальным образом работать с данными в выбранной парадигме, сводящее к нулю предполагаемые выгоды от выбора.

Спускаясь на физический уровень, специфика СУБД становится основным фактором эффективности реализации схемы базы данных вышестоящих уровней. Так реляционная модель на своём логическом уровне оперирует понятиями отношений, атрибутов, ключей. На физическом же уровне РСУБД предлагает программисту пользоваться соответствующими терминами.

Табл. 1. Соответствие некоторых терминов реляционной модели (логический уровень) их реализаций в СУБД (физический уровень)

Термин реляционной модели	**Соответствующие термины РСУБД**
Отношение	Таблица-кластер Таблица-куча Секционированная таблица Табличная функция Вид (view, виртуальная таблица)
Проекция	Вид (view) Материализованный вид Табличная функция
Кортеж	Строка (запись) Строка (с построчным сжатием)
Атрибут	Колонка (столбец) Вычислимая колонка Вычислимая хранимая колонка
Ключ	Первичный ключ Ограничение уникальности (unique) Уникальный индекс

Термин реляционной модели	Соответствующие термины РСУБД
Домен	Встроенный тип Пользовательский тип

Как видно на примере соответствий всего для нескольких терминов, логический уровень значительно отличается от физической реализации в том числе неоднозначностью соответствия. Так в реляционной модели ключом называется минимальный набор атрибутов отношения, позволяющий однозначно идентифицировать кортеж. Ключей, возможно составных, то есть содержащих более одного атрибута, может быть несколько. В РСУБД для каждой таблицы можно определить лишь один ключ, который называется первичным (primary key). Остальные ключи придётся определять соответствующим ограничением целостности или уникальными индексами.

Например, в отношении «Пользователи» веб-сайта кроме первичного ключа в виде внутреннего уникального номера-идентификатора, можно определить такие ключи как «имя регистрации» и «адрес электронной почты». В соответствующей таблице первичным ключом будет идентификатор, а соответствующим колонкам назначены ограничения уникальности.

Некоторые средства СУБД могут быть использованы для реализации разных элементов реляционной модели. Например, вид, как правило, представляет собой проекцию.

```
CREATE VIEW power_users AS
SELECT u.id, u.name
FROM users u
  INNER JOIN users_groups ug ON u.id = ug.id_user
  INNER JOIN groups g ON g.id = ug.id_group
WHERE g.sysname = 'POWER_USERS'
```

Однако, с помощью видов можно создавать и полноценные отношения, находящиеся в режиме «только чтение»

```
CREATE VIEW colors
AS
  SELECT 1 AS id, 'Красный' AS name
  UNION
  SELECT 2 AS id, 'Зеленый' AS name
```

```
  UNION
  SELECT 3 AS id, 'Синий'   AS name
```

Тот же принцип может быть использован и для табличных функций, которые в первом варианте просто являются параметрическими видами.

```
CREATE FUNCTION users_by_group(@sysname nvarchar(16))
RETURNS TABLE
AS
RETURN
(
  SELECT u.id, u.name
  FROM users u
    INNER JOIN users_groups ug ON u.id = ug.id_user
    INNER JOIN groups g ON g.id = ug.id_group
  WHERE g.sysname = @sysname
)
```

Кроме описанной классификации, также существует разделение на *инфологическое* и *даталогическое* моделирование [7] с выработкой соответствующих схем, в целом соответствующих концептуальным и логическим.

Первичные и прочие ключи

Ключ — один из главных элементов реляционной модели. Без него все ваши таблицы превращаются в нагромождение строк на перекрестье столбцов.

Ключ — минимальный набор атрибутов, совокупность значений которых однозначно определяет кортеж в отношении.

Требование к минимальности означает, что из данного набора (множества) атрибутов следует отсеять те, чьи значения в совокупности не влияют на однозначность определения кортежа.

Если перед вами таблица со многими колонками, и одна или более из них содержит в совокупности уникальные значения, значит ключ таблицы состоит их этих колонок. Таких ключей может быть более одного.

Первичный ключ — один из ключей, выбранный в качестве основного.

На практике первичным ключом таблицы выбирают наименьший по количеству входящих в него столбцов.

Много копий сломано в дискуссиях о «правильном» выборе первичных ключей [14,15]. Действительно, выбор неподходящего набора атрибутов или даже их типов на стадии проектирования принесёт дополнительные проблемы в реализации и сопровождении. Остановимся на нескольких важных моментах.

1. Выделение первичного ключа сущности — нетривиальная задача. В реальном мире только искусственно созданные объекты имеют простой и однозначный идентификатор, который можно принять за ключ. Например, номер телефона, почтовый индекс или штрих-код товара. В отношении объектов, созданных природой задача не столь проста и не всегда разрешима в принципе. Поэтому на практике таким объектам присваивают искусственный идентификатор, в простейшем случае представляющий собой просто порядковый номер определённого формата. Например, люди в системе социального страхования имеют генерируемый по определённому алгоритму уникальный номер.

2. Существующие ограничения СУБД накладывают на проектировщика обязанности по предварительной проверке реализуемости создаваемых схем. Если в теории вы можете считать первичным ключом совокупность всех атрибутов отношения, то на практике такой ключ в большинстве случаев не может быть даже создан из-за лимита длины своего значения.

3. Тип и количество атрибутов в первичном ключе также непосредственно влияют на производительность, размер хранимых данных и стоимость внесения изменений. Например, ключ на основе строкового типа в общем случае будет медленнее такового, на базе числового типа даже при одинаковом размере хранения, потому что при сравнении строк учитывается используемая кодировка, регистр и порядок следования символов данной кодовой страницы. Если же ключ содержит более одного атрибута, то во всех связанных

таблицах придётся эти атрибуты дублировать (о внешних ключах мы поговорим ниже).

4. При реализации приложений унификация типа первичного ключа всех таблиц позволяет существенно сократить время разработки за счёт создания типовых модулей обобщения процедур ввода и обработки данных, проверки связей, обработки ошибок, локализации и т.п.

В итоге, наиболее универсальным вариантом для проектировщика транзакционного приложения является выбор искусственного идентификатора числового типа. В простейшем варианте, 32-разрядный целый тип со знаком позволяет иметь 2^{31} = 2 147 483 648 уникальных неотрицательных значений для каждой таблицы, что является достаточным для большинства случаев.

Однако, в условиях распределенной БД, имеющей два и более узла, для подобных ключей возникает проблема генерации глобально-уникальных значений, ведь, например, созданный в БД подразделения №1 клиент с внутренним номером «125» не тот же самый клиент с номером «125», созданным в подразделении №2. На практике эта проблема решается по-разному, вот возможные варианты:

- для каждой БД выделяется непересекающееся множество значений первичного ключа, например, БД 1 генерирует номера 1, 5, 10 и т. д., а БД 2 — 2, 6, 11...;
- к генерируемому значению в качестве старших разрядов числа добавляется номер БД, например, 125 в БД 1 превращается в 10000000125 и 20000000125, что несколько увеличивает размер хранимых данных;
- вместо 32-разрядного целого числа используется так называемый UUID[16], который по своей внутренней структуре представляет собой и, как правило, хранится в виде 128-разрядного целого числа

[16] UUID (Universally Unique Identifier) — открытый стандарт идентификации Open Software Foundation (OSF), используемый для генерации глобально-уникальных идентификаторов в распределенной среде вычислений.

(подробнее о методе хранения UUID следует смотреть в документации по конкретной СУБД);

- идентификатор остаётся локально-уникальным, используется таблица соответствий вида «идентификатор в локальной БД — идентификатор в удалённой БД»;
- смешанные подходы, например, локальный идентификатор для внутренних связей и глобальный, как идентификатор строки при обмене данными.

Каждое из приведённых решений имеет свои преимущества и недостатки. Наиболее общим подходом является использование UUID, однако есть как минимум две причины, по которым такое решение не любят администраторы баз данных:

- значения трудно читаемы человеком, что затрудняет работу с интерактивными запросами, их выводящими. Сравните, например, ключи «201568» и «2344C060-C338-11E3-9C1A-0800200X9B66»;
- размер каждого значения ключа — 128 бит, что в 4 раза больше 32-разрядного целого. Соответственно, для БД с объёмами в миллионы строк увеличивается размер файлов хранения, как и размер индексов, привязанных к значениям кластерного ключа (подробнее см. «Физическая организация памяти»). Для обхода этой проблемы можно объявить ключ на базе UUID некластерным и ввести в таблицу автоинкрементную целочисленную колонку, играющую по сути роль идентификатора строки, создав по ней уникальный кластерный индекс.

Пора подвести некоторые итоги по теме ключей.

Выбор первичного ключа, его тип и унификация будут оказывать большое влияние на дальнейшую разработку приложений. Поэтому если вы рассчитываете на развитие своего проекта и расширение номенклатуры пользователей, то следует подойти к вопросу со всей серьёзностью.

Несмотря на то, что ключи — элемент, присущий реляционной модели, они скорее всего будут в дальнейшем использоваться в качестве универсальных идентификаторов объектов в ваших приложениях.

Разница между так называемыми «естественными» и «суррогатными» ключами достаточно условна, зачастую «естественный» ключ является автоматически генерируемым по определённому формату и алгоритму «суррогатом» во внешней по отношению к вам системе, например в БД налоговой или пенсионной службы. Не стоит тратить время на выяснение вопросов, ответ на которые напрямую зависит от философской позиции авторов. Разумнее придерживаться практик, которые могут обеспечить максимальную гибкость и простоту при внесении изменений в систему, несмотря на то, что любая БД — наиболее консервативный её компонент.

Внешние ключи и связи

Связь между таблицами в реляционной модели реализуется внешним ключом.

> *Внешний ключ* — множество колонок подчинённой таблицы, соответствующее ключу главной таблицы

Я намеренно опустил уточняющее прилагательное «первичный», потому что связь в реляционной модели может проходить по любому ключу. Многие СУБД поддерживают декларативную ссылочную целостность, позволяющую определить внешний ключ со ссылкой не только на первичный ключ таблицы, но и на любой другой набор её колонок, объявленных уникальными.

В приведённом примере связь между таблицами профилей (profiles) и сообщений (messages) осуществляется по первичному ключу. Связь является обязательной (mandatory).

> *Обязательная связь* — строки в подчинённой таблице не могут существовать без соответствующих строк в главной таблице

Обязательная связь реализуется объявлением соответствующего столбца внешнего ключа как не допускающего неопределённых значений NOT NULL.

С таблицей очереди отправки (mailing_queue) связь осуществляется по ключу «адрес электропочты»; соответствующий столбец объявлен уникальным. Кроме того, вторая связь является ключевой.

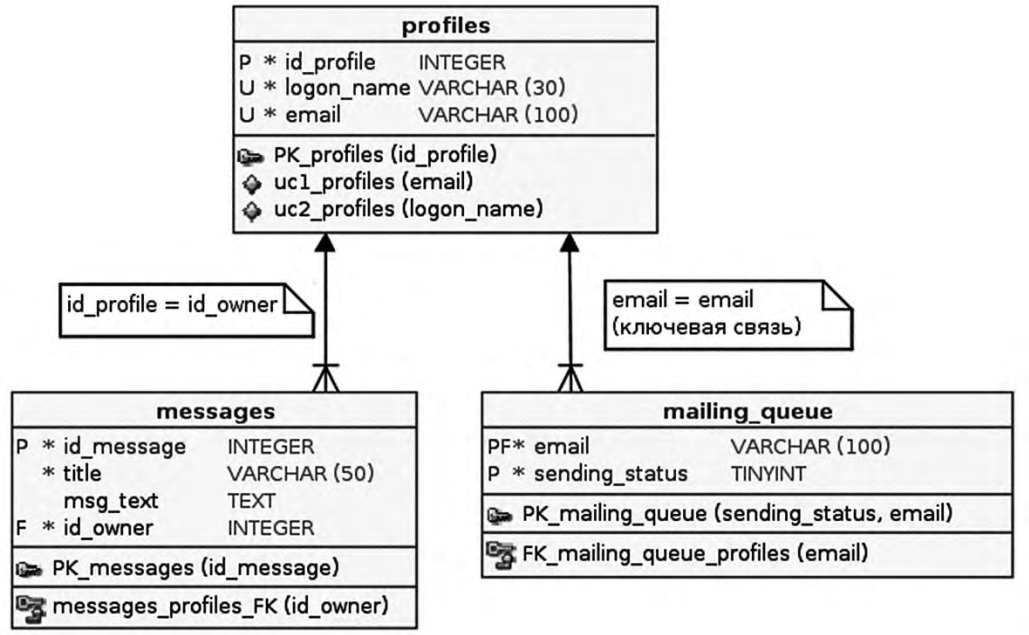

Рис. 18. Примеры связей

Ключевая связь — связь, при которой внешний ключ таблицы входит в состав одного из её ключей.

Из этого определения следует также, что ключевая связь всегда является обязательной.

Ключевая связь на практике встречается часто, например, таблица жилых домов имеет ключ в виде идентификатора улицы и номера дома. Таким образом, между улицами и расположенными на них домами существует ключевая связь, даже если вы используете для домов суррогатный автоинкрементный первичный ключ, а колонки «Улица» и «Номер дома» забыли пометить ограничениями уникальности.

Если представить приведённую схему в виде SQL-скрипта, то в реализации SQL Server получим следующий код.

```
CREATE TABLE mailing_queue (
    email          VARCHAR (100) NOT NULL,
    sending_status TINYINT NOT NULL,
    CONSTRAINT PK_mailing_queue
      PRIMARY KEY CLUSTERED (sending_status, email)
)
```

```
GO

CREATE TABLE messages (
    id_message INTEGER NOT NULL,
    title      VARCHAR (50) NOT NULL,
    msg_text   VARCHAR(MAX),
    id_owner   INTEGER NOT NULL,
    CONSTRAINT PK_messages PRIMARY KEY (id_message)
)
GO

CREATE TABLE profiles (
    id_profile INTEGER NOT NULL,
    logon_name VARCHAR (30) NOT NULL,
    email      VARCHAR (100) NOT NULL,
    CONSTRAINT PK_profiles PRIMARY KEY (id_profile)
)
GO

ALTER TABLE mailing_queue
  ADD CONSTRAINT FK_mailing_queue_profiles
  FOREIGN KEY (email)
  REFERENCES profiles (email)
GO

ALTER TABLE messages
  ADD CONSTRAINT FK_messages_profiles
  FOREIGN KEY (id_owner)
  REFERENCES profiles (id_profile)
GO
```

Нормализация и денормализация

Нормализация схемы реляционной БД оказывает существенное влияние буквально на все аспекты взаимодействия с БД: от затрат на модификацию структур и данных до производительности запросов приложений и хранимых объёмов информации. В ряде случаев структуры могут быть сознательно денормализованы, что созвучно с другим словом «деморализованы». Однако, следует хорошо понимать, с какой целью это было сделано и полностью отдавать себе отчёт о последствиях. В общем же случае безопаснее всего придерживаться простого правила:

Нормализация — не догма, но чтобы её нарушать, нужны основания

На практике проектирования схем баз данных достижение третьей нормальной формы (3НФ) считается достаточным условием для большинства случаев.

Чему служат нормальные формы проще всего понять на примерах.

1НФ

Первая нормальная форма (1НФ) выполняется, если все значения атрибутов (читай, колонок таблицы) атомарны, то есть неделимы.

Собственные типы данных СУБД считаются атомарными, исключение могут составлять массивы, в том числе символьные (текстовые) и байтовые. Следует также понимать, что атомарность может быть относительна выбранного взгляда со стороны предметной области и контекста. Например, телефонный номер в базе данных маркетинга содержится в одной колонке, тогда как у телефонных операторов он разделяется на номера АТС, шлейфов и т.п. Колонки для хранения комментариев, подлежащих последующей обработке приложением, также отчасти нарушают принцип атомарности.

По этой же причине не стоит рассматривать отдельно целую и дробные части действительного числа или даже пару «дата-время»: дальнейшая детализация не имеет смысла с точки зрения моделируемой области, где они атомарны.

Предположим, мы нарушили 1НФ и стали хранить фамилии, имена и отчества клиентов в одной колонке. Пока операторы вносили информацию, эта ошибка проектирования особенно не мешала, Однако, на следующем этапе понадобилась отчётность, в которой ФИО клиентов выводились бы в виде фамилии и инициалов. Оказалось, что некоторые записи вместо «Сидоров Петр Иванович» содержат «Петр Иванович Сидоров», в других отчества нет вовсе, в третьих фамилия двойная и не всегда записана через тире, в четвёртых после фамилий расставлены запятые... Эту проблему пришлось решать программированием совсем нетривиальной логики с элементами распознавания по словарю. Было потрачено много времени и

средств, но в отчётности нет-нет да и проскакивали непонятные значения типа «Оглы П.Б.Б.».

Следует отметить, что при добавлении к этому учёту клиентов-иностранцев, проектировщиков логической схемы БД не спасла бы и более структурированная форма из трёх колонок для раздельного хранения фамилий, имён и отчеств. Потому что это проблема уровня концептуального проектирования и соответствующих моделей: необходим синтез не привязанной к модели данных структуры, способной вмещать в себя комбинации имён людей разных стран и культур.

2НФ

Вторая нормальная форма (2НФ) означает, что выполнены требования 1НФ, при этом все атрибуты целиком зависят от составного ключа и не зависят ни от какой его части.

На первый взгляд кажется, что нарушения 2НФ практически невозможны, потому что чаще всего в качестве первичных ключей используются автоинкрементные целочисленные значения или иные суррогаты для реализации ссылок. Однако, в определении говорится о ключах вообще, а не только о первичных. В отношении может быть несколько ключей, и некоторые из них могут являться составными. Такие ключи следует подвергнуть проверке в первую очередь.

Ассоциативная таблица — таблица, имеющая **ключевые** связи с двумя и более таблицами

Например, если каждая операция сбыта мебельной продукции в таблице продаж однозначно характеризуется колонками идентификатора товарной позиции, даты продажи и идентификатором покупателя, то нахождение в той же таблице столбца «Тип материала», зависящего непосредственно от товарной позиции, должно немедленно привлечь ваше внимание.

Аномалия в данном случае приведёт только к избыточности хранения в виде размера идентификатора, помноженного на число строк таблицы (без учёта индексов). Но если в той же таблице обнаружится ещё и колонка

«Контактный телефон», присущая атрибутике покупателя, то последствия окажутся более серьёзными. Кроме избыточности хранения при ошибке ввода придётся исправлять номер телефона во всех записях о продажах данному покупателю.

Кроме приведённых примеров, при наличии в таблицах нескольких ключей необходимо, с позиций логики предметной области, определить, являются ли эти ключи присущими данной сущности или же они суть внешние ключи другой сущности, пока ещё не выделенной в процессе проектирования.

3НФ

Третья нормальная форма (3НФ) означает, что выполнены требования 2НФ, при этом в между атрибутами отношения нет транзитивных зависимостей.

Что такое транзитивная зависимость легко понять на примере уже упоминавшейся выше таблицы продаж — типичного примера ассоциативной таблицы.

Предположим, что продажа каждой товарной позиции имеет своим основанием документ (заказ, счёт и т.д.), а её стоимость характеризуется ценой, количеством и валютой. В этом случае имеем следующие зависимости между атрибутами (колонками):

- «Идентификатор продажи» → «Номер документа»
- «Идентификатор продажи» → «Код валюты»
- «Номер документа» → «Код валюты»

Эти зависимости транзитивны: каждая продажа однозначно определяет свой документ-основание и расчётную валюту, однако, валюта определяется ещё и документом.

Результатом нарушения 3НФ является избыточность хранения и необходимость обновления данных в связанной таблице. Так, если вы оставите колонку «Код валюты» в таблице продаж, то при изменении

валюты документа придётся также обновлять все связанные с ним строки продаж.

Деморализуем... то есть денормализуем: «звезда» и «снежинка»

Как можно понять из вышеприведённых примеров, основными целями нормализации являются:

- устранение избыточности при хранении данных, приводящей к увеличению размера БД;
- исключение необходимости модификации данных в связных таблицах для минимизации времени и операций, проводящихся в одной транзакции. Или, как выражаются специалисты, уменьшить *толщину транзакции*, потому что толстые транзакции мешают при многопользовательской работе взаимными блокировками и увеличением времени отклика системы. Речь об этом пойдёт в отдельной главе.

Но список заявленных целей касается приложений транзакционных.

В приложениях интерактивной аналитической обработки приоритет меняется: на первый план выходит время отклика системы, в ущерб которому данные могут быть избыточны.

Зачем это нужно?

Наиболее дорогостоящая с точки зрения вычислительных ресурсов операция между большими таблицами — соединение. Соответственно, если в одном запросе необходимо «провентилировать» несколько таблиц, состоящих из многих миллионов строк, то СУБД потратит достаточно много времени на такую обработку. Пользователь в это время может отойти выпить кофе. Интерактивность обработки практически исчезает и приближается к таковой для обработки пакетной. Даже хуже, в пакетном режиме пользователь с утра получает все запрошенные накануне данные и спокойно работает с ними, подготавливая новые запросы к вечеру.

Чтобы избежать ситуации тяжёлых соединений таблицы денормализуют. Но не абы как. Существуют некоторые правила, позволяющие считать

денормализованные с точки зрения транзакционной обработки таблицы «нормализованными» согласно правилам построения таблиц для хранилищ данных.

Основных схем, считающихся «нормальными» в аналитической обработке, две: «снежинка» и «звезда». Названия хорошо отражают суть и следуют непосредственно из картинки связанных таблиц.

В обоих случаях центральным элементом схемы являются так называемые *таблицы фактов*, содержащие интересующие аналитика события, транзакции, документы и другие занятные вещи. Но если в транзакционной БД один документ «размазан» по нескольким таблицам (как минимум по двум: заголовки и строки-содержание), то в таблице фактов одному документу, точнее, каждой его строке или набору сгруппированных строк, соответствует одна запись. Сделать это можно денормализацией двух вышеупомянутых таблиц.

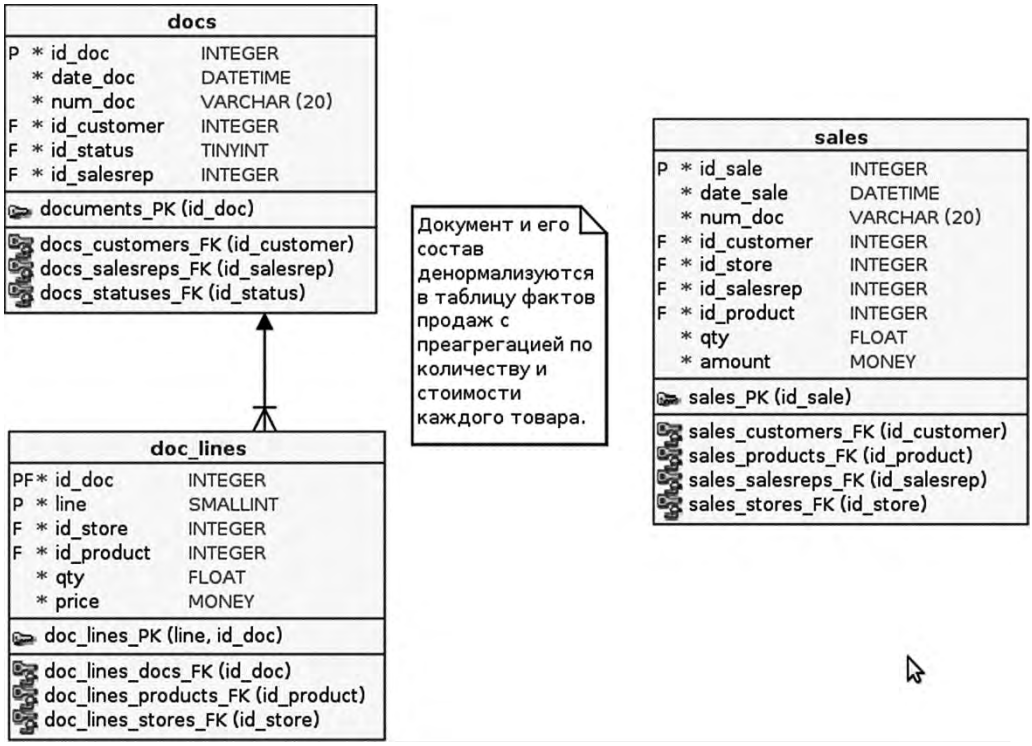

Рис. 19. Денормализация документов в таблицу фактов

Теперь можно оценить, насколько облегчится для выполнения СУБД запрос, например, следующего вида: определить объёмы продаж муки клиентам «ООО Пирожки» и «ЗАО Ватрушки» за период.

В нормализованной транзакционной БД

```
SELECT
  SUM(dl.qty) AS total_qty, SUM(dl.price) AS total_amount,
c.name
FROM
  docs d
  INNER JOIN doc_lines dl ON d.id_doc = dl.id_doc
  INNER JOIN customers c ON d.id_customer = c.id_customer
  INNER JOIN products p ON dl.id_product = p.id_product
WHERE
  c.name IN ('ООО Пирожки', 'ЗАО Ватрушки') AND
  p.name = 'Мука' AND
  d.date BETWEEN '2014-01-01' AND '2014-02-01'
GROUP BY c.name
```

В аналитической БД

```
SELECT
  SUM(s.qty) AS total_qty, SUM(s.amount) AS total_amount,
c.name
FROM
  sales s
  INNER JOIN customers c ON d.id_customer = c.id_customer
  INNER JOIN products p ON dl.id_product = p.id_product
WHERE
  c.name IN ('ООО Пирожки', 'ЗАО Ватрушки') AND
  p.name = 'Мука' AND
  s.date BETWEEN '2014-01-01' AND '2014-02-01'
GROUP BY c.name
```

Вместо тяжёлого соединения между двумя таблицами документов и их состава с миллионами строк, СУБД достаётся прямая работа с таблицей фактов и лёгкие соединения с небольшими вспомогательными таблицами, без которых также можно обойтись, зная идентификаторы.

```
SELECT
  SUM(s.qty) AS total_qty, SUM(s.amount) AS total_amount,
s.id_customer
FROM
  sales s
WHERE
```

```
  s.id_customer IN (1025, 20897) AND
  s.id_product = 67294 AND
  s.date BETWEEN '2014-01-01' AND '2014-02-01'
GROUP BY s.id_customer
```

Вернёмся к схемам «звезда» и «снежинка». За кадром первого рисунка остались таблицы клиентов, их групп, магазинов, продавцов и, собственно, товаров. При денормализации эти таблицы, называющиеся *измерениями*, также соединяются с таблицей фактов. Если *таблица фактов* ссылается на *таблицы-измерения*, имеющие ссылки на другие измерения (измерения второго уровня и выше), то такая схема называется «снежинка».

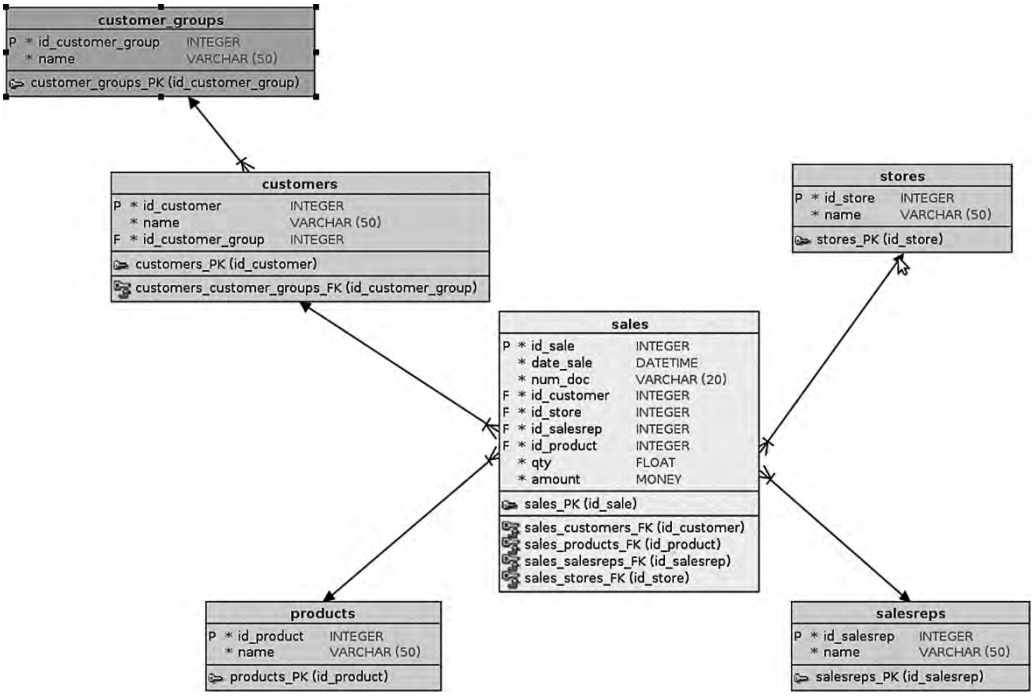

Рис. 20. Таблица фактов в схеме «снежинка»

Как можно заметить, для запросов, включающих фильтрацию по группам клиентов, приходится делать дополнительное соединение.

```
SELECT sum(amount)
FROM sales s
     INNER JOIN customers c ON s.id_customer = c.id_customer
WHERE c.id_customer_group IN (1, 2, 10, 55)
```

В таком случае денормализацию можно продолжить и опустить измерение второго уровня на первый, облегчив запросы к таблице фактов.

Схема, в которой таблица фактов ссылается только на измерения, не имеющие второго уровня, называется «звезда». Число таблиц измерений соответствует числу «лучей» в звезде.

Схема «Звезда» полностью исключает иерархию измерений и необходимость соединения соответствующих таблиц в одном запросе.

```
SELECT sum(amount)
FROM sales s
WHERE s.id_customer_group IN (1, 2, 10, 55)
```

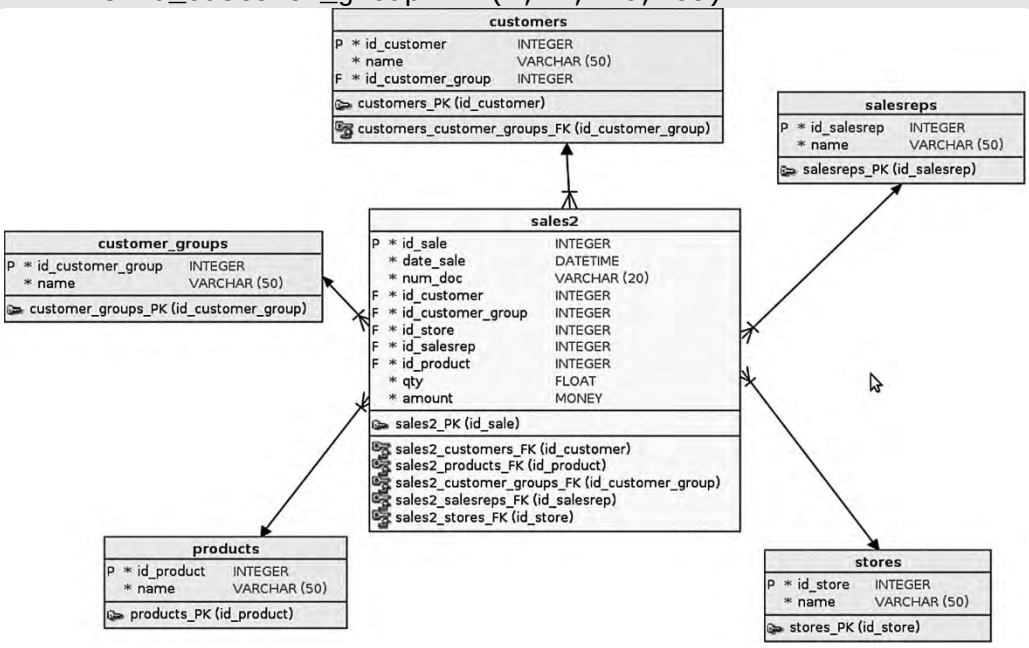

Рис. 21. Таблица фактов в схеме «звезда»

Обратной стороной денормализации всегда является избыточность, являющаяся причиной увеличения размера БД как в случае транзакционных, так и аналитических приложений. Давайте посчитаем примерную дельту на приведённом выше примере преобразования «снежинки» в «звезду».

Положим, таблица продаж не использует компрессию данных и содержит около 500 миллионов строк, а количество групп покупателей порядка 1000. В этом случае мы можем использовать в качестве типа идентификатора `id_customer_group` короткое целое (shortint, smallint), занимающее 2 байта.

В некоторых СУБД, например Oracle, специальные целочисленные типы на уровне определений схемы БД отсутствуют, необходимо использовать универсальный логический тип numeric(N), где N — число хранимых разрядов. Размер хранения такого числа рассчитывается по специальной формуле, приводимой в документации по физическому хранению данных, и, как правило, он превышает таковой для низкоуровневых типов вроде «16-битное целое» на 1-3 байта.

Будем считать, что наша СУБД поддерживает двухбайтовый целочисленный тип (например, PostgreSQL, SQL Server, Sybase и другие). Тогда добавление соответствующей колонки `id_customer_group` в таблицу продаж вызовет увеличение её размера как минимум на 500 000 000 * 2 = 1 000 000 000 байт ≈ 1 гигабайт.

Универсальных рекомендаций по денормализации не существует, это всегда компромисс между размером БД и временем выполнения запросов. Если вы исчерпали все возможные способы оптимизации запросов и физического хранения на данной схеме БД, то следует рассмотреть возможность её дальнейшей денормализации, что, однако, не является единственным путём решения проблем производительности системы. Более распространённый способ — организация на основе хранилища данных ещё более агрегированных таблиц (витрин) и многомерных кубов, которые и будут непосредственно служить базами данных для запросов пользователей.

Типовая архитектура данных аналитических приложений

Особенность аналитической обработки — тяжёлые запросы по произвольным критериям выборки. Сколь ни старались бы вы оптимизировать хранилище данных, всегда найдутся запросы, выполняющиеся медленнее, чем требуется.

Поэтому проектировщики аналитических систем пришли к идее разделения собственно *хранилища* (data warehouse), где данные представлены в минимально агрегированном виде, и конечных БД, адаптированных под нужды пользователей разных профилей и предметных областей.

В роли баз данных конечных пользователей могут выступать как вполне реляционные *витрины данных* (datamart), так и другие типы СУБД, прежде всего, основанные на многомерных моделях.

Витрина данных представляет собой реляционную БД или даже одну таблицу фактов, содержащую более агрегированную и отфильтрованную информацию, извлечённую непосредственно из хранилища данных.

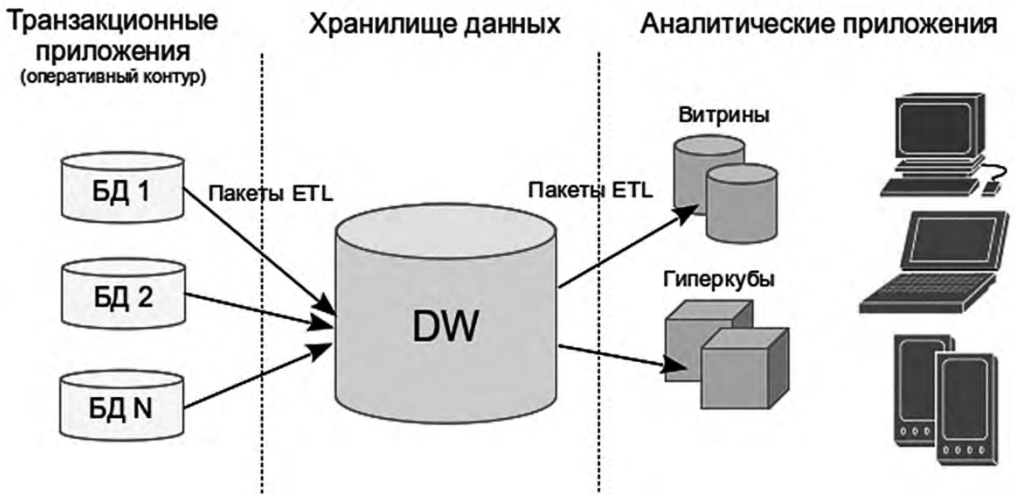

Рис. 22. Логическая архитектура аналитических приложений

Например, реальная схема «звезда» из предыдущей главы должна содержать десятки измерений, в том числе определяющие географию продаж и классификацию товаров. В этом случае имеет смысл извлекать информацию по конкретным регионам и типам товаров, помещая её в витрины данных пользователей соответствующих подразделений. Торговые представители в данной стране не нуждаются в информации о продажах на материках другой стороны глобуса. Агрегированная и отфильтрованная таблица продаж соответствующей витрины может содержать на порядки меньше записей, чем исходная таблица хранилища.

Такой подход облегчает задачу достижения приемлемой производительности интерактивных запросов пользователей при минимальной денормализации. Оборотная сторона — увеличение количества баз данных, требующих администрирования. Возрастают

расходы на разработку и поддержание в актуальном состоянии многочисленных пакетов ETL[17] для переноса данных из одной БД в другую.

Как и в случае денормализации таблиц, декомпозиция единой БД хранилища на многочисленные витрины и кубы — предмет компромисса между требуемой производительностью и накладными расходами на содержание более сложной децентрализованной инфраструктуры.

Переносимость между СУБД

Сюжет переносимости приложений между разными СУБД является с технологической точки зрения в большей степени предметом программирования, нежели проектирования. Тем не менее, о некоторых проблемах следует подумать гораздо раньше, чем будут написаны первые строки кода и отлажены первые запросы.

Есть достаточно распространённое мнение, источником которого являются прежде всего, специалисты по эксплуатации конкретных СУБД, знающие цену интеграции, тонкой настройки и оптимизации физического уровня. Суть мнения состоит в следующем высказывании: «Невозможно разработать приложение, эффективно работающее с разными СУБД».

Не привязываясь к размытости термина «эффективно», следует признать, что это вполне логичная интерпретация другого известного в системной инженерии факта «универсальная система менее эффективна, чем специализированная». Да, универсальная ЭВМ может быть гораздо менее производительна, чем специализированная. Однако, мы все в большинстве своём работаем на универсальных персональных ЭВМ, а не на спецвычислителях, имеющих свои области применения.

С другой стороны, вы уже знаете, что миры реляционной модели данных и промышленных РСУБД хоть и пересекаются, но далеко не на 100 %. Таким образом, задача переносимости имеет место, и решать её надо, даже

[17] От англ. Extract-Transform-Load (ETL), программа для извлечения, преобразования и загрузки данных между их источником и приёмником. Программа обычно работает в пакетном режиме, поэтому ETL часто называются пакетами.

заранее зная, что система будет в чем-то уступать аналогу, привязанному к конкретной СУБД.

Основных подходов к проектированию переносимых приложений баз данных несколько:

- максимальное абстрагирование от СУБД;
- абстрагирование от входного языка СУБД;
- использование подмножества входного языка, реализованного в целевых СУБД.

Ниже мы кратко рассмотрим перечисленные подходы, но перед этим следует понимать, что задача переносимости между **любыми** СУБД абсурдна по своей постановке и не должна стоять перед проектировщиками в принципе, как и задача создания вечного двигателя второго рода. Необходимо в самом начале зафиксировать, во-первых, модель данных, а во-вторых, некоторый ограниченный список СУБД, которые предполагается поддержать.

Абстрагирование от СУБД

Суть подхода состоит в использовании иной модели логического уровня, которая может быть спроецирована на реляционную и обратно без значимых потерь в семантике (смысловой начинке). В связи с массовым распространением в 1980-90-х годах объектно-ориентированного подхода в программировании, за такую модель была взята объектная.

Слой абстракции объектно-ориентированного приложения от СУБД реализуется средствами объектно-реляционной проекции (ОРП), описанными в одной из предшествующих глав.

Из недостатков объектной модели в применении к данной задаче наибольшие неудобства вызывает слабая или даже отсутствующая стандартизация и вытекающая отсюда привязка к конкретным реализациям ООП в разных средах программирования. Тем не менее, если оставаться в рамках одной среды, например, Java или .Net, то задачу абстрагирования можно решить в частном случае, выделив при этом некоторые общие подходы.

В целом, подход ОРП показывает работоспособность для решения задачи переносимости. Однако, обеспечение приемлемой производительности вне логики CRUD является сложной задачей, требующей хорошего понимания особенностей работы целевых СУБД и знания SQL разработчиками на продвинутом (advanced) уровне. ОРП легко позволяет скрыть некомпетентность программиста в области баз данных, поэтому создание приложений, оптимально работающих с СУБД по сравнению с непосредственным доступом становится более сложным процессом, требующим постоянного контроля качества кода, порождающего SQL.

Известным примером коробочного продукта, реализующего подход, является платформа создания учётных систем «1С».

Абстрагирование от входного языка СУБД

В данном подходе необходимо создать собственный язык определения и манипуляции данными и его отображение на входные языки целевых СУБД. Подобная задача в общем виде не является тривиальной и находится на уровне научно-исследовательской работы. Для упрощения реализации устанавливаются следующие ограничения на целевые СУБД:

- являются реляционными;
- поддерживают SQL базового уровня и некоторый набор встроенных функций (например, манипуляции с датами, строками, математические).

Введённый входной язык представляет собой расширенное макроопределениями общее для целевых СУБД подмножество SQL и встроенных функций.

Преимущество макропроцессора по сравнению с полноценным транслятором состоит в относительной простоте реализации или даже в возможности использовать готовые инструменты с открытым исходным кодом, например m4, стандартный для Linux-среды. Недостатки также известны: ошибки уровня компиляции, отлавливаемые транслятором, в случае макроопределений обнаруживаются позднее, уже на стадии трансляции языка целевой платформы.

Например, следующий запрос содержит расширения входного языка в виде двух макроопределений `IsTrue` и `Substr`. Макрос `IsTrue` позволяет оперировать булевыми значениями, а `Substr` раскрывается в функцию выделения подстроки заданного размера начиная с указанной позиции.

```
SELECT *
FROM tasks
WHERE IsTrue(completed) AND Substr(name, 5, 3) = 'SYS'
```

В случае SQL Server, для представления булевых значений имеется тип данных `bit`, где 0 кодирует «ложь», а 1 - «истину». СУБД поддерживает и встроенную функцию `substring`. Тогда расширенный оператор SELECT будет раскрыт следующим образом.

```
SELECT *
FROM tasks
WHERE completed = 1 AND substring(name, 5, 3) = 'SYS'
```

В СУБД Oracle в качестве типа для представления булевых значений принято использовать односимвольный `varchar2(1)` со значениями 'Y' и 'N'. Функция извлечения подстроки имеет аналогичный набор параметров, но называется `substr`. Соответственно, запрос на макроязыке преобразуется в отличающийся от того, что использовался для SQL Server.

```
SELECT *
FROM tasks
WHERE completed = 'Y' AND substr(name, 5, 3) = 'SYS'
```

СУБД PostgreSQL поддерживает встроенный тип данных `boolean` для булевых значений, а функция `substring` имеет значительно отличающуюся от предыдущих случаев сигнатуру.

```
SELECT *
FROM tasks
WHERE completed = true AND substring(name from 5 for 3) =
'SYS'
```

Более подробно технология и примеры использования расширенного входного языка рассмотрены в главе, посвящённой модульному тестированию.

Использование подмножества входного языка

Наиболее простой подход по критерию его реализации в приложениях. Собственно, никаких дополнительных слоёв абстракций для доступа к данным не требуется. Однако, простота реализации оборачивается сложностями программирования запросов к СУБД, ограниченных только подмножеством входного уровня (entry level) стандарта SQL-92.

На практике обойтись таким минимальным языком трудно, приходится сознательно ограничивать обработку запроса средствами СУБД, например, не используя встроенных функций, завершая её уже в приложении над полученным набором данных. Для преодоления некоторых несовместимостей в форматах и синтаксисе запросов могут помочь унифицированные средства доступа к СУБД, такие как ODBC, JDBC, ADO.Net, разнообразные DAC (Data Access Components) для Delphi и FreePascal. Могут быть полезны escape-последовательности ODBC, обязательно использование параметров в запросах вместо текстовых констант. Компоненты должны обладать функцией обработки «живых» запросов, позволяющей модифицировать таблицы на основе метаданных без каких-либо явных манипуляций с SQL.

Однако, ограничения подмножества SQL являются причиной усложнения кода запросов и обработки средствами приложения. Поэтому данный способ можно рекомендовать только в достаточно простых программах обрабатывающих небольшие объёмы данных.

Типовые структуры

В этой главе мы рассмотрим случаи, часто встречающихся в практике разработчика информационных систем. Однако, предлагаемые решения не являются шаблонными в традиционном понимании «взял и прилепил», выбор конкретного способа реализации должен совершаться осознанно согласно имеющимся требованиям.

Моделирование связей разных типов

Реляционная модель непосредственно позволяет определить только связи типа «один-ко-многим». Для этого используется внешний ключ. Каким образом можно отобразить связи других типов?

Связь «многие-ко-многим»

Связи «многие-ко-многим» моделируют при помощи двух связей «один-ко-многим». Для реализации в схему вводится дополнительная таблица, содержащая внешние ключи на таблицы, между которыми устанавливается связь.

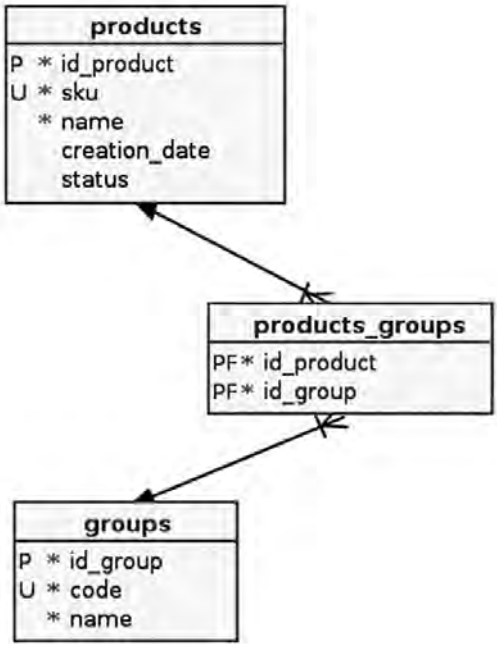

Рис. 23. Связь «многие-ко-многим»

Таблица связей может иметь и собственный суррогатный первичный ключ, если это необходимо, например, для унификации доступа в приложении или для уменьшения размера кластерного ключа. Но в простейшем случае достаточно определить два внешних ключа в качестве первичного, как показано на схеме.

Если таблица связей должна иметь собственные атрибуты или отношения с более чем двумя таблицами, то речь идёт о *таблице ассоциации*,

являющейся отображением сущности. Такое изменение может возникнуть при уточнении модели, в этом случае наличие собственного первичного ключа облегчит реструктуризацию, сведя её лишь к добавлению новых столбцов.

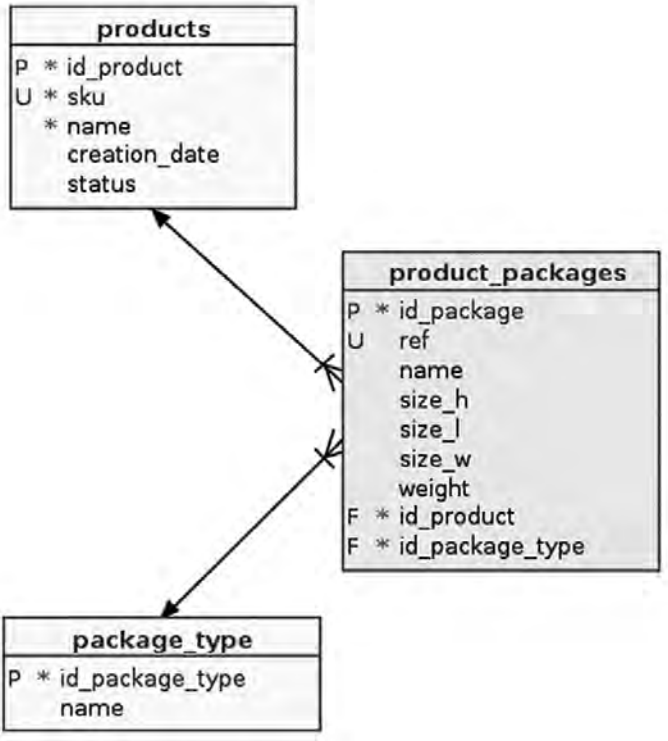

Рис. 24. Таблица ассоциаций

Связь «один-к-одному»

Для реализации связи «один-к-одному» применяются два взаимодополняющих способа. В первом случае решение достаточно очевидное: мы используем внешний ключ, на который наложено дополнительное ограничение уникальности.

В частном случае внешний ключ может быть одновременно и первичным, например, при моделировании наследования типов (классов). Однако, данный способ имеет и недостатки.

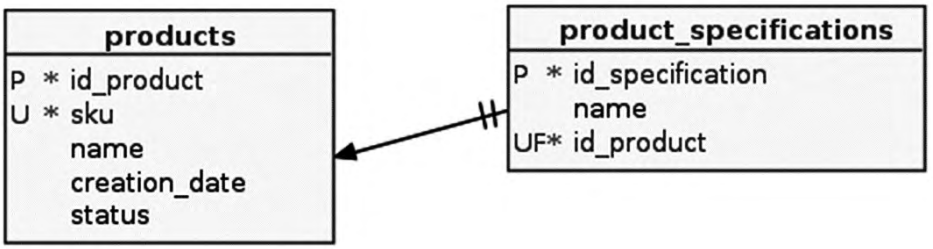

Рис. 25. Связь «один-к-одному» посредством уникального внешнего ключа

Во-первых, для объявления внешнего ключа уникальным связь должна быть обязательной, и тогда соответствующая колонка таблицы декларируется как NOT NULL. В приведённом примере связь действительно обязательна, что наиболее часто встречается на практике, и спецификация не может существовать без продукта. Но между местом в команде и спортсменом такая связь может быть необязательной в случае игры в неполном составе.

Обойти данное ограничение можно, если вместо декларативного определения колонки NOT NULL реализовать делающий соответствующие проверки триггер. Вот пример для PostgreSQL.

```
CREATE TABLE players
(
  num_player integer NOT NULL,
  name character varying(50),
  CONSTRAINT pk_players PRIMARY KEY (num_player)
);

CREATE TABLE places_in_team
(
  num_place integer NOT NULL,
  num_player integer NULL,
  CONSTRAINT pk_places_in_team PRIMARY KEY (num_place),
  CONSTRAINT fk_players_places_in_team
    FOREIGN KEY (num_player)
    REFERENCES players(num_player)
);

CREATE OR REPLACE FUNCTION places_in_team_check_num_player()
  RETURNS trigger AS
```

```
$$
DECLARE
  num_place integer;
BEGIN
  IF EXISTS(
    SELECT 1 FROM places_in_team
    WHERE num_player = NEW.num_player) THEN
      RAISE 'Игрок % уже задействован', NEW.num_player;
  END IF;
  RETURN NEW;
END;
$$
LANGUAGE plpgsql;

CREATE TRIGGER tg_places_in_team_check_num_player
BEFORE INSERT OR UPDATE
ON places_in_team
FOR EACH ROW
EXECUTE PROCEDURE places_in_team_check_num_player();
```

Протестируем триггер

```
INSERT INTO players (num_player, name) VALUES (1, 'Иванов');
INSERT INTO players (num_player, name) VALUES (2, 'Сидоров');
INSERT INTO places_in_team (num_place, num_player) VALUES (1,
1);
-- игрок не назначен
INSERT INTO places_in_team (num_place, num_player) VALUES (2,
NULL);
-- выдаст ошибку, т.к. игрок уже назначен на первую позицию
INSERT INTO places_in_team (num_place, num_player) VALUES (3,
1);

********** Error **********
ERROR: Игрок 1 уже задействован
```

Во-вторых, соединение проводится всегда только по одному и тому же внешнему ключу, id_product таблицы product_specification в примере. В случае же необязательной связи соединение выгоднее проводить по внешнему ключу таблицы, имеющей меньшее число строк.

Второй способ реализации связи «один-к-одному» состоит из двух перекрёстных ссылок «один-ко-многим», что вносит некоторую избыточность, но позволяет обойти указанный недостаток.

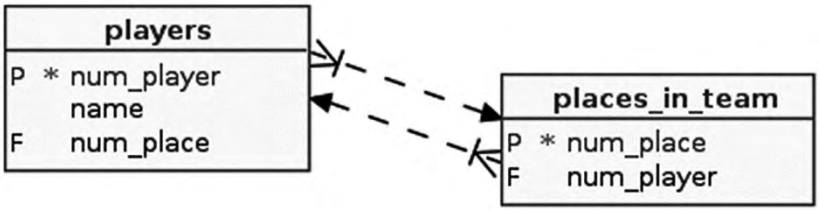

Рис. 26. Связь «один-к-одному» посредством перекрёстных ссылок

Как следует из первого способа, для необязательных связей необходимо реализовать проверку целостности триггером или, что менее эффективно, на уровне приложения. Таким образом, вам придётся программировать две функции проверки вместо одной. Для транзакционного приложения это очевидная плата за избыточность, приводящая к снижению скорости вставки новых и модификации имеющихся записей. Однако, в случае аналитической обработки данные находятся в режиме «только чтение», проверки не требуются, тогда как оптимизация скорости соединения таблиц может быть более существенным фактором.

Хронологические данные

Примеры, когда необходимо помнить историю изменений или регистрировать последовательности событий, мы можем найти в самых разных областях: архив документов, история изменения цен или котировок, журнал событий (аудит), журнал хозяйственных операций, протоколы измерений эксперимента (показания датчиков) и т. д.

Можно выделить два наиболее общих случая, для которых требуется использование временных рядов:

- изменение во времени состояния объекта;
- регистрация событий, происходящих с объектом.

В первом случае перед разработчиком стоит задача хранить историю изменения объекта, например, документа, чтобы иметь возможность

восстановить его состояние в заданный момент времени. Кроме собственно организации структур данных, рассматриваемых в рамках статьи, необходимо создать целую подсистему, основанную на принципах документооборота. Наиболее критичным при этом будет время отклика системы для получения проекций документов на определённый момент времени в оперативном режиме.

Во втором случае требуется хранить и восстанавливать историю действий, связанных с объектом. Как правило, эти данные предназначены для последующей аналитической обработки, поэтому здесь более важной будет скорость первичного сбора информации, прежде всего, вставки новых записей.

Перечисленные действия не являются взаимоисключающими. Например, хранение истории изменения атрибутов документа не исключает ведения журнала произведённых с ним операций. В технической литературе также иногда встречается термин «темпоральные базы данных» (temporal database), но он относится только к первому случаю.

Способ 1. Хранение даты изменения

Наиболее простой способ организации временных рядов кажется очевидным: к имеющемуся ключу (идентификатору объекта) нужно добавить ещё одну колонку — дату изменения объекта.

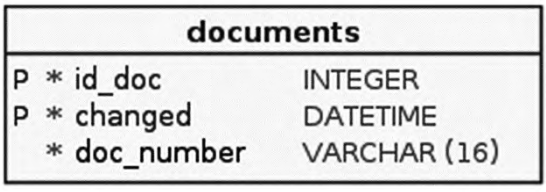

Рис. 27. Способ хранения даты изменения

Столбец changed (дата изменения) имеет здесь простой физический смысл — обозначение момента, когда произошло изменение состояния объекта, например атрибута документа. Промежуток между этой датой и датой следующего изменения будет определять период актуальности состояния.

Определить открытый интервал просто: он отсчитывается от максимальной даты в таблице и уходит в будущее. Если объект некоторое время был недействительным, то такую семантику придётся отражать использованием пустых (NULL) значений полей либо специальной колонкой для отражения статуса.

Из описания виден недостаток этого способа: чтобы определить временные рамки периода, нужно просматривать и другие строки таблицы. Это ведёт к утяжелению запросов ввиду необходимости соединять таблицу с самой собой (self-join) или пользоваться подзапросами.

```
SELECT *
FROM documents
WHERE changed =
    (SELECT MAX(changed)
     FROM documents
     WHERE changed <= '2014-02-01') /* дата актуальности */
```

К достоинствам способа относятся простота, отсутствие избыточности и соответствующих проверок на пересечения периодов, быстрая вставка новых записей. К недостаткам — относительно «тяжёлые» соединяющиеся на себя (self-join) или вложенные запросы, необходимость использования NULL-значений или дополнительная колонка статуса для отражения «пустых» периодов.

Метод хранения даты состояния можно рекомендовать при интенсивной вставке записей и при отсутствии частых запросов по периодам.

Способ 2. Хранение интервала

Данный метод является попыткой устранить недостатки предыдущего. Чтобы избежать тяжёлых запросов с соединяющимися на себя таблицами, можно хранить интервал целиком. Колонка «Дата изменения» переименовывается в `date_from` (начало интервала) и остаётся в составе первичного ключа, так как предполагается, что интервалы не пересекаются. Также нужно добавить в таблицу вторую колонку `date_to` (окончание интервала).

Модификация структуры даёт возможность извлечь данные линейным запросом.

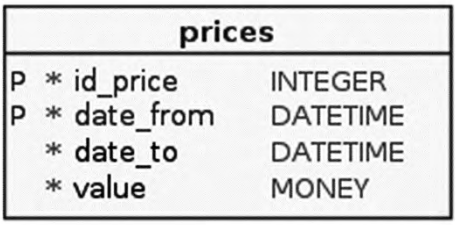

Рис. 28. Способ хранения интервала

```
SELECT *
FROM prices
WHERE '2014-02-01' BETWEEN date_from AND date_to
```

Запрос стал лёгким не бесплатно, цена за производительность при выборке: избыточность данных. Также возникает новая проблема: для проверки непротиворечивости границ интервалов необходимо создать в таблице триггер, что, несомненно, отразится на скорости вставки новых записей и модификации существующих.

Структура вызывает определённые осложнения для определения открытых интервалов, например, когда цена действует с момента её утверждения до неизвестной пока даты установления новой. Для решения такой проблемы приходится использовать фиктивные значения минимальной и максимальной даты, поддерживаемые конкретной СУБД. Например, соответствующие значения для SQL Server 2005 - «1 января 1753 года» и «31 декабря 9999 года», должны быть подходящими в большинстве случаев.

Преимущества способа — простота и эффективность запросов. Недостатки — необходимость дополнительной колонки для хранения даты окончания интервала, накладные расходы на поддержание непротиворечивости (проверка пересечения), меньшая скорость вставки, дополнительные условия определения открытых периодов.

Метод хранения интервалов можно рекомендовать в случае интенсивных и массивных запросов поиска при невысоких требованиях к скорости вставки и допустимом использовании процедурного расширения (триггеров), являющегося привязкой к конкретной СУБД.

Способ 3. Хранение номера периода (интервала)

Данный способ является фактическим стандартом кодирования измерений типа «дата» в аналитической обработке. В транзакционной обработке он наиболее уместен, когда одни и те же интервалы многократно используются для различных сущностей. Наиболее характерный пример использования — бухгалтерские задачи с их понятиями учётных периодов.

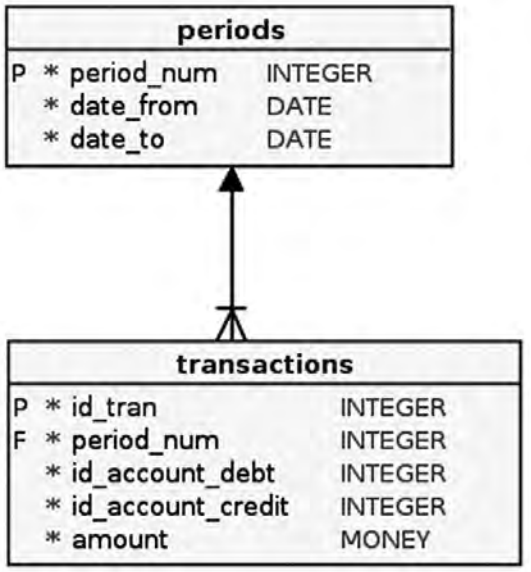

Рис. 29. Способ хранения номера интервала

В запросах появляется дополнительное соединение таблицей периодов, но обычно её длина исчисляется сотнями строк, что не влияет существенным образом на производительность.

```
SELECT t.*, p.date_from, p.date_to
FROM transactions t
     INNER JOIN periods p ON t.period_num = p.period_num
WHERE '2014-02-01' BETWEEN p.date_from AND p.date_to
```

Запросы, связанные с получением данных последнего открытого периода или выполнение нескольких запросов поиска по одному периоду хорошо оптимизируются СУБД. В первом случае нужен простой подзапрос `MAX(period_num)` по ключу без условий. Во втором — значение номера периода предварительно запоминается в переменной, служащей параметром, после чего выполняется пакет запросов. Сложности моделирования открытых периодов снижаются, например, если учёт

ведётся только постфактум, актуальным считается период с максимальным номером.

```
SELECT *
FROM transactions
WHERE period_num = (SELECT MAX(period_num) from periods)
```

Преимущества метода — меньшая избыточность за счёт повторного использования интервалов (периодов), относительная простота запросов, разнесение логики хранения периодов и самих объектов по разным таблицам; недостатки — более сложная структура, накладные расходы на поддержание непротиворечивости, не снижающие тем не менее, скорость вставки в связные таблицы. Метод можно рекомендовать для решения задач бухгалтерского и управленческого учёта.

Сравнение перечисленных способов

Для сравнения сведём основные характеристики в таблицу.

Табл. 2. Характеристики способов организации хронологических данных

Критерий оценки	**Способ 1**	**Способ 2**	**Способ 3**
Требуемая структура: количество таблиц/ связей/ колонок	1/0/2	1/0/3	2/1/4
Запрос типа «состояние на заданную дату»	Подзапросы и self-join	Линейный	Линейный с соединением
Скорость вставки	Высокая	Низкая	Высокая
Избыточность хранения	Нет	Да	Да
Поддержка непротиворечивости	Не нужна	Нужна	Нужна

Следует также упомянуть, что в некоторых промышленных СУБД, например, SQL Server 2008 Enterprise Edition и выше, имеется встроенная реализация временных рядов. Подобное решение может быть использовано, если привязка к СУБД и её редакции не является препятствием, а технические ограничения предлагаемого решения соответствуют рамкам вашего проекта.

Иерархические данные и деревья в SQL

Моделирование иерархических структур средствами реляционной СУБД встречается практически в любой информационной системе. Программисту приложений баз данных часто приходится сталкиваться с древовидными структурами. Находится множество примеров в самых разных предметных областях: классификация продукции, комплектация изделия, иерархия должностей, административно-территориальное деление, генеалогическое древо, наконец, просто дерево перебора вариантов или дерево классов. В статье «Способы реляционного моделирования иерархических структур данных» [30] был дан подробный обзор и сравнительный анализ характеристик основных подходов к решению задачи. Однако, статья оформлена в соответствии с форматом научного журнала, поэтому изложение материала с отсылками в теорию графов и множеств может показаться трудным для понимания и практического применения. Ниже мы рассмотрим сюжет с более популярной точки зрения.

В общем случае задача сводится к моделированию многоуровневой связи «главный-подчинённый», «предок-потомок», «общий-конкретный». Говоря математическим языком, мы **моделируем граф без циклов**.

Способ 1. Список смежности

Структура представляет собой интуитивно понятную организацию иерархии в виде таблицы с замкнутой на саму себя связью (рефлексивная связь). Корневые вершины отличаются от других пар пустым (`NULL`) значением ссылки на предка (колонка `id_parent`).

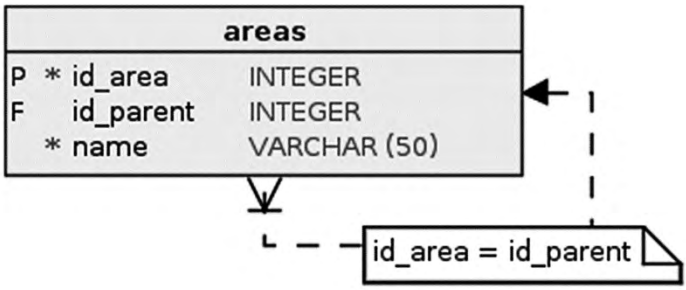

Рис. 30. Схема способа «Список смежности»

Название способа непосредственно отражает суть: в реляционной модели матрица смежности [30] может быть представлена в виде множества (списка) пар с номерами (идентификаторами, кодами) вершин, где каждая пара определяет ориентированную дугу между вершинами графа.

Табл. 3. Пример заполнения таблицы areas (территории)

id_area	id_parent	name
1	NULL	Санкт-Петербург
2	1	Центральный район
3	1	Московский район
4	1	Невский район
5	3	МО Новоизмайловское
6	3	МО Кузнецовское
7	4	МО Рыбацкое

Для выполнения многих выборок требуется поддержка рекурсивных запросов. Как правило, СУБД умеют их выполнять, но если вы столкнётесь с противоположным случаем, то выборки придётся строить с использованием других механизмов, например, временных таблиц или хранимых процедур и функций. Рассмотрим примеры типовых запросов для Microsoft SQL Server.

Выборка поддерева

Колонка level отображает глубину узла выбранного поддерева, за нулевой принимается начальный узел.

```
WITH subtree (id_area, id_parent, name, level)
AS (SELECT id_area, id_parent, name, 0 AS level
    FROM areas
    WHERE id_area = 3 /* выбранный узел */
    UNION ALL
    SELECT areas.id_area, areas.id_parent, areas.name, level
+ 1
    FROM areas
         INNER JOIN subtree ON areas.id_parent =
subtree.id_area
    WHERE areas.id_parent IS NOT NULL
)
SELECT id_area, name, level
FROM subtree;
```

Табл. 4. Результат выборки поддерева (список смежности)

id_area	name	level
3	Московский район	0
5	МО Новоизмайловское	1
6	МО Кузнецовское	1

Выборка всех предков (путь к узлу от корня)

```
WITH subtree (id_area, id_parent, name, level)
AS (SELECT id_area, id_parent, name, 0 AS level
    FROM areas
    WHERE id_area = 5 /* код узла */
    UNION ALL
    SELECT areas.id_area, areas.id_parent, areas.name, level
+ 1
    FROM areas
         INNER JOIN subtree ON areas.id_area =
subtree.id_parent)
SELECT id_area, name,
       (SELECT MAX(level) FROM subtree) - level + 1 AS level
FROM subtree
ORDER BY level;
```

Табл. 5. Результат выборки предков (список смежности)

id_area	name	level
1	Санкт-Петербург	0
3	Московский район	1
5	МО Новоизмайловское	2

Проверка, входит ли узел в поддерево

```
WITH subtree (id_area, id_parent)
AS (SELECT id_area, id_parent
    FROM areas
    WHERE id_area = 5 /* узел, проверяемый на вхождение */
    UNION ALL
    SELECT areas.id_area, areas.id_parent
    FROM areas
         INNER JOIN subtree ON areas.id_area =
subtree.id_parent)
SELECT CASE
  WHEN EXISTS (SELECT 1
               FROM subtree
               WHERE id_area = 3 /* корень поддерева */)
```

```
    THEN N'Входит'
  ELSE N'Не входит'
END AS result;
```

Подсчёт количества всех потомков узла

Запрос сходен с выборкой поддерева и последующим подсчётом количества выбранных узлов.

```
WITH subtree (id_area, id_parent)
AS (SELECT id_area, id_parent
    FROM areas
    WHEREid_parent = 3 /* код корня */
    UNION ALL
    SELECT areas.id_area, areas.id_parent
    FROM areas
         INNER JOIN subtree ON areas.id_parent =
subtree.id_area
    WHERE areas.id_parent IS NOT NULL
)
SELECT COUNT(1) AS qty
FROM subtree;
```

Определение высоты узла

```
WITH subtree (id_area, id_parent, level)
AS (SELECT id_area, id_parent, 0 AS level
    FROM areas
    WHERE id_area = 5 /* код узла */
    UNION ALL
    SELECT areas.id_area, areas.id_parent, level + 1
    FROM areas
         INNER JOIN subtree ON areas.id_area =
subtree.id_parent)
SELECT MAX(level) AS level
FROM subtree;
```

Способ «Подмножества»

В этом способе дерево представляется вложенными подмножествами. Чтобы не путать с рассматриваемым следующим способом, по недоразумению также называемым в англоязычной среде nested sets, было выбрано более нейтральное название «Подмножества».

Корневой уровень включает в себя все подмножества — узлы первого уровня. Узлы первого уровня, в свою очередь, включают в себя все узлы второго уровня и так далее. Например, иерархия административно-территориального деления муниципалитета может выглядеть следующим образом.

Рис. 31. Представление иерархии территорий в виде вложенных подмножеств

В терминах реляционной модели схема будет выглядеть следующим образом.

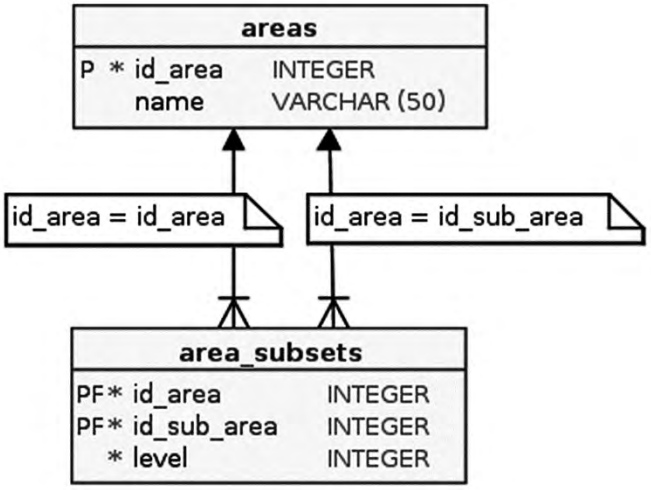

Рис. 32. Схема способа «Подмножества»

Табл. 6. Пример заполнения таблицы areas (территории)

id_area	name
1	Санкт-Петербург
2	Московский район

id_area	name
3	МО Новоизмайловское
4	МО Кузнецовское
5	Невский район
6	МО Рыбацкое
7	Центральный район

Табл. 7. Пример заполнения таблицы area_subsets (подмножества территорий)

id_area	id_sub_area	level
1	1	0
1	2	1
1	3	2
1	4	2
1	5	1
1	6	2
1	7	1
2	2	0
2	3	1
2	4	1
3	3	0
4	4	0
5	5	0
5	6	1
6	6	0
7	7	0

В способе «Подмножества» каждый элемент, кроме ссылки на непосредственных потомков, содержит ссылки и на потомков всех последующих уровней иерархии, создавая избыточность хранения. Количественная оценка избыточности для разных случаев сбалансированности дерева приведена в статье [30], а поддержать целостность данных придётся несложным триггером.

Рассмотрим, насколько типовые запросы стали короче и эффективнее.

Выборка поддерева

```
SELECT areas.id_area, areas.name, area_subsets.level
FROM area_subsets
```

```
     INNER JOIN areas ON area_subsets.id_sub_area =
areas.id_area
WHERE area_subsets.id_area = 2 /* корень поддерева */
ORDER BY level
```

Табл. 8. Результат выборки поддерева (подмножества)

id_area	name	level
2	Московский район	0
3	МО Новоизмайловское	1
4	МО Кузнецовское	1

Выборка предков

```
SELECT areas.id_area, areas.name,
       (SELECT MAX(s.level)
        FROM area_subsets s
        WHERE s.id_sub_area = area_subsets.id_sub_area
       ) - area_subsets.level + 1 AS level
FROM area_subsets
     INNER JOIN areas ON area_subsets.id_area = areas.id_area
WHERE area_subsets.id_sub_area = 3 /* узел */
ORDER BY level
```

Табл. 9. Результат выборки предков (подмножества)

id_area	name	level
1	Санкт-Петербург	0
2	Московский район	1
3	МО Новоизмайловское	2

Вхождение в поддерево

```
SELECT result = CASE
    WHEN EXISTS(
      SELECT 1 FROM area_subsets
        WHERE id_sub_area = 4 /* узел */
              AND id_area = 2 /* корень поддерева */)
    THEN N'Входит'
    ELSE N'Не входит'
END
```

Подсчёт количества всех потомков узла

```
SELECT COUNT(1) - 1 AS qty
FROM   area_subsets
WHERE id_area = 2 /* корень поддерева */
```

Определение высоты узла

```
SELECT MAX(level) AS level
FROM   area_subsets
WHERE  id_sub_area = 4 /* узел */
```

Способ «Маршрут обхода»

Согласно теории графов, для обхода дерева существует три способа: можно проходить узлы в префиксном, в инфиксном и в суффиксном порядке. Префиксный порядок обхода дерева рекурсивно определяется так: сначала корень дерева, потом узлы левого поддерева в префиксном порядке, наконец, узлы правого поддерева в префиксном порядке. Пример обхода в префиксном порядке приведён на рисунке ниже.

Хранение маршрута обхода дерева в префиксном порядке также встречается у Джо Селко [26], однако в книге способ носит не вполне соответствующее его сути название «вложенные множества» (nested sets).

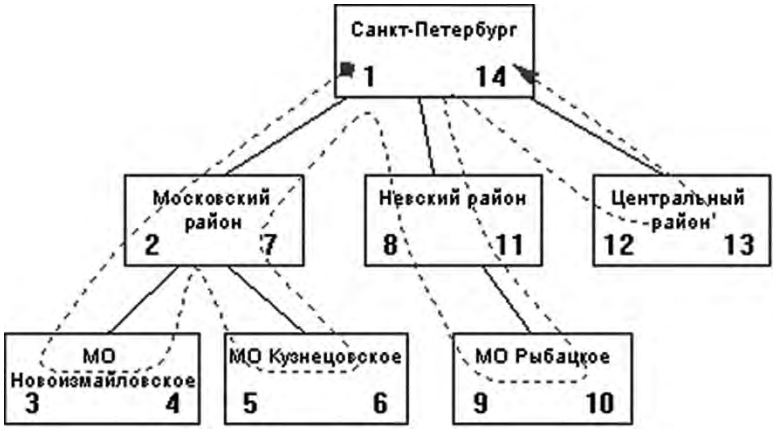

Рис. 33. Маршрут обхода дерева в префиксном порядке

Каждый квадрат на рисунке обозначает узел, цифра в левом его углу является порядковым номером этапа маршрута при входе в узел, а цифра справа — номером при выходе, когда тем же способом пройдены все потомки.

Нетрудно заметить, что номера потомков всегда располагаются в интервале между соответствующими номерами предка, сколь угодно

дальнего. Храня порядок обхода дерева, этим свойством можно воспользоваться в типовых запросах, избежав рекурсии.

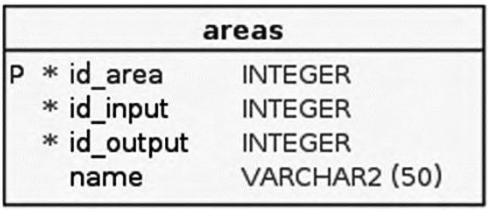

Рис. 34. Схема способа «Маршрут обхода»

Табл. 10. Пример заполнения таблицы в способе «Маршрут обхода»

id_area	name	id_input	id_output
1	Санкт-Петербург	1	14
2	Московский район	2	7
3	МО Новоизмайловское	3	4
4	МО Кузнецовское	5	6
5	Невский район	8	11
6	МО Рыбацкое	9	10
7	Центральный район	12	13

Избыточность хранения делает необходимым пересчёт порядка обхода при добавлении новых или перемещении существующих узлов (удаление можно игнорировать). В соответствующем триггере для этого необходимо реализовать последовательный порядок обхода. Но, например, если добавляется элемент самого нижнего уровня, то приходится пересчитывать **все** номера «выше» или «правее», что может быть сравнимо с затратами на пересчёт маршрута по всему дереву.

Для минимизации последствий таких пересчётов, можно нумеровать входы и выходы из узлов с некоторым интервалом, например, 100 или 1000, что в значительной степени зависит от предварительных оценок количества хранимых узлов дерева. В этом случае вставка новых элементов будет происходить без полной перенумерации.

Выборка поддерева

В запросе не вычисляется глубина узла. Добавление этой функции приведёт к введению выполняющегося для каждого элемента агрегирующего подзапроса (см. листинг 15).

```
SELECT a1.id_area, a1.id_input, a1.name
FROM areas a1
     INNER JOIN areas a2
       ON a1.id_input BETWEEN a2.id_input AND a2.id_output
WHERE a2.id_area = 2 /* корень поддерева */
ORDER BY a1.id_input
```

Табл. 11. Результат выборки поддерева (маршрут обхода)

id_area	id_input	name
2	2	Московский район
3	3	МО Новоизмайловское
4	5	МО Кузнецовское

Выборка предков

Выборка всех предков симметрична предыдущему запросу относительно BETWEEN. В запросе не вычисляется глубина узла.

```
SELECT a1.id_area, a1.id_input, a1.name
FROM areas a1
     INNER JOIN areas a2
       ON a2.id_input BETWEEN a1.id_input AND a1.id_output
WHERE a2.id_area = 4 /* узел */
ORDER BY a1.id_input
```

Табл. 12. Результат выборки предков (маршрут обхода)

id_area	id_input	name
1	1	Санкт-Петербург
2	2	Московский район
4	5	МО Кузнецовское

Вхождение в поддерево

```
SELECT result = CASE
    WHEN EXISTS(SELECT 1 FROM areas as a1, areas as a2
      WHERE a1.id_area = 4 /* узел */
            AND a2.id_area = 2 /* корень поддерева */
            AND a1.id_input BETWEEN a2.id_input AND
a2.id_output)
```

```
    THEN N'Входит'
    ELSE N'Не входит'
END
```

Подсчёт количества всех потомков узла

```
SELECT COUNT(1) AS qty
FROM (SELECT a1.id_area
      FROM areas a1
           INNER JOIN areas a2
             ON a1.id_input BETWEEN a2.id_input AND
a2.id_output
      WHERE a2.id_area = 2 /* узел */
     ) t
```

Определение высоты узла

```
SELECT COUNT(1) AS level
FROM (SELECT a1.id_area
      FROM areas a1
           INNER JOIN areas a2
             ON a2.id_input BETWEEN a1.id_input AND
a1.id_output
      WHERE a2.id_area = 4 /* узел */
     ) t
```

Способ «Материализованные пути»

Суть способа заключается в хранении пути от вершины до данного узла в явном виде. Путь одновременно является и ключом. Наиболее близкая аналогия представления способа — знакомая всем нумерация частей, разделов и глав в книге.

Табл. 13. Пример заполнения таблицы areas (материализованные пути)

area_path	name
1	Санкт-Петербург
1.1	Московский район
1.1.1	МО Новоизмайловское
1.1.2	МО Кузнецовское
1.2	Невский район
1.2.1	МО Рыбацкое
1.3	Центральный район

Способ является наиболее наглядным с точки зрения кодификации элементов: каждый узел получает значение, которое пригодно для восприятия пользователем. Путь и его части несут смысловую нагрузку. Такие свойства используются в классификациях, предназначенных для широкого применения, например, в стандартизованных справочниках территорий (ОКАТО), отраслей экономики (ОКВЭД, NAICS), медицинских диагнозов (МКБ — международный классификатор болезней) и во многих других областях.

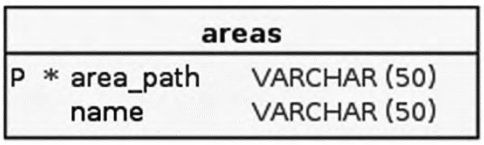

Рис. 35. Схема способа «Материализованные пути»

Ограничив максимальное количество уровней иерархии и число прямых потомков, можно обойтись без разделителей, используя символьные коды с фиксированной разбивкой на группы разрядов. Пустые лидирующие разряды в группе заполняются нулями.

Однако запросы не всегда могут быть эффективно реализованы на уровне СУБД, так как, например, поиск подстроки, вызывает сканирование таблицы вместо поиска по ключу или его начальному фрагменту.

Выборка поддерева

В запросе не вычисляется глубина узла.

```
SELECT   *
FROM     areas
WHERE    area_path LIKE '1.1%' /* корень поддерева */
ORDER BY area_path
```

Табл. 14. Результат выборки поддерева (материализованные пути)

area_path	name
1.1	Московский район
1.1.1	МО Новоизмайловское
1.1.2	МО Кузнецовское

Выборка предков

В запросе не вычисляется глубина узла.

```
SELECT   *
FROM     areas
WHERE    '1.2.1' /* узел */ LIKE area_path + '%'
ORDER BY area_path
```

Табл. 15. Результат выборки предков (материализованные пути)

area_path	name
1	Санкт-Петербург
1.2	Невский район
1.2.1	МО Рыбацкое

Вхождение в поддерево

```
WITH
  node(area_path) AS (SELECT '1.2.1'),
  subtree(area_path) AS (SELECT '1.2')
SELECT result = CASE
    WHEN EXISTS(SELECT 1
      FROM node, subtree
      WHERE SUBSTRING(node.area_path, 1,
LEN(subtree.area_path))
            = subtree.area_path)
    THEN N'Входит'
    ELSE N'Не входит'
END
```

Подсчёт количества всех потомков узла

```
SELECT COUNT(1) - 1 AS qty
FROM   areas
WHERE  area_path LIKE '1.1%' /* корень поддерева */
```

Определение высоты узла

```
SELECT COUNT(1) AS level
FROM   areas
WHERE  '1.1.2' /* узел */ LIKE area_path + '%'
```

Критерии сравнения

В качестве критериев сравнения приведённых способов используются следующие показатели.

- **Сложность схемы базы данных**. Определяется как количество достаточных для реализации таблиц, ссылок (связей) между ними и колонок, содержащих данные о структуре графа (матрице смежности).
- **Запросы на извлечение данных**. Характеризуются количеством необходимых соединений. Наиболее сложным вариантом является рекурсивный запрос, в котором число соединений в цикле соответствует глубине иерархии. Например, выборка поддерева с 5 уровнями будет осуществляться в цикле из 5 итераций, результат каждой из которых соединяется с предыдущим.
- **Запросы на изменение данных**. Запросы на изменение данных, такие, как вставка и удаление узлов, характеризуются необходимостью дополнительных операций со связанными узлами и обновлением избыточных данных.
- **Избыточность хранения данных** и необходимость поддержки целостности. Характеризуется необходимостью дополнительного императивного кода (триггеров) помимо декларативной ссылочной целостности.

Сведя перечисленные характеристики в общую таблицу, мы получим сравнительную картину, являющуюся неплохим подспорьем для выбора одного из способов реализации.

Табл. 16. Сравнительные характеристики рассмотренных способов

	Списки смежности	**Подмноже-ства**	**Хранение маршрута об-хода**	**Материализо-ванные пути**
Сложность схемы базы данных (количество)				
Таблиц	1	2	1	1
Ссылок	1	2	0	0
Колонок	3	5	4	2

	Списки смежности	Подмножества	Хранение маршрута обхода	Материализованные пути
Запросы на извлечение данных (число соединений)				
Выборка поддерева, число соединений	H-N+1 (*)	2	2	1
Выборка пути от узла до корня (предков)	N (*)	2	2	1
Вхождение в поддерево	H-N+1 (*)	1	2	0 (сравнение значений двух строк)
Подсчёт количества всех потомков узла	H-N+1 (*)	1	2	1
Определение высоты узла	N (*)	1	2	1
Соответствие порядка следования узлов сортировке по ключу	нет	нет	нет	да
Запросы на изменение данных				
Прямая вставка новых узлов для листа	да	нет	да, если есть свободный номер, иначе пересчёт диапазонов	да
Прямая вставка новых узлов для внутренней вершины	нет, изменение ссылки потомков	нет, перегруппировка подмножеств	нет, пересчёт диапазонов	нет, пересчёт номеров узлов
Прямое перемещение поддерева	да, изменение ссылки на предка	нет, перегруппировка подмножеств	нет, пересчёт диапазонов	нет, пересчёт номеров узлов

	Списки смежности	**Подмножества**	**Хранение маршрута обхода**	**Материализованные пути**
Прямое удаление поддерева	нет, рекурсивное (каскадная DRI)	нет, перегруппировка подмножеств	да, заданием диапазона	да, заданием шаблона подстроки
Избыточность хранения	**нет**	**да**	**да**	**да**
оценка избыточности хранения дерева из N вершин	N	не хуже ((N-1)/2)·N	2·N	не хуже ((N-1)/2)·N
Ограничения высоты дерева	нет	нет	нет	да
Императивная поддержка целостности	нужна	нужна	нужна	нужна

Обозначения в таблице: (*) – рекурсивный запрос, H – высота дерева, N – текущий уровень.

Следует понимать, что в приведённых решениях нет «плохих» или «хороших» способов: проектировщик может сделать свой выбор оптимального решения исходя из условий конкретной задачи на основании предлагаемого множества критериев. Благодаря своей интуитивности, наиболее часто встречается на практике реализация списка смежности, однако необходимость рекурсивных запросов является узким местом, несомненно влияющим на производительность ваших приложений.

Интернационализация/локализация данных и проброс контекста

В англоязычной среде термины интернационализация и локализация, ввиду их длины, сокращают до `i18n` и `l10n`, соответственно.

Интернационализация — технологическая адаптация программного обеспечения к разным языкам и культурам пользователей.

Локализация — собственно процесс перевода пользовательского интерфейса с одного языка и культуры на другие.

В русскоязычной среде оба понятия часто смешивают и употребляют обобщающий термин «локализация», очевидно, ввиду его меньшей длины.

Не вдаваясь в терминологический спор, существенным, в рамках сюжета книги, является разделение интернационализации на адаптацию непосредственно интерфейса пользователя и той части информации, которая извлекается из базы данных. Иными словами, разделим локализацию данных, хранящихся в БД, и их представления на уровне приложений.

Форматы отображения дат, чисел, разделителей и прочих величин, зависящих от региональных настроек, остаются на ответственности клиентского приложения. Даже простая SQL-консоль, как правило, с такой задачей справляется. Тогда как текстовая информация требует перевода, хранящегося на уровне БД.

Каким же образом можно осуществить интернационализацию данных в БД?

Линейный подход к решению приводит к необходимости дублирования колонок строковых типов, например, добавляя к ним идентифицирующий культуру суффикс в виде пары «язык-страна».

Ничего выдумывать не нужно, стандарт ISO 639-1 определяет двухсимвольные коды для всех известных языков. Другой стандарт, ISO 3166-1, задаёт аналогичные коды для всех стран.

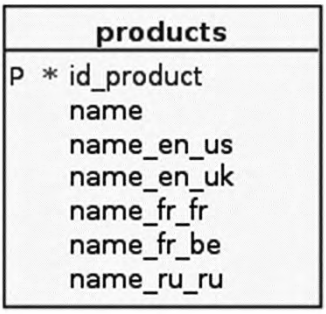

Рис. 36. Интернационализация добавлением колонок

Хотя такой способ может быть нагляден и удобен при работе с одной таблицей, с технологической точки зрения он труден для унификации и имеет серьёзные ограничения на число культур ввиду соответствующих ограничений СУБД на количество столбцов и общую длину строки.

Трудность унификации состоит в том, что один и тот же запрос нужно всякий раз определять для каждой культуры, так как необходимо выбирать разные колонки. SQL будет всякий раз формироваться приложением динамически, что снизит производительность коротких и лёгких запросов.

```
SELECT id_product, name_en_us AS name
FROM products
```

В статическом же варианте любой запрос будет нагружен многоярусными конструкциями селектора.

```
SELECT id_product,
  CASE
    WHEN code_loc = 'en_us' THEN name_en_us
    WHEN code_loc = 'fr_fr' THEN name_fr_fr
    WHEN code_loc = 'ru_ru' THEN name_ru_ru
    ELSE name /* перевод по умолчанию */
  END AS name
FROM products
     /* константу можно передать параметром */
     CROSS JOIN
       (SELECT CAST('en_us' AS varchar(5)) AS code_loc) l
```

Рекомендовать этот способ можно лишь в условиях ограниченного несколькими культурами периметра задачи и требования отсутствия в запросах соединений, например, для обеспечения производительности аналитической обработки. Тем не менее, следует вначале убедительно обосновать, почему не подходят другие способы, не требующие реструктуризации при каждом добавлении языка.

Поскольку таблицы двумерны, то любое необходимое для моделирования расширение по горизонтали (числом столбцов) всегда может быть заменено расширением по вертикали (числом строк) с добавлением к ключу столбца-дискриминатора. Это общий приём.

Если в исходную нелокализованную таблицу добавить столбец-дискриминатор «Код культуры», введя его в состав первичного ключа, то при неизменной остальной структуре таблица будет расти в длину.

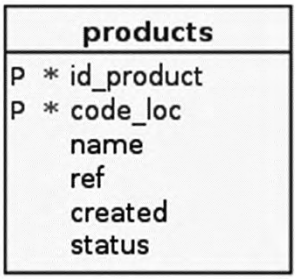

Рис. 37. Интернационализация прямым добавлением дискриминатора.

Такой подход решает проблемы унификации, потому что запрос остаётся неизменным для всех языков, меняется лишь условие фильтрации.

```
SELECT *
FROM products
WHERE code_loc = 'ru_ru'
```

Очевидно, что не все колонки таблицы требуют локализации, и часть данных одного и того же объекта будет дублироваться из строки в строку для разных культур.

Мы **нарушили вторую нормальную форму** — некоторые атрибуты зависят только от части составного ключа, а именно от идентификатора продукта. Поэтому изменение любого из значений, не подлежащих локализации, необходимо синхронизировать с другими дублирующимися. Придётся программировать поддерживающий целостность триггер.

Избежать перечисленных проблем позволит вынесение всех столбцов, подлежащих локализации, в таблицу-расширение, связанную с основной отношением «один-ко-многим».

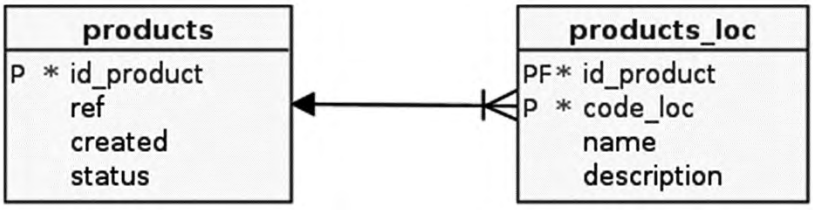

Рис. 38. Интернационализация с помощью таблицы расширения.

В этом способе, не нарушающем нормализацию, единственным недостатком является дополнительное соединение в запросе, которое достаточно просто скрыть соответствующим видом.

```
CREAT VIEW v_products_loc AS
SELECT p.*, pl.name, pl.description
FROM products p
     INNER JOIN products_loc pl ON p.id_product =
pl.id_product
```

Тогда запрос примет прежний вид

```
SELECT *
FROM v_products_loc
WHERE code_loc = 'ru_ru'
```

Следующий шаг унификации схемы — кодификация культур в целочисленные значения и добавление соответствующей таблицы-справочника.

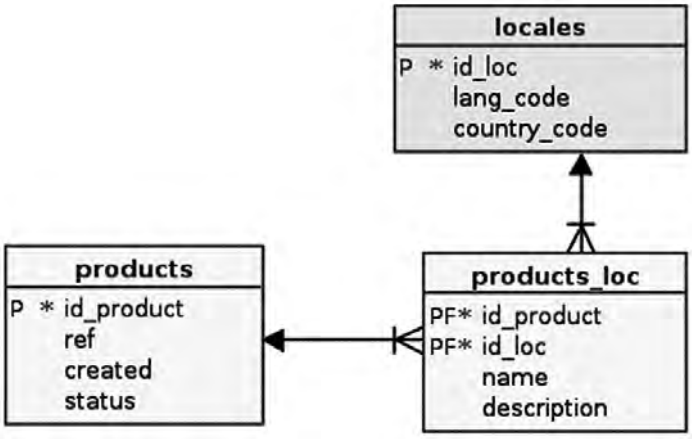

Рис. 39. Использование идентификатора культуры.

Замена кодов культур на их идентификаторы не добавит соединения с таблицей `locales` в запросы, потому что в качестве параметра будет выступать непосредственно идентификатор. Можно использовать как собственные значения, так и имеющие более широкое применение, например, коды локалей Windows (см. таблицу Microsoft Locale ID Values в документации MSDN).

```
SELECT *
FROM v_products_loc
WHERE id_loc = 2
```

Приведённые запросы формируются в приложении, которое знает контекст соединения с СУБД для текущего пользователя и передаёт соответствующий код или идентификатор культуры в запрос. Но как быть, если обработка осуществляется на уровне СУБД, например, в хранимой процедуре или в триггере?

В процедуру код культуры можно передать одним из параметров, что хотя и усложняет интерфейс, но решает проблему ad hoc. А вот в триггере параметров не может быть в принципе, передать туда управляющие воздействия явным порядком очень непросто.

Поэтому если вы планируете реализовать часть логики средствами СУБД, необходимо предусмотреть хранение контекста текущего пользователя или сессии, одним из элементов которого будет код культуры.

Такой подход называется *пробросом контекста из приложения в СУБД*, его суть состоит в следующих приёмах:

- поддержание в актуальном состоянии списка активных сессий;
- привязка контекста к сессии;
- связывание сессии с внутренним идентификатором соединения СУБД;
- при обработке на уровне СУБД получение элементов контекста по внутреннему идентификатору.

Внутренний идентификатор соединения, как правило, присутствует в любой СУБД, но его реализация может быть совершенно разной. Например, в SQL Server это глобальная переменная уровня сессии `@@spid`, возвращающая целочисленный номер. В PostgreSQL для той же цели может использоваться системная функция `pg_backend_pid()`. В общем случае, нужно обратиться к технической документации по вашей СУБД и поискать в разделе системных функций те из них, что связаны с состоянием соединения.

Схема БД немного усложняется, добавляется таблица сессий, в которой хранятся и элементы контекста.

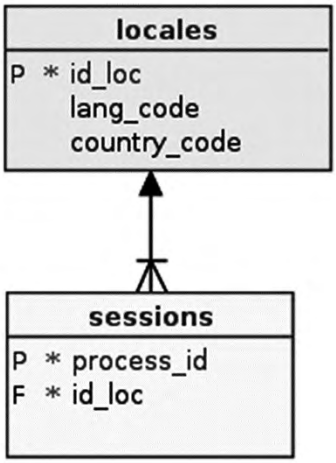

Рис. 40. Хранение кода культуры в контексте сессии.

Дополнительную сложность представляют собой приложения, не поддерживающие постоянное соединение и работающие через их пул. Такая ситуация типична для веб-приложений. В этом случае при каждом новом подключении приложение должно прежде всего обновить контекст. Ниже пример для SQL Server.

```
UPDATE sessions
SET id_loc = 3
WHERE process_id = @@spid;
IF @@rowcount = 0
  INSERT INTO sessions (process_id, id_loc)
  VALUES(@@spid, 3);
```

Удобнее всего подобный код реализовать в виде хранимой процедуры, которую приложение вызывает при каждом соединении. После проброса контекста на уровень СУБД можно непосредственно использовать его в запросах, не прибегая к передаче соответствующих параметров до завершающего отсоединения.

```
CREATE PROCEDURE product_update_name
  @id_product int, @name nvarchar(50)
AS
BEGIN
  SET NOCOUNT ON;
  UPDATE products_loc
  SET name = @name
  WHERE id_product = @id_product
```

```
        AND id_loc = (SELECT id_loc FROM sessions WHERE
process_id = @@spid);
END;
```

Проброс контекста также упростит работу с видами. Определение v_products_loc можно переписать следующим образом

```
CREAT VIEW v_products_loc AS
SELECT p.*, pl.name, pl.description
FROM products p
     INNER JOIN products_loc pl ON p.id_product =
pl.id_product
WHERE id_loc = (SELECT id_loc FROM sessions WHERE process_id
= @@spid)
```

Тогда запрос, выводящий информацию по продукту с учётом культуры пользователя упростится до минимума.

```
SELECT *
FROM v_products_loc
```

Но и это ещё не все.

Многие СУБД поддерживают триггеры для видов, что может существенно упростить работу с локализованными данными. Для этого следует создать простой INSTEAD OF триггер, стартующий всякий раз, когда над видом выполняется оператор вставки, обновления или удаления. Ниже пример для SQL Server и случая обновления.

```
CREATE TRIGGER v_product_loc_update ON v_product_loc INSTEAD
OF UPDATE
AS
BEGIN
  SET NOCOUNT ON;
  UPDATE products_loc
  SET name = @name
  FROM products_loc pl
       INNER JOIN inserted i ON pl.id_product = i.id_product
  WHERE pl.id_loc = (SELECT id_loc FROM sessions WHERE
process_id = @@spid);
END;
```

При наличии подобного триггера на определённый выше вид v_products_loc, виртуальную таблицу можно непосредственно изменять

оператором UPDATE, не специфицируя текущую культуру, определяемую контекстом текущей сессии.

```
UPDATE v_product_loc
SET name = 'Мука в/с'
WHERE id_product = 457
```

Этот запрос будет эквивалентен прямому изменению таблицы расширений.

```
UPDATE products_loc
SET name = 'Мука в/с'
FROM products_loc pl
WHERE id_product = 457 AND
      id_loc = (SELECT id_loc FROM sessions WHERE process_id
= @@spid)
```

Метаданные

Невозможно разработать сколь-нибудь сложную настраиваемую информационную систему без управления метаданными. В самой простой программе типа записной книжки с тремя таблицами метаданные присутствуют в неявном, «зашитом» в код виде: пользовательские названия колонок в отображаемых таблицах, список атрибутов для поиска по критериям, связи «главная-подчинённая», форматы экспорта и импорта, расширение атрибутов хранимых данных.

Если таблиц в приложении будет не три, а уже тридцать, то в программировании появляется необходимость унификации компонентов с целью повторного использования ряда функций, экранных форм, протоколов обмена и т. д. Для решения этой задачи необходимы метаданные, управляющие функционалом таких компонентов.

Самодокументируемая, управляемая настройками без перепрограммирования система — почти идеальный случай в корпоративном софтостроении, и метаданные — один из важных строительных блоков и подходов к её проектированию.

В более широком смысле, понятие и назначение метаданных шире, оно восходит к онтологиям в их понятии в рамках информатики, то есть к формализованным и структурированным описаниям некоторой области знаний, в том числе предметной области.

Метаданные — формализованные структурированные описания системы.

Действительно, если мы разработали компонент для поиска по таблице, пользуясь метаданными реляционной схемы, почему бы не поднять его функционал на концептуальный уровень сущности с учётом, например, ограничений значений домена атрибутов, смысла связей и бизнес-правил?

Основным препятствием к такого рода разработке является сложность разработки формальных описаний предметных областей, таксономий, словаря. Если при создании сложного тиражируемого продукта уровня корпоративной информационной системы развитый механизм метаданных на порядки упрощает настройку и сопровождение, освобождая от рутинного программирования, то в заказном софтостроении относительно небольших систем (порядок 100 тысяч строк кода и 100 таблиц) глубокая проработка онтологий, как правило, является нерентабельной.

Тем не менее, даже в условиях сжатых сроков и бюджета необходимо реализовать некоторый минимум, позволяющий избежать бессмысленного повторного кодирования однотипных функций.

Что может взять на вооружение программист?

Во-первых, многие СУБД кроме собственно метаописаний таблиц, колонок, ограничений целостности и т. д. поддерживают хранение пользовательских метаданных. В этом случае каждая таблица и колонка могут быть ассоциированы с некоторым списком вида «ключ-значение», куда можно вносить не только документирующую информацию, но и влияющие на поведение приложений параметры. Например, ассоциация с колонкой «Номер документа» пары «Маска ввода=99_999999» позволит настроить ввод соответствующих значений в экранных формах в соответствии с требованиями клиента.

Аналогичным образом при использовании ОРП в модели домена предметной области можно определять пользовательские атрибуты свойств класса. Выразительные возможности языка программирования в этом случае будут значительно богаче хранимых списков «ключ-значение».

Однако, такие подходы имеют ограничения по масштабируемости. Пока с базой данных работает одно приложение, а домен-сборка компилируется в одном проекте не проявляется неоднозначности интерпретации метаданных. Но уже на стыке двух предметных областей начинают происходить непредвиденные и, потому, неприятные события. Например, если в рамках обслуживания клиентов мы определили, что ключевым параметром поиска будет название фирмы, то в бухгалтерии могут предпочесть находить нужного клиента по его налоговому номеру.

Решение в рамках метаданных «ключ-значение» представляет собой типичный «костыль» софтостроения: в ключ через разделитель, например, точку, добавляется название, контекст или иная отличительная характеристика приложения.

Каноническое решение в рамках домена предметной области ещё менее красиво: каждая область должна иметь свой домен, соответственно, классы `CRM.Client` и `Accounting.Client` определены в разных пространствах имён, имея, соответственно, разные значения атрибутов `DefaultSearchProperty` уровня класса. Будучи спроецированными на одну и ту же таблицу базы данных `clients`, оба класса обнаруживают сильное структурное и поведенческое сходство. Чтобы избежать повторного кодирования реализации, программисту придётся обобщать, вводя класс «абстрактного клиента» в некий супердомен `Business.Core`, или в буквальном смысле собирать класс по частям, агрегируя его из классов-«кубиков», определённых тоже в пространстве имён супердомена.

Упомянутый пример — один из важных недостатков подхода «модель предметной области», прекрасно работающего в рамках одиночных приложений и соответствующих доменов. Как только задачи пересекают границы предметных областей, то коса находит на камень, и замок из слоновой кости превращается в избушку бабы Яги. Однако, целью главы вовсе не является критика подходов, пропагандируемых Мартином Фаулером [28], тем более, что его деятельность по наведению в головах элементарного порядка, пускай и шаблонного, является скорее позитивной на массовом уровне «прокачки» героев-программистов разными «гуру» компьютерной алхимии.

Итак, если для вашего случая использование встроенных возможностей СУБД по хранению метаданных недостаточно, придётся приступить к проектированию структур, способных не только хранить необходимые для управления программной системой метаданные, но и расширяться без изменения скелета.

Один из вариантов синтеза такой схемы был опубликован в статье о проектировании ядра информационной системы [29]. Однако предложенный вариант был ориентирован на реализацию объектно-ориентированной надстройки средствами самой СУБД и предполагал иерархию наследования всех классов в системе от одного общего предка. Такой подход, несмотря на широкие рамки применения, не всегда может быть использован вследствие требований к системе, например, невозможности привязки к конкретной СУБД. Поэтому ниже приведён минимальный вариант схемы хранения метаданных без названных ограничений.

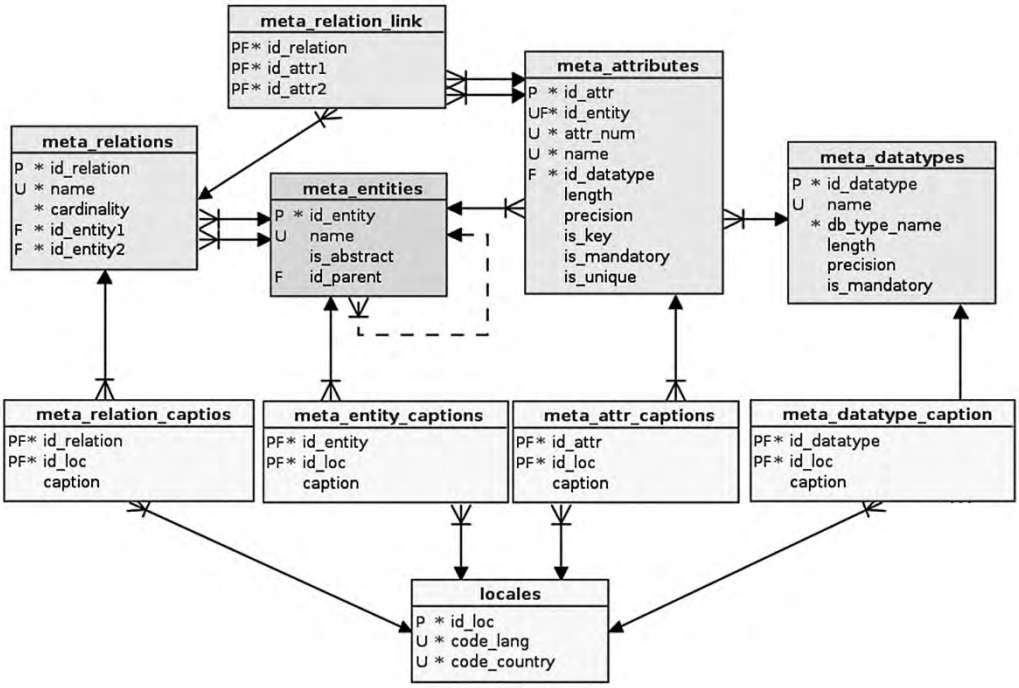

Рис. 41. Схема хранения метаданных.

Центральным элементом является таблица meta_entities, содержащая информацию о сущностях. Если вы предпочитаете терминологию ООП, то можно говорить о классах. Для сущностей предусмотрена возможность наследования типов и наличия абстрактных предков. Полагаем, что каждой сущности соответствует таблица, имя которой определяется значением name.

Сущности обладают атрибутами, описание которых хранится в соответствующей связанной таблице meta_attributes. Атрибут кроме своего системного имени (name), соответствующего названию колонки таблицы, имеет и порядок следования в списке согласно номеру (attr_num). Значения атрибута могут быть ключевыми (is_key), то есть составлять первичный ключ, обязательными (is_mandatory), чтобы соответствующие колонки были объявлены как NOT NULL, и уникальными (is_unique), то есть входить в уникальный индекс или ограничение уникальности столбца.

Тип данных атрибута определяется связью с соответствующим типом из таблицы meta_datatypes. Как правило, имя соответствует встроенному типу данных СУБД, но можно описывать и пользовательские типы, например t_doc_number, определённый как varchar(15).

Связи между сущностями описываются таблицами отношений meta_relations и спецификации связи meta_relation_links. Последняя содержит определения соответствий между внешними и первичными или другими ключами. Имя отношения (name) является именем соответствующего ограничения внешнего ключа в СУБД.

Наконец, используя приведённый в предыдущей главе подход к интернационализации данных, таблицы xxx_captions содержат пользовательские названия всех перечисленных элементов метамодели: сущностей, атрибутов, типов данных и связей.

Аналогичным образом в описание можно добавить определения индексов, не нарушая каркаса. Предлагаю вам сделать это самостоятельно.

Синтезированной схемы метаданных уже достаточно, чтобы вы смогли разработать для несложный генератор запросов, позволяющий освободить

программистов от написания рутинного кода по нескончаемому потоку пожеланий пользователей информационной системы.

Другой пример — параметризация бизнес-правил, тема необъятная. Перед программистом всегда стоит выбор: реализовать правило в коде ad hoc или разработать более общее решение с настройками. Введение в метаданные понятий функций и параметров может помочь в создании абстрактного механизма управления правилами обработки.

Помогут метаданные и в процессе обмена информацией между подсистемами. Любой XML-документ становится возможным снабдить исчерпывающей схемой, исключающей неоднозначности неполно структурированных моделей данных.

В руках умелого разработчика технология метаданных позволит решить многие проблемы настройки заказной информационной системы или коробочного программного продукта не прибегая к перепрограммированию.

Реестр объектов и аудит

Для приближения к терминологии ООП объектами в данном случае являются экземпляры сущностей. Строки в соответствующих таблицах.

Для решения ряда задач бывает необходимо иметь унифицированный доступ ко всем объектам в системе. Например, для определения прав доступа, рассматриваемых в следующей главе. Или для параметризации бизнес-правил, когда на входе или в управлении одного и того же правила могут подаваться объекты разных классов.

Таким подходом является сквозная идентификация всех объектов в системе по некоторому уникальному идентификатору id_object. При наследовании всех сущностей (классов) от одного абстрактного предка такая идентификация происходит прозрачным образом. Однако в системе возникает узкое место — таблица абстрактных объектов, связанная со всеми остальными таблицами. Ещё труднее интегрировать в эту схему с одним предком уже существующие подсистемы.

Децентрализованным решением сквозной идентификации является реестр объектов.

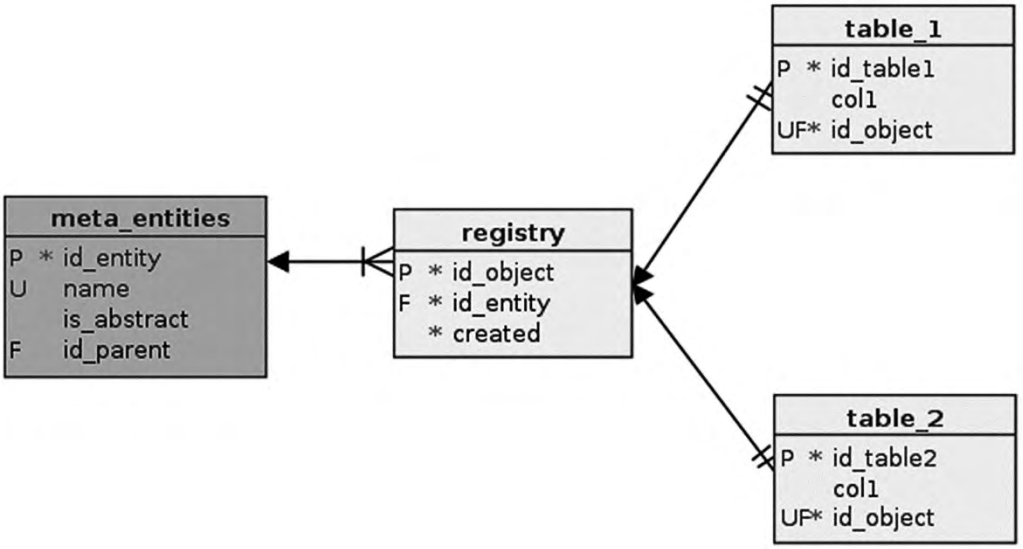

Рис. 42. Реестр объектов.

На схеме реестр связан с таблицами отношением «один-к-одному», однако, такая связь не является обязательной, в отличие от схемы с общим предком. В высоконагруженных приложениях вы можете разделить во времени и между процессами этапы вставки записи в таблицу и получения в реестре нового `id_object`. Также, возможно физическое разделение базы данных на системную, где хранятся метаданные и реестр, и прикладную. В этом случае следует объявить колонку связанной таблицы `id_object` как `NULL`, не нужно декларировать внешние ключи и отграничения уникальности.

Что даёт реестр кроме унифицированного подхода?

Например, можно вывести статистику количества объектов, сгруппированных по их классам. Вместо просмотра всех тех таблиц, что описаны в метаданных, пишем простой запрос.

```
SELECT count(1), e.name
FROM registry r
     INNER JOIN meta_entities e ON r.id_entity = e.id_entity
GROUP BY e.name
ORDER BY 1 DESC
```

И это только простой пример, навскидку.

С помощью реестра можно организовать аудит, отслеживая не только дату создания объекта, но и моменты его последнего изменения, включая пользователя, совершившего модификации.

Для сохранения журнала изменений соответствующая таблица связывается с реестром. При его отсутствии такую таблицу пришлось бы создавать для каждой сущности, включённой в аудит. Управлять аудитом тоже просто: добавим в метаданные уровня сущности параметры, определяющие кроме собственно необходимости отслеживания изменений длину истории, детализацию и другие необходимые для вашего случая атрибуты.

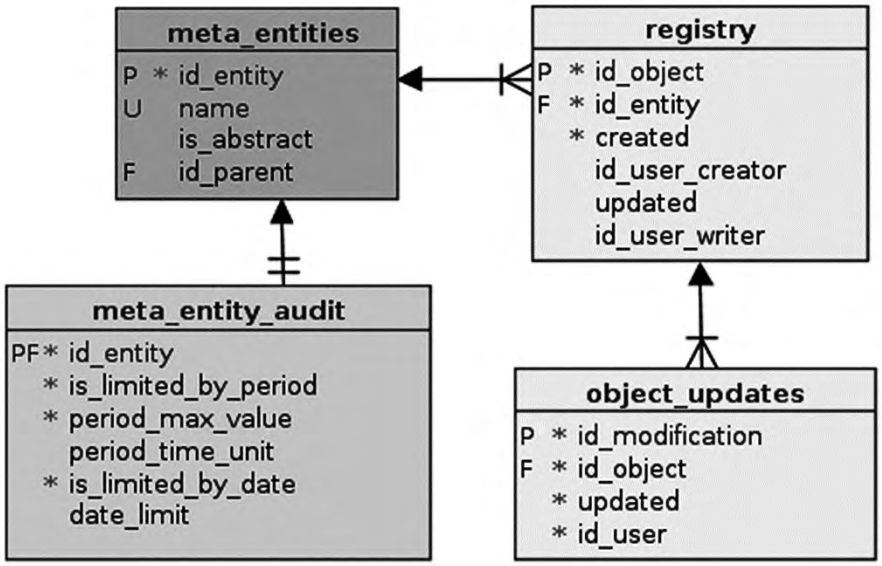

Рис. 43. Централизованный журнал изменения объектов.

Безопасность и доступ к данным

Тема безопасности данных весьма широка, поэтому в рамках главы мы рассмотрим только часто встречающуюся задачу разграничения доступа к данным со стороны пользователей и групп.

Практически все реляционные СУБД имеют развитые внутренние механизмы, позволяющие назначить пользователям права на уровне таблиц, столбцов и операторов SQL. Более того, эти механизмы стандартизованы на уровне соответствующих операторов SQL, таких как GRANT или REVOKE.

Использование встроенной системы безопасности позволяет, например, приказчику Сидорову делать выборки из таблицы заказов без возможности её модифицировать, при этом некоторые колонки недоступны. Однако, подобная детализация не является достаточной, и в общем случае проектировщику информационной системы требуется ограничивать доступ на уровне конкретных записей.

Один из подходов к решению задачи состоит в поддержке списка контроля доступа (ACL — Access Control List), нередко использующийся в рамках файловых систем. Список является ассоциацией между объектами доступа, субъектами и правами. Само наличие записи в списке уже может означать разрешение чтения состояния объекта субъектом доступа — пользователем, группой, приложением и т. д.

В проектировании нам поможет описанный в предыдущей главе подход к организации реестра объектов.

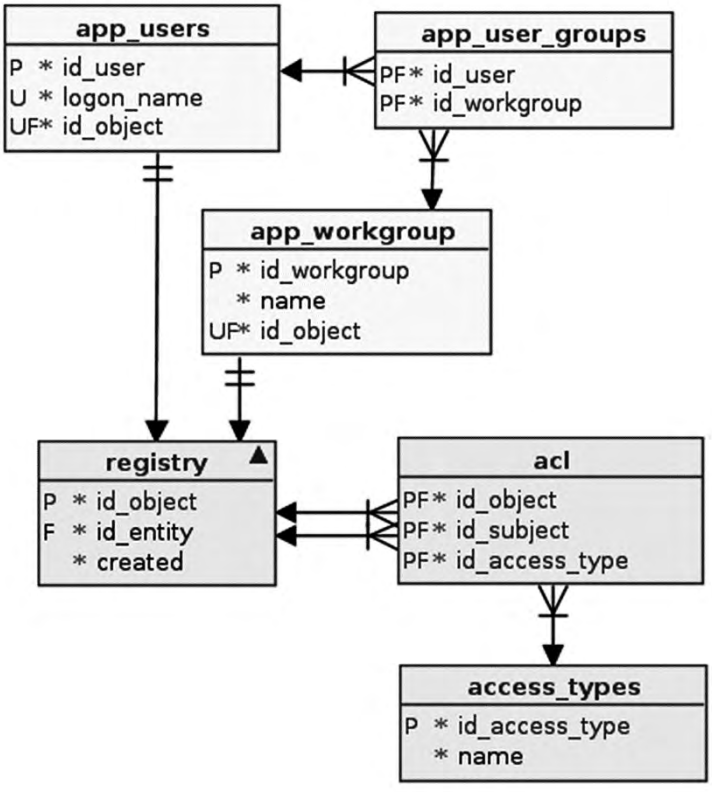

Рис. 44. Доступ к данным по списку контроля.

Особенностью схемы с использованием реестра является его двойственная роль: из реестра берётся как идентификатор объекта доступа, так и идентификатор субъекта. В роли последних выступают объекты сущностей (классов) app_user и app_workgroup.

Чтобы вывести список заказов с учётом прав доступа пользователя, используется следующий запрос.

```
SELECT o.*
FROM orders o
     INNER JOIN acl ON o.id_object = acl.id_object
     INNER JOIN app_users u ON u.id_object = acl.id_subject
WHERE u.logon_name = 'Иванов_Иван' AND
      acl.id_access_type = 1
      /* Положим, 1 - чтение, 2 - вставка и т.д. */
```

Если запрос кажется громоздким для частого использования в приложении, то оборачиваем его в вид.

```
CREATE VIEW orders_acl AS
SELECT o.*, u.logon_name, acl.id_access_type
FROM orders o
     INNER JOIN acl ON o.id_object = acl.id_object
     INNER JOIN app_users u ON u.id_object = acl.id_subject
```

С учётом возможного вхождения пользователя в группы вид усложнится:

```
CREATE VIEW orders_acl AS
SELECT o.*, u.logon_name, acl.id_access_type
FROM orders o
     INNER JOIN acl ON o.id_object = acl.id_object
     INNER JOIN app_users u ON u.id_object = acl.id_subject
UNION
SELECT o.*, u.logon_name, acl.id_access_type
FROM orders o
     INNER JOIN acl ON o.id_object = acl.id_object
     INNER JOIN app_workgroups w ON w.id_object =
acl.id_subject
     INNER JOIN app_user_group ug ON ug.id_workgroup =
w.id_workgroup
     INNER JOIN app_users u ON u.id_user = ug.id_user
```

В обоих случаях первоначальный запрос упрощается:

```
SELECT *
FROM orders_acl
WHERE logon_name = 'Иванов_Иван' AND
```

```
    id_access_type = 1
```

Кажется, что список контроля доступа — хороший способ, покрывающий любые ситуации, требующие разграничения действий на уровне строк таблиц. Недостатком способа является достаточно низкий уровень манипуляции. В информационных системах с развитыми ограничениями принято определять правила не на уровне строк, а по группам или функциональным признакам. Например, оператор Петров имеет право только создавать новые документы типа «Заказ», но не может их в дальнейшем ни читать, ни изменять. Или товаровед Васильева может производить все действия кроме удаления над товарами категории «Лаки краски».

Если вы используете механизм ACL в качестве самого нижнего уровня определения прав, то вышеприведённые примеры нужно будет реализовывать в виде функций, оперирующих непосредственно с содержимым таблицы `acl`. Например, при сохранении документа типа «Заказ» у пользователя «Петров» удаляются все права на данный документ. При изменении состава категории товаров «Лаки и краски» для пользователя «Васильева», соответственно, нужно обновить права на все эти товары.

Часто встречающийся вопрос: как определить права не на сам объект, а на его класс (сущность)? Ведь, например, право на создание объекта не может быть задано явным образом, так как объект ещё не существует.

Наиболее простым решением при наличии метаданных и реестра является связывание записи в реестре со строкой описания сущности в таблице `meta_entities`. Тогда можно задавать права непосредственно на все объекты данного класса. Но более гибкий вариант без взаимообратных связей — добавление в метаданные сущности типа «Описание сущности» (таблица `entity_descriptors`) и связывание с ней реестра.

Следствием из названного недостатка подхода — низкого уровня манипуляции правами доступа, являются возможные проблемы производительности.

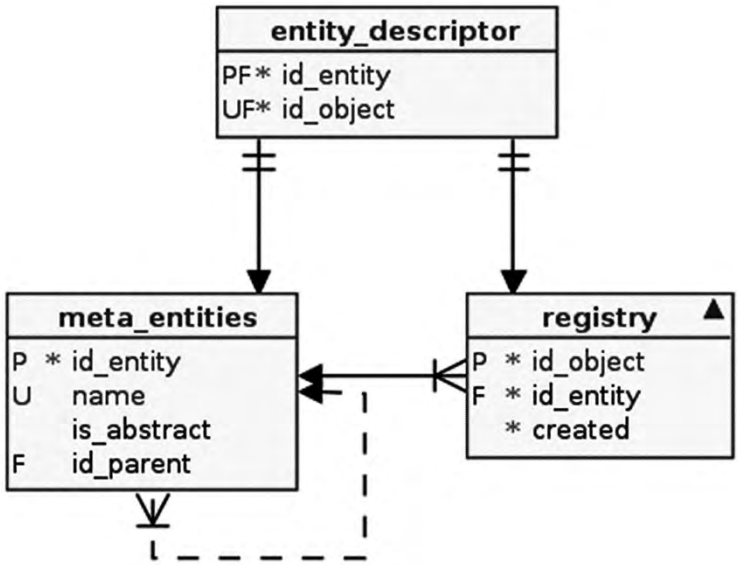

Рис. 45. Объект «Описание сущности» служит для определения прав уровня класса.

Действительно, список представляет собой по сути разреженную матрицу, столбцами которой являются субъекты, строками — объекты, а значениями — ещё один список из назначенных прав.

Давайте прикинем накладные расходы. Допустим, в транзакционной системе хранится 10 миллионов объектов, число пользователей не превышает 300. Требуется всего 4 уровня прав: чтение, создание (это права на класс, их не учитываем), модификация и удаление. Тогда при условии, что для каждого пользователя все права определены мы получаем предельную цифру в 9 миллиардов записей в таблице `acl`. Это большая цифра даже для аналитических систем, а в транзакционной она станет узким местом.

На практике, конечно, не требуется определять права всем пользователям ко всем объектам, и реальные цифры будут на два-три порядка меньше, то есть сравнимы с числом всех объектов в системе: как минимум какой-то пользователь их создавал, значит, скорее всего имеет право его читать. Но и 10 миллионов записей в `acl` могут ощутимо снизить производительность при интенсивном использовании коротких запросов, учитывающих контроль доступа. Каковы возможные способы оптимизации?

Прежде всего, обратим внимание на таблицу типов прав (access_type). Если число необходимых уровней невелико, то можно использовать непересекающиеся по битовой маске значения, являющиеся степенями двойки. В идентификаторе 32-разрядного целочисленного типа можно таким образом закодировать до 31 значения.

Например, пусть требуются 4 типа доступа: 0 — права чтения, 1 — создания, 2 — модификации и 4 — удаления объекта. Тогда проверка на наличие прав модификации (2) сводится к сравнению значения с числами, у которых установлен бит во втором разряде ($2_{10} = 0010_2$) из диапазона $[0..2^N-1]$, где N — число типов:

```
SELECT *
FROM orders_acl
WHERE logon_name = 'Иванов_Иван' AND
      id_access_type IN (2, 3, 6, 7)
```

Если СУБД поддерживает битовые индексы и соответствующие операции булевой алгебры, то проверка становится ещё проще:

```
SELECT *
FROM orders_acl
WHERE logon_name = 'Иванов_Иван' AND
      id_access_type & 2 = 2
```

Такой подход гарантирует, что таблица `acl` не будет расти с добавлением новых типов и уровней доступа.

Другое направление оптимизации, на которое следует обратить внимание, это оценка преобладания в системе прав разрешительного или запретительного характера. Если пользователи в своей массе имеют доступ ко всем объектам по умолчанию, за исключением некоторой их части, то следует «инвертировать» механизм ACL, полагая наличие соответствующей записи не разрешением, а запретом на доступ к соответствующему объекту.

Кроме того, возможна организация ACL исключительно на уровне групп. В этом случае не пользователи, а только их группы имеют права доступа на другие группы, объединяющие объекты. Например, все члены группы «Товароведы» имеют права редактирования всех объектов класса «Товар». Такой подход на порядки снижает длину таблицы `acl`, упрощает запросы и

увеличивает их производительность при условии, что сами группы не будут столь велики, чтобы включать в себя миллионы объектов.

Проектирование физического хранения

В предыдущих главах рассматривалось в основном логическое устройство системы, не зависящее, в общем случае, от уровня физического хранения. Зачастую такая независимость создаёт у программиста иллюзию ненужности рассмотрения деталей физического устройства и проектирования БД, отдавая этот аспект на откуп администраторам БД. Тем не менее, многих проблем можно избежать, если учитывать некоторые особенности физического уровня ещё на этапах планирования и разработки.

Проектирование физического хранения касается множества направлений и критериев, наиболее существенными из которых можно назвать следующие:

- организация долговременной памяти (дискового или иного хранилища);
- оценка требуемой оперативной памяти (ОЗУ);
- индексация данных;
- секционирование данных.

Все вышеозначенные пункты влияют, в первую очередь, на производительность вашей системы. Конечно, современные СУБД в той или иной степени обладают внутренними механизмами самонастройки, облегчающих работу администратора БД, но их возможности ограничены. С ростом объёмов БД, числа пользователей, структур (таблиц) и номенклатуры приложений полагаться на саморегуляцию приходится все в меньшей степени. Что же касается организации дисковой подсистемы или наличествующего объёма ОЗУ, то здесь, конечно, СУБД сама по себе не может сделать ничего.

Физическая организация памяти

Из того, что РСУБД на внешнем уровне оперирует таблицами, колонками и строками, никоим образом не следует, что соответствующие структуры в памяти СУБД также представляют из себя аналоги таблиц. Подробный разбор многочисленных способов физической организации данных приведён в монографиях [1,2,4,5], поэтому кратко рассмотрим только один из широко используемых — сбалансированные деревья (B-Trees).

Сбалансированные деревья отличаются от знакомых многим бинарных тем, что каждый родительский узел может иметь более двух ссылок на дочерние. Ссылки распределены по узлам таким образом, чтобы число уровней не превышало заданный порог N. Это гарантирует, что число считывания узлов из долговременной памяти при поиске значения в дереве не превысит N.

Строки таблиц, как правило, организованы в блоки более высокого уровня — страницы (pages), размер которых фиксирован или может задаваться глобальными настройками. Например, в SQL Server размер страницы равен восьми килобайтам. В свою очередь, страницы могут быть сгруппированы в единицы хранения более высокого уровня и размера, специфичных для разных СУБД. Введение страниц вызвано как минимум двумя причинами:

- при распределении памяти СУБД проще оперировать блоками фиксированного размера;
- внутри страниц можно резервировать пространство для вставки и обновления новых записей без перераспределения памяти, что увеличивает производительность.

Таким образом, на одной странице может помещаться разное количество строк таблицы в зависимости от их физического размера. Сами страницы организованы в дерево сбалансированного типа, порядок следования элементов нижнего уровня которого задаётся первичным ключом.

Таблица-кластер или *кластерный ключ* — организация физического хранения в виде сбалансированного дерева, в котором данные отсортированы по ключу.

Определение лишний раз объясняет, почему любая таблица должна быть определена вместе со своим первичным ключом. Но уже не на уровне теоретических рекомендаций, а с позиций практики физических реализаций. В противном случае требуется обоснование.

Из определения также следует, что таблица может быть организована в кластер единственным способом. Изменение колонки первичного ключа ведёт к полной физической реорганизации страниц.

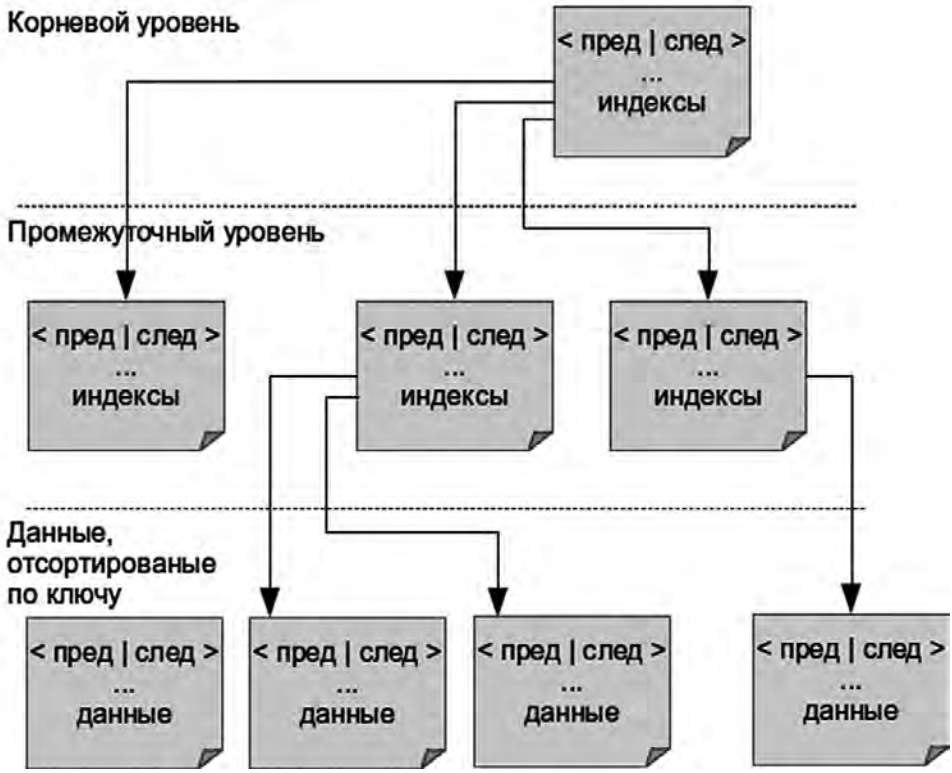

Рис.46. Пример страничной организации памяти в таблице-кластере

Если первичный ключ объявлен кластерным, что в некоторых СУБД принимается «по умолчанию», то наилучшим вариантом его реализации будет целочисленный автоинкрементный счётчик или генератор некоторых последовательных величин.

В отсутствие кластерного ключа страницы и записи располагаются в произвольном порядке, такая таблица называется «куча» (heap), что вполне соответствует беспорядочному распределению памяти.

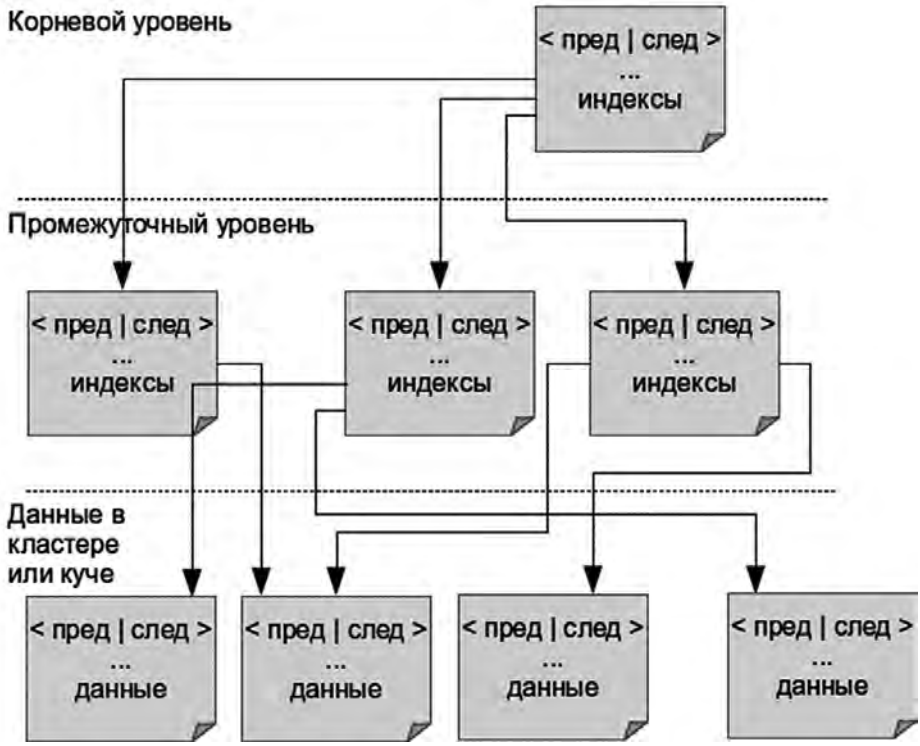

Рис.47. Пример организации некластерного индекса в таблице

У кучи могут быть некоторые преимущества, например быстрая вставка, если первичный ключ таблицы не объявлен кластерным, а его значения не последовательны. Но всякий раз игнорирование возможности кластерной организации таблицы нужно обосновывать. Прежде всего потому, что поиск по кластерному индексу гораздо более быстрый, чем по обычному.

В транзакционных приложениях, ориентированных на CRUD-логику, использование кластерного первичного ключа следует принять за правило. Напротив, в аналитических БД, где поиск по первичному суррогатному ключу не является частой операцией, а соединения между таблицами фактов редки, кластер может быть построен по неключевой колонке, являющейся наиболее часто используемой в запросах, например по дате факта.

Оперативная и долговременная память

В настоящее время объёмы ОЗУ превосходят объёмы дисковых массивов, использовавшихся ещё 10-15 лет назад. У программиста может создастся неверное впечатление о размытости границ между этими двумя видами памяти, тогда как важно понимать принципиальное отличие.

ОЗУ обеспечивает хранение только в режиме включённого устройства.

Долговременная память обеспечивает сохранение информации после выключения устройства (компьютера), в том числе при сбое электропитания.

Скорость доступа к данным, расположенным в долговременной памяти, на порядки медленнее по сравнению с ОЗУ. Современная оперативная память имеет время произвольного доступа около 5-10 наносекунд, тогда как жёсткие диски позиционируют свои считыватели только за 5-15 микросекунд, а твердотельные накопители (solid state drive) — немногим менее одной микросекунды. Разница на три порядка, то есть примерно в 1000 раз, является более чем существенной.

Одна из основных функций СУБД — обеспечивать долговременное хранение информации. Поэтому даже если объем ОЗУ настолько велик, что вся БД может целиком разместиться в оперативной памяти, следует отдавать себе отчёт: за фасадом запросов к СУБД идёт хотя и максимально оптимизированный, но все же непрекращающийся обмен с дисковой подсистемой, производительность которой может создавать узкие места. Если же объем ОЗУ недостаточен для размещения требуемой части данных в оперативном доступе, то обмен байтами с дисками будет происходить гораздо чаще.

Для наглядного сравнения производительности дисковой системы нам не нужен сервер, достаточно двух простых «винчестеров» на рабочем компьютере программиста, чтобы провести простой эксперимент. Я не привожу полных текстов программ испытания за исключением ключевых фрагментов, предоставляя вам самим повозиться на досуге.

- Создайте базу данных, разместив оба файла (данных и журнала) на одном физическом диске. Для этого взгляните на справку по команде CREATE DATABASE вашей СУБД, там должны быть опции физического размещения файлов. Например, для SQL Server

```
CREATE DATABASE testdb
ON PRIMARY (
  NAME = N'testdb',
  FILENAME =
N'C:\MSSQL\MSSQL10_50.SQL01\MSSQL\DATA\testdb.mdf',
  SIZE = 4096MB,
  MAXSIZE = UNLIMITED,
  FILEGROWTH = 1024MB
)
LOG ON (
  NAME = N'testdb_log',
  FILENAME =
N'D:\MSSQL\MSSQL10_50.SQL01\MSSQL\DATA\testdb_log.ldf',
  SIZE = 512MB,
  MAXSIZE = UNLIMITED,
  FILEGROWTH = 512MB
)
```

- Создайте в БД две одинаковые таблицы простой структуры, достаточно двух колонок: целочисленный идентификатор (он же первичный ключ) и название в виде строки длиной до 100 символов.
- Засеките время и заполните первую таблицу тестовыми величинами, например порядковый номер и название вида «Номер №NNN», где NNN — значение идентификатора. Обозначьте время выполнения, как T_1. Число строк должно быть достаточным, чтобы оценить разницу, например 100 000 или миллион.
- Засеките время и скопируйте данные из первой таблицы во вторую одним оператором SQL. Обозначьте время выполнения как T_2.

```
INSERT INTO Таблица2 SELECT * FROM Таблица1
```

Теперь удалите тестовую БД и создайте новую, у которой файлы данных и журнала будут располагаться на двух разных физических дисках. Повторив манипуляции с данными вы получите, соответственно, времена

выполнения T_3 и T_4. Сравнив их, вы увидите, что $T_1<T_3$, а $T_2<T_4$. **Разница в производительности может составлять более 100%, то есть в 2 раза**.

Объяснение полученного результата достаточно простое. Каждая операция вставки, как короткая, так и длинная, выполняется в транзакции. Механизм транзакций использует файл журнала для размещения не подтверждённых пока изменений. После подтверждения, которым в нашем случае является окончания выполнения оператора SQL, данные копируются из журнала непосредственно в файл данных, освобождая пространство другим транзакциям. Очевидно, что если оба файла находятся на одном диске, то одновременные чтение и запись будут невозможны, тогда как при размещении на разных дисках операции проводятся параллельно.

Приведённый пример максимально упрощён, в нем есть немало допущений. Так, перенос подтверждённых изменений из журнала в данные имеет гораздо более сложный механизм управления, специфичный для СУБД как на уровне стратегии (пессимистические стратегии ждут подтверждения, затем пишут в файл данных, оптимистические пишут в файл данных, затем, если случилась ошибка, откатывают изменения), так и на уровне режимов восстановления данных. В обычном случае данные из журнала переносятся в БД только после операции резервного копирования журнала или выдачи явной команды. Если БД находится в режиме репликации с другими, то очистка журнала также ждёт , пока подписчики получат все изменения. Часто возникающая при этом проблема — «распухание» файлов журнала, если один из подписчиков становится временно недоступен.

Да, в реальности происходящее внутри СУБД оказывается на порядок сложнее, иначе зачем были бы нужны администраторы БД. Но то, что даже в простом случае два диска вместо одного могут удвоить производительность вашей программы, надеюсь, вы уже поняли.

Дисковые массивы

С точки зрения эксплуатации пример с двумя дисками подходит разве что для автономных приложений и небольших групп пользователей. Как минимум, надо проводить резервное копирование и поставить для этого

третий диск, возможно внешний. В целом, системы внутреннего хранения, то есть размещение дисков внутри системного блока, подходят только для рабочих станций и небольших физических серверов рабочих групп и малых предприятий. В настоящее время имеет место мощная тенденция к виртуализации таких серверов в составе кластеров, представляющих доступ по сети (подробнее см. информацию по ЦОД — центрам обработки данных).

Однако, наш пример должен по меньшей мере вызывать вопросы: «Если дисков будет не два, а больше, увеличится ли производительность пропорционально их числу?» В общем случае ответ будет: «Да, увеличится, но не всегда кратно числу дисков».

Дисковый массив — два и более физических дисков, объединённых в едином устройстве, видимом на уровне операционной системы и СУБД как один и более логических дисков.

В англоязычной терминологии дисковый массив называется RAID, расшифровка этой аббревиатуры изменилась с «redundant array of inexpensive disks» в 1987 году до современного «redundant array of independent disks». Существуют многочисленные типы RAID, поэтому я не стану перечислять все уровни, существующих массивов, ограничившись наиболее распространёнными.

RAID0 или RAID нулевого уровня представляет собой два и более дисков, объединённых в единое целое с чередованием записи, но без резервирования. Чередование означает, что блоки одного и того же файла будет равномерно размещены по дискам, «размазаны». Таким образом, и чтение и запись файла будут идти параллельно, а общая производительность будет примерно кратна числу дисков по сравнению с системой с одним физическим диском. Однако, в массиве нет резервирования, поэтому отказ одного из дисков вызовет отказ и всего массива. Эффективная ёмкость хранения равна сумме ёмкостей используемых дисков.

RAID1 или RAID первого уровня представляет собой два резервирующих друг друга диска. Отказ одного из них не приводит к отказу массива. Скорость чтения примерно вдвое выше, чем у одиночного диска, но

скорость записи практически не отличается, потому что данные дублируются. Эффективная ёмкость хранения равна ёмкостей наименьшего из используемых дисков. Как правило, используются диски одинакового размера, но гибкие средства настройки позволяют, например, «отзеркалировать» один диск на два меньших.

RAID5 организован из трёх и более дисков, хранящих непосредственно информацию и контрольные суммы, позволяющие восстановить потерянные блоки. Скорость чтения примерно сравнима с таковой у трёхдискового RAID0, но скорость записи ниже за счёт необходимости расчёта и сохранения контрольных сумм. Эффективная ёмкость хранения равна сумме ёмкостей используемых дисков (они одинаковы) за вычетом одного диска. Относительная дешевизна RAID5 в сочетании с отказоустойчивостью сделали этот тип массивов очень популярным в 1990-е годы.

RAID10 является комбинацией RAID первого и нулевого уровней: каждый из дисков дублируется. Соответственно, минимальное число дисков равно четырём. Система отказоустойчива, скорость записи примерно кратна числу дисков, делённому на 2, скорость чтения — общему числу дисков. RAID10 может быть разбит на несколько независимых секций, называющихся логическими единицами, подмассивами (LUN — Logical Unit Number), позволяющих организовать раздельный доступ к данным со стороны СУБД и приложений. Эффективная ёмкость хранения равна половине суммы ёмкостей используемых дисков.

В целом современные производители СУБД, особенно для транзакционных приложений, рекомендуют ориентироваться на RAID10. Несмотря примерно на 30% большую стоимость по сравнению RAID5, обеспечивается более высокая производительность и прозрачное разделение на логические подмассивы. Эта опция может быть очень важна, если внешний массив используется несколькими физическими или виртуальными серверами одновременно (так называемый разделяемый массив). Однако для аналитической обработки, характеризующейся, как мы помним, операциями чтения, RAID5 по-прежнему может обеспечивать высокую скорость доступа к данным, за исключением стадий заполнения и обновления хранилищ.

Оперативная память

В этом случае я вас огорчу: чёткого алгоритма предварительного определения требуемого объёма ОЗУ не существует. Нельзя сказать что-то вроде «возьмите объем БД и умножьте на 0,1». Многое также зависит от типа обработки. Для транзакционных приложений будет достаточно, если ОЗУ вместит кеш оперативных данных. Для аналитических — даже размещение БД в ОЗУ целиком может не дать ожидаемого ускорения.

Поэтому я советую данный вопрос решать, во-первых, на основе тестов, а во-вторых, поскольку тесты эти будут специфичными для каждой СУБД, с помощью администратора БД.

В качестве примера приведу достаточно простой способ, подходящий для транзакционного приложения, работающего с Microsoft SQL Server. Эта СУБД предоставляет в распоряжение администратора большое число статистических параметров (Performance counters), которые можно включить и наблюдать динамику непосредственно через монитор производительности Windows (Windows Performance Monitor).

Если выбрать из списка в группе Buffer Manager показатель «Buffer cache hit ratio», показывающего процент страниц, найденных в кэше и, таким образом, считанных без обращения к диску, то его изменения за продолжительный период не должны выходить за пределы 99%. Это означает, что из 100 обращений к данным, СУБД должна в 99 случаях считывать их из ОЗУ, а не с диска.

Если этот показатель стабильно колеблется, снижаясь до величин меньших 99%, то необходимо увеличить объем ОЗУ. Перед этим не забудьте проверить, сколько памяти выделено в системе для СУБД, ведь многим из них задают конфигурацию с верхними ограничениями объёма используемой физической памяти, предотвращая уход системы «в своп»[18]. Думаю, все же без администратора БД в этих вопросах не обойтись.

[18] От англ. swapping — обмен между ОЗУ и файлом подкачки в механизме реализации виртуальной памяти операционной системы

Индексация данных

По большому счёту, создание индексов, увеличивающих производительность запросов к СУБД — это работа администратора, который руководствуется статистикой их выполнения по всей БД в целом.

Также существуют случаи, когда поставщик программного продукта просто не может заранее создать даже необходимые индексы. Например, в таблице продаж одна из колонок является внешним ключом для таблицы товаров, в другая — для таблицы покупателей. Если у предприятия, покупающего тиражируемую программную систему, достаточно много как товаров, так и клиентов, то оба индекса, построенные по значениям этих колонок, будут использоваться СУБД при выполнении запросов. Однако, если предприятие продаёт многие тысячи товаров десятку клиентов или, наоборот, сбывает десяток товаров тысячам покупателей, то один из индексов окажется бесполезным. Эту ситуацию можно определить только в период предпродажной подготовки и внедрения программной системы на предприятии, но подготовиться к ней следует заранее.

Тем не менее, сказанное выше — не повод, чтобы поставлять вашу систему в «голом» виде, то есть с таблицами, ключами и прочими ограничениями целостности, но без индексов. Как минимум, следует проиндексировать внешние ключи, если вы предполагаете, что в большинстве случаев значение в колонке будет плотным, а индекс, соответственно, селективным.

Плотность — отношение количества уникальных значений в одной или нескольких колонках к общему числу значений (т. е. строк в реляционной таблице).

Селективность индекса — величина обратная плотности, показывающая, какое количество строк таблицы может быть выбрано в среднем по одному значению.

Из этих определений следует что максимальной плотностью, равной единице, и, соответственно, наилучшей селективностью обладают первичные ключи и уникальные индексы по другим ключам таблицы.

Например, в таблице документов колонка «Состояние» может принимать 3 значения из связанной таблицы-справочника «Состояния документов». Если в базе данных имеется 100 000 документов, то плотность значений в колонке состояний будет равна всего 0,00003, а попытка поиска по одному из значений состояний в таблице документов будет в среднем возвращать треть содержимого таблицы, более 33 тысяч строк. Такой индекс является бесполезным на практике, лишь занимая место на диске — СУБД не использует его при исполнении запросов.

Построение индексов также касается встроенных запросов приложения.

> *Встроенный запрос* — запрос фиксированной структуры с заданным числом параметров, значения которых могут быть определены непосредственно перед выполнением

Например, если в приложении имеется форма поиска человека по фамилии, то имеет смысл проиндексировать соответствующую колонку таблицы. Если же кроме фамилии используется ещё и дата рождения, то следует создать один *составной индекс* по этим двум колонкам или два простых индекса.

Принцип оценки селективности также относится и ко встроенным запросам. Другой широко известный пример — колонка, хранящая значение пола человека. Её также бесполезно индексировать, а любые запросы с фильтрацией исключительно по этому признаку будут производиться путём сканирования всей таблицы.

Если в индекс попадает две и более колонок, то их рекомендуемый порядок следования от более плотной к менее плотной.

Важно понимать, что индексация не является бесплатным пирожным. Каждый индекс:

- занимает дисковое пространство, увеличивая общий размер БД и, соответственно, число операций чтения/записи;
- требует своего обновления при изменении данных, что замедляет операции вставки, модификации и удаления.

Таким образом, индексирование должно быть основано на расчётах и проверках, что ускорение выполнения запросов одних приложений не вызовет деградацию производительности у других. В этом случае роль администратора БД возрастает, так как в отличие от программиста отдельного приложения он оценивает использование БД в целом со стороны всей совокупности эксплуатируемых приложений.

Секционирование данных

Секционирование представляет собой разделение одной таблицы на несколько секций (разделов, partitions) по заданному на одной или нескольких колонках признаку.

Например, таблица продаж имеет колонку «Дата продажи». Если предприятие достаточно крупное, то таблица может расти на десятки и сотни тысяч строк в день. Нетрудно посчитать, что за 3-4 года эксплуатации таблица может вырасти до сотен миллионов строк. Вставка каждой новой записи будет вызывать обновление всех индексов. При этом в оперативном контуре транзакционные приложения используют только относительно недавние данные. Зачем этим приложениям работать со всей совокупностью данных таблицы?

Решением в такой ситуации является секционирование таблицы на основе значений колонки «Дата». Предположим, что доступным для изменений в оперативном контуре является период последнего месяца, а для финансового учёта — последний квартал. Тогда разделение на секции может выглядеть следующим образом.

Основной смысл секционирования состоит в физическом размещении частей одной таблицы по разным файлам, взаимодействие с которыми на уровне файловой системы и дискового массива осуществляется параллельно. В этом случае приложения оперативного контура, добавляющие, изменяющие и удаляющие данные работают не с единственным объёмным файлом всех продаж, а с одним файлом небольшого размера, хранящим данные последнего месяца. Аналогично, финансовые приложения работают только с тремя небольшими файлами. Это существенно уменьшает количество необходимых операций ввода/вывода.

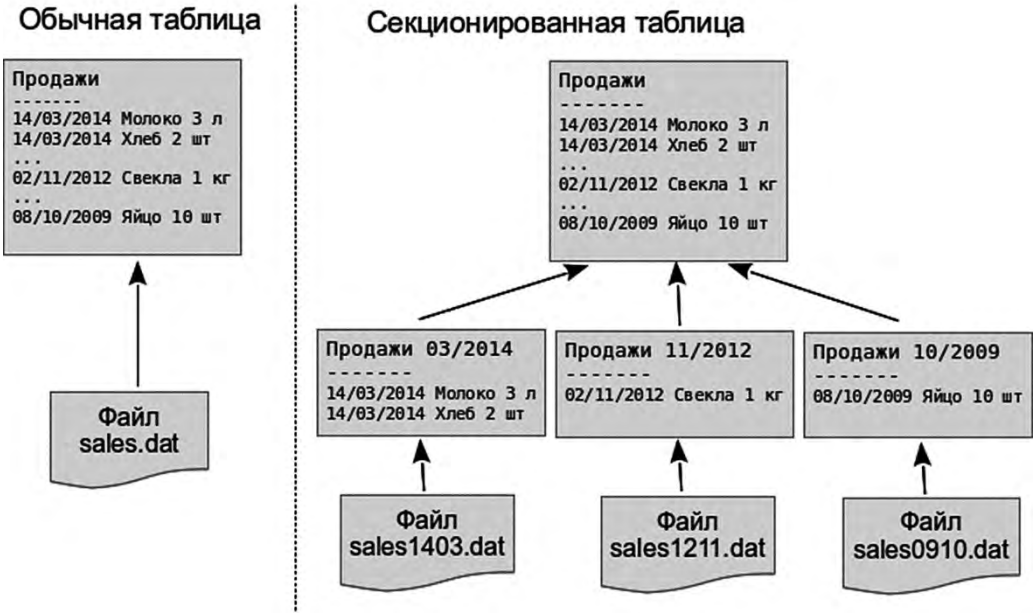

Рис. 48. Пример секционирования таблицы

Если ещё немного приблизить пример к реальности, то для секционирования в этом случае хватило бы 4 секций: секции 1, 2 и 3, соответствующие последним месяцам и кварталу, и секция 4, содержащая все остальные данные. В такой схеме необходима ежемесячная ротация данных: секция «3» сливается с четвёртой, «1» и «2» становятся, соответственно, «2» и «3», добавляется новая секция «1».

В наиболее развитых СУБД секционирование может быть определено непосредственно на уровне таблицы независимым для приложения способом. Запросы приложения по-прежнему обращены к самой таблице независимо от того, секционирована она или нет. Таков, например, внутренний механизм поддержки секционирования в Microsoft SQL Server посредством функций и схем секционирования, или в PostgreSQL через реализацию наследования.

Если СУБД не поддерживает явное секционирование, то во многих случаях будет работать достаточно простой старинный метод с использованием множества однотипных таблиц и горизонтально объединяющего их вида (view). В нашем примере для этого пришлось бы создавать таблицы одинаковой структуры sales1403, sales1402, sales1401 и

т д. Над этими таблицами создаётся вид sales, включающий их объединение.

```
CREATE VIEW sales (…)
AS
SELECT … FROM sales1403
UNION ALL
SELECT … FROM sales1402
UNION ALL
SELECT … FROM sales1401
…
```

Основной недостаток такого подхода — более низкий уровень абстракции и, как следствие, несколько большая зависимость приложений от физического проектирования. Так дополнительные операции по поддержке секций (слияние, разделение) должны проводиться не самой СУБД в терминах соответствующих команд, а администратором БД посредством множества команд SQL. Кроме того, если СУБД не поддерживает механизм триггеров для видов, то непосредственные операции вставки, модификации и удаления над sales могут оказаться невозможными, приложению потребуется самостоятельно выбирать соответствующую нижележащую таблицу salesYYMM.

Неполно структурированные данные и высокая нагрузка

Относительность понятия высокой нагрузки

Тема высоконагруженных приложений может возникать в совершенно разных контекстах, но последнее десятилетие наиболее частое упоминание было связано со службами и сайтами глобальной сети, ориентированных на массового пользователя.

Однако, если поискать информацию в Интернет, то сложно найти даже собственно определение высокой нагрузки. Интуитивно понятно, что сервисы типа Facebok, обслуживающие сотни миллионов пользователей, относятся к данной категории приложений. А если пользователей немного, или в их качестве выступают периферийные устройства, генерирующие большой поток информации, можно ли отнести нагрузки к высоким?

Возникла ситуация, подобно описанной в главе «Большие данные как состояние отрасли», когда в отсутствие формальных критериев смысл термина начинает размываться, превращаясь в маркетинговый инструмент для обоснований освоения бюджетов проектов.

> *Высоконагруженная программная система* — система, для обеспечения заданных эксплуатационных характеристик которой требуется максимальная (или близкая к максимальной) эффективность использования оборудования.

Не претендуя на полноту, данное определение является достаточно хорошим, рабочим, позволяя отнести к высоко нагруженным не только службы очередной социальной сети, но и любые другие системы, чьи возможности ограничены аппаратурой. Например, встроенные системы. Разработать программное обеспечение устройства, процессор которого работает на частоте 10 мегагерц при доступном ОЗУ в 128 килобайт, обрабатывающем тысячи пакетов данных в секунду ничуть не проще, чем для веб-службы простановки штампов времени на документах, обслуживающей сотни тысяч запросов за тот же промежуток времени.

Чтобы, как говорится, почувствовать разницу, возьмём вполне конкретный пример автоматизированной системы бронирования и продажи билетов национального уровня «Экспресс-2», эксплуатировавшейся в СССР и РФ с 1982 по 2005 годы. В цифрах на 1988 год система представляла собой:

- 15 региональных центров, объединённых единой сетью передачи данных;
- 2000 терминалов по продаже и бронированию билетов;
- заданная нагрузка — 15 заказов/сек.

Первоначально в системе использовалась отечественная аппаратная платформа ЭВМ ЕС (ряд 3), позднее, в 1990-х, мэйнфреймы IBM System/370, аналогами которых они являлись. Приведём некоторые характеристики ЭВМ ЕС 1046:

- процессор с частотой 1,3 МГц;
- ОЗУ 4-8 мегабайт (цикл 1 микросекунда);
- пропускная способность каналов ввода/вывода 8,1 мегабайт/сек.

Одна ЭВМ с указанными характеристиками поддерживала работу сотен терминалов, обменивалась данными по единой сети и могла проводить несколько заказов (бронирование или продажу) в секунду.

Можно ли назвать систему «Экспресс-2» высоконагруженной? Безусловно, несмотря на весьма скромные по современным меркам показатели по числу выполняющихся за секунду транзакций.

Для сравнения доступных разработчикам мощностей, возьмём аналогичные характеристики сервера для дома и малых предприятий HP MicroServer N40L:

- процессор с частотой 1,5 ГГц;
- ОЗУ 4-8 гигабайт (DDR3-800, цикл 15 наносекунд);
- пропускная способность каналов ввода/вывода (накопители SATA) 1,5 гигабайта/сек.

Практически по всем перечисленным показателям, маленький сервер из 2010 года превосходит универсальную ЭВМ 1984 года на три порядка, то есть в 1000 раз. Оставляя за рамками вопросы пропускной способности сети передачи данных, надёжности и отказоусточивости ЭВМ, возможности обслуживания сотен терминалов, теоретически, наш небольшой сервер должен при использовании аналогичного программного обеспечения обслуживать поток около 1000 заказов в секунду.

Смею вас заверить, это большая цифра, соответствующая уровню международных систем бронирования. И чтобы обеспечить её на таком сервере даже в режиме испытаний группе разработчиков придётся изрядно попотеть как в проектировании, так и в реализации, в прямом смысле выжимая байты и герцы из этого «железа».

Для чего приведён пример? Если вы сталкиваетесь с проблемами производительности на конкретной аппаратной конфигурации, прежде чем

говорить о высоких нагрузках вспомните о старом советском мэйнфрейме ЕС 1046 и приблизительно оцените, сколько аналогичных транзакций в секунду должна была бы выполнять ваша система, чтобы по определению и по праву называться высоконагруженной.

Особенности использования РСУБД и НСМД (NoSQL)

В главе «О применимости NoSQL» было отмечено, что одна из областей применения технологии — первичный сбор неполно структурированных данных. Одной из причин выбора NoSQL-решения иногда называют необходимость быстрой вставки и редактирования документов, структура которых сложна и может меняться.

Действительно, реляционный подход к моделированию и сопутствующая ему нормализация данных раскладывают один сложный документ во множество отношений, потенциально снижая производительность транзакций. При высокой нагрузке атомарная операция вставки записей в десятки связанных таблиц может стать узким местом.

Одно из главных правил использования РСУБД в условиях высокой нагрузки — **минимизация необходимых соединений в запросах.** В идеале, запрос не должен иметь никаких соединений вообще, за исключением, возможно, небольших таблиц-справочников.

Тем не менее, реляционная СУБД может успешно справиться с задачами, требующими работы в условиях высокой нагрузки и неполно структурированных данных. Для этого желательна поддержка встроенных типов хранения XML, JSON и других форматы НСМД. Так, например, тип данных XML для колонок поддерживается в СУБД SQL Server, PostgreSQL и других.

Общий подход заключается в выделении из хранимых объектов (документов) некоторых атрибутов, являющихся ключевыми для поиска. Они сохраняются в соответствующих колонках таблицы. Остальная часть документа заносится в колонку типа XML.

Для примера приведу проект мониторинга в реальном времени проезда автомобилей через портал на скоростной магистрали. Оборудование

фиксирует момент проезда, делает фотографию и распознает номер машины. Данные заносятся в базу, где хранятся некоторый период времени до перемещения в централизованное хранилище. При интенсивном движении поток вставки на одном портале достигает десятков документов в секунду, размер каждого, включая изображение варьируется в пределах 1 мегабайта. Для БД централизованного хранилища эти цифры усредняются за период хранения (буферизации) и умножаются на число порталов, составляющих десятки и сотни единиц.

Например, при средней пропускной способности в 5 машин в минуту, за одни сутки буферизации при 200 порталах в центральной СУБД получаем достаточно интенсивный поток со средней скоростью около 16 документов в секунду.

Хотя нельзя назвать усреднённое число в 16 документов/сек действительно высокой нагрузкой, проблема кроется не в количестве, а в объёме транзакции. С учётом размера изображений, база данных растёт примерно на полтора терабайта в день.

В качестве ключевых атрибутов выступают внутренний идентификатор записи, время прохода, номер машины или признак «не распознан», скорость машины. В остальной части документа XML содержит разнообразные данные телеметрии оборудования и диагностику. Наконец, колонка бинарного типа хранит изображение, коих может быть два. Если фронтальная камера выдаёт снимок, по которому номер не распознаётся соответствующей подсистемой, то операцию повторяет тыловая камера.

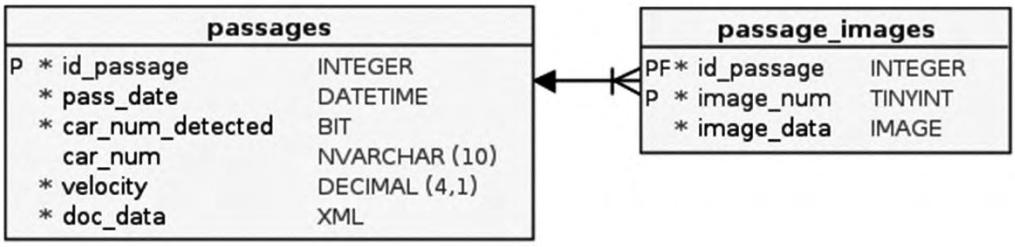

Рис. 49. Схема хранения неполно структурированных данных.

Схема представляет собой достаточно типовой образец скрещивания реляционного подхода с неполно структурированными моделями.

Действительно, хранить все данные в реляционном виде не имеет смысла по трём причинам:

- атрибуты и связи сущности «Проход автомобиля» многочисленны, в нормализованном виде они должны быть разнесены по многим таблицам, а в денормализованном — создавать вместо одной несколько записей с многочисленными колонками. Это существенно повлияет на скорость вставки;
- номенклатура атрибутов может меняться. Например, для разных марок аппаратуры и типов дорог телеметрия будет отличаться;
- для оперативного управления требуется лишь поиск по нескольким ключевым атрибутам.

Синтезированная на рис. 49 схема отвечает перечисленным требованиям: она обеспечивает высокую производительность, атрибутика документа может меняться без модификации структур БД, обеспечивается быстрый поиск и выборки по вынесенным из XML-документа в реляционную таблицу атрибутам.

Использование XML в данном случае не является принципиальным, с тем же успехом колонка может хранить JSON. Возможно также использовать собственный внутренний текстовый или двоичный формат, в случае если РСУБД не поддерживает соответствующие встроенные типы данных.

Вторая важная особенность приведённой схемы — наличие отдельной таблицы для хранения изображений. Дело здесь не в нормализации, даже если каждой записи о проходе автомашины соответствовала бы только одно фото, то все равно следовало бы вынести эту информацию в отдельную таблицу. Почему?

В главе «Проектирование физического хранения» говорилось о возможностях распараллеливания операций с файлами БД. Явное разделение на две таблицы позволит оптимизировать хранение, предписав СУБД разместить каждую из них в отдельном физическом файле. Вот так, например, это можно сделать средствами SQL Server.

```
/* Создаём файловую группу */
```

```
ALTER DATABASE pass ADD FILEGROUP fg_pass_images;

/* Добавляем в группу физический файл */
ALTER DATABASE pass ADD FILE (
  NAME = 'images_data',
  FILENAME = 'G:\imgdata\images_data.ndf')
TO FILEGROUP fg_pass_images;

/* Создаём таблицу изображений на отведенной файловой группе
*/
USE pass;
CREATE TABLE passage_images (
  id_passage integer NOT NULL ,
  image_num  tinyint NOT NULL ,
  image_data image NOT NULL ,
  CONSTRAINT PK_passage_images
    PRIMARY KEY CLUSTERED (id_passage, image_num)
) ON fg_pass_images;
```

СУБД PostgreSQL также позволяет задать для создаваемой таблицы её физическое пространство размещения (tablespace).

```
/* Создаём табличное пространство */
CREATE TABLESPACE images_data LOCATION '/mnt/imgdata/';

/* Создаём таблицу на пространстве */
CREATE TABLE passage_images (
  id_passage integer  NOT NULL ,
  image_num  smallint NOT NULL ,
  image_data bytea    NOT NULL ,
  CONSTRAINT PK_passage_images
    PRIMARY KEY (id_passage, image_num)
) TABLESPACE images_data;
```

Аналогичным образом была организована схема данных в другом веб-приложении. Разница заключалась лишь в отсутствии таблицы хранения изображений, но зато пиковые нагрузки CRUD запросов к документам достигали порядка единиц тысяч в секунду.

Достоинства описанного гибридного подхода очевидны: сохранив гибкость и быстродействие реляционных запросов мы можем оперировать документами сложной структуры в условиях относительно высокой нагрузки. Большую роль в производительности также играет проектирование физического хранения. Тем не менее, оно было бы

малоэффективным без предварительного логического проектирования с учётом особенностей и ограничений реляционной модели.

Определенным недостатком является некоторая жёсткость критериев выборки в запросах, ограниченных вынесенными на уровень колонок атрибутов документа. Для расширения потребуется добавить соответствующие столбцы.

Следует заметить, что реляционные СУБД, поддерживающие XML на уровне встроенного типа, позволяют осуществлять поиск и по всей совокупности хранящихся в колонке документов, но с худшей производительностью, чем по столбцам таблицы. Соответствующая функциональность аналогична предлагаемой NoSQL СУБД. О работе с XML будет рассказано в специальной главе раздела «Программирование».

Нужно ли моделировать?

Если вы поставлены перед необходимостью создания нового приложения на уже существующей базе данных в эксплуатации, то вопрос о моделировании данных, скорее всего, не встанет. Процесс ограничивается изучением существующей структуры, если она документирована, или археологией, в противном случае. Добавляются виды и хранимые процедуры, возможно, нескольких колонок и таблиц.

Иное дело, когда вы разрабатываете новое приложение, автономное или, что встречается чаще в корпоративном софтостроении, интегрирующееся с имеющимися системами на уровне БД и других служб.

При рассмотрении подхода разработки по моделям [13] была высказана многократно подтверждённая на практике мысль, что далеко не все программисты способны даже просто поверить в то, что какие-то картинки со стрелками могут заменить написание кода руками. Однако, моделирование баз данных представляет, на мой взгляд, наиболее проработанную и обладающую богатым арсеналом инструментов область пересечения информационных технологий с автоматизированным проектированием.

Несмотря на наличие нескольких нотаций (IDEF1, ER, метод Баркера и др.), все они имеют принципиальное сходство и содержат относительно

небольшое число элементов (сущности, связи, атрибуты...). Если взять для сравнения гротескный эклектический UML, то количественная разница будет сразу заметна.

Использование графического инструментария для моделирования (автоматизированного проектирования) имеет два основных назначения:

- получение целостной картины всей базы данных с возможностью непосредственной детализации
- генерация SQL-скриптов (сценариев инициализации) для одной или нескольких целевых СУБД

Конечно, если приложение оперирует десятком таблиц, то можно и моделировать, и хранить «картинку» с полной детализацией непосредственно в дефрагментированном мозге программиста или даже в коллективном сознании всего трудового коллектива. Однако, уже при 30-50 таблицах начинаются проблемы, прежде всего при изменении структуры. Назначение колонок и других элементов схемы тоже надо где-то описывать, и вскоре рядом с проектом разработки приложения в инструментальной среде будет вестись какая-нибудь электронная таблица, постоянно запаздывающая в связи с очередными изменениями. В варианте ведения описаний непосредственно в программном коде тоже возникает немало проблем как дублирования текстов комментариев в файлах программ разных слоёв, так и их ручного переноса при реструктуризации.

Давайте рассмотрим простой пример, показывающий отличие ручного описания от автоматизированного проектирования базы данных. В качестве инструментария возьмём бесплатный pgModeler (http://www.pgmodeler.com.br), распространённый в процессах разработки для СУБД PostgreSQL.

Также следует отметить, что многие среды разработки, такие как Visual Studio или Delphi содержат урезанные по возможностям встроенные средства для работы с таблицами, колонками и прочими элементами модели непосредственно в соединении с СУБД.

Моделирование против ручного кодирования: пример

Предположим, в приложении необходимо вести учёт контактов и связанных с ними компаний: адреса, телефоны, «явки».

Программист №1 (П1) владеет средствами автоматизированного проектирования. Открыв среду pgModeler, он добавляет на экран две новых таблицы — «Компании» и «Контакты», в свойствах создаёт колонки соответствующих типов, затем соединяет их отношением «один-ко-многим». Получается примерно такая картинка (см. рис. 50).

Нажав на панели слева кнопку «Source», можно увидеть SQL-код, генерируемый средой.

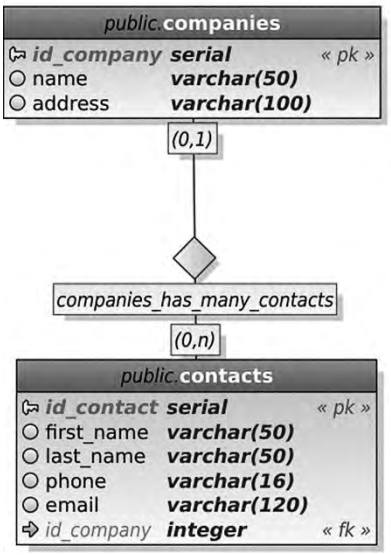

Рис.50. Модель на первом этапе

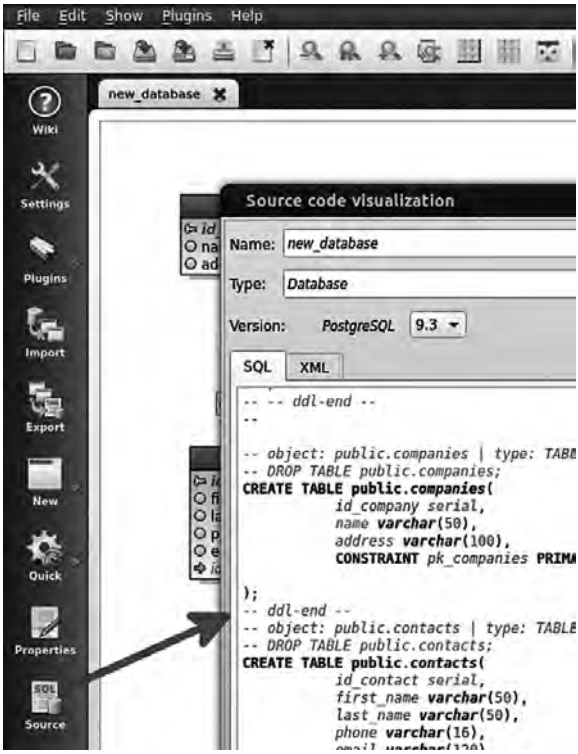

Рис.51. Просмотр генерируемого кода

Сгенерированный код можно скопировать в буфер обмена и непосредственно выполнить в консоли SQL-запросов СУБД.

Можно поступить иначе: экспортировать (кнопка «Export») нашу схему непосредственно на сервер, определив соответствующие параметры соединения с СУБД.

Рис.52. Экспорт схемы БД

Необходимо также экспортировать схему в файл, сохранить его под заданным именем, пусть это будет `create_model.sql`, и добавить в соответствующее депо (repository) вашей системы контроля версий исходников программ (надеюсь, вы уже используете SVN, Git или другие средства контроля).

После всех проведённых П1 манипуляций в СУБД будут созданы две таблицы с соответствующими колонками, двумя первичными и одним внешним ключом.

Программист №2 (П2) предпочитает работать непосредственно с исходными кодами на SQL. Он просто открывает в обычном редакторе с подсветкой синтаксиса новый файл и пишет примерно такой текст.

```
CREATE TABLE public.companies(
      id_company serial,
      name varchar(50),
      address varchar(100),
      CONSTRAINT pk_companies PRIMARY KEY (id_company)
);

CREATE TABLE public.contacts(
      id_contact serial,
      first_name varchar(50),
      last_name varchar(50),
      phone varchar(16),
      email varchar(120),
      id_company integer,
      CONSTRAINT pk_contacts PRIMARY KEY (id_contact)
);

ALTER TABLE public.contacts
  ADD CONSTRAINT fk1_companies_contacts
  FOREIGN KEY (id_company)
  REFERENCES public.companies (id_company);
```

Затем файл также сохраняется под именем `create_model.sql`, добавляется в систему контроля версий и выполняется с консоли на целевом СУБД-сервере. Результат тот же.

Как можно заметить, если результат тот же, то зачем же тратить время, добавляя в процесс дополнительный инструмент проектирования, соответствующую формальную стадию и необходимость инструмент осваивать? Ответ неочевиден, если на этом этапе ваша разработка схемы базы данных заканчивается.

Но для любого софтостроения характерна необходимость внесения изменений.

Предположим, на следующем этапе выяснилось, что адрес нужно хранить в виде трёх колонок: непосредственно уличный адрес, город и почтовый индекс.

П1 открывает модель, в свойствах таблицы добавляет две колонки «Город» и «Почтовый индекс» соответствующих типов. Вновь экспортирует схему в уже имеющийся файл `create_model.sql`, и пересоздаёт структуру на СУБД-сервере.

П2, пожимая плечами, открывает в редакторе исходный файл `create_model.sql`, и дописывает в оператор создания таблицы две строки

```
CREATE TABLE public.companies(
      id_company serial,
      name varchar(50),
      address varchar(100),
      city varchar(30),
      postal_code varchar(10),
      CONSTRAINT pk_companies PRIMARY KEY (id_company)
);
```

П2 сохраняет файл, прогоняет его в SQL-консоли и, если не было ошибок, также архивирует в депо.

Как можно заметить, на втором этапе никакой разницы в операциях практически нет, трудоёмкость уравнялась.

На третьем этапе оказывается, что адреса у компании и контактного лица могут быть разными. Для компании надо указывать юридический адрес, а для контакта в этой же фирме — адрес непосредственного местонахождения конторы. И эти два адреса могут быть разными.

Оба программиста достаточно квалифицированны, чтобы на корню подавить в себе желание продублировать соответствующие колонки адреса в таблице «Контакты». Они создают новую таблицу «Адреса» и ссылаются на неё, соответственно, из таблиц компаний и контактов. Причин у такого решения много. Например, если не рассматривать адрес в качестве отдельной сущности, то в дальнейшем при изменении структуры адресов, делать это придётся дважды, как минимум. Если к тому времени вам не придёт в голову умножить свои плодотворные усилия повторением атрибутов в таблицах сотрудников, клиентов, филиалов и т.п.

П1 открывает среду проектирования, добавляет новую таблицу и атрибуты, рисует две связи, удаляет одну старую связь. Дальше как обычно: экспортируем новую схему в файл и непосредственно на сервер.

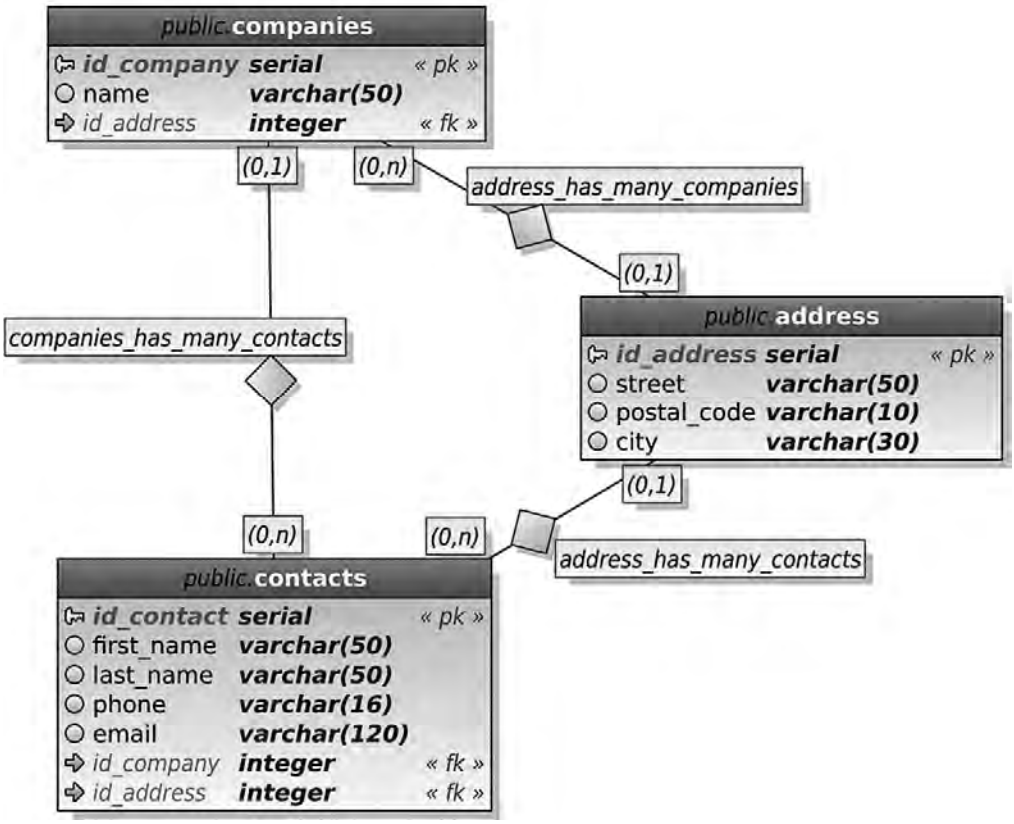

Рис.53. Модель на третьем этапе

П2, почесав затылок, открывает файл и начинает править код. Так, во-первых, таблицу «Адреса» нужно создавать раньше остальных, чтобы добавлять отграничения внешних ключей сразу после создания двух связанных таблиц, сохраняя структурность линейного текста. Во-вторых, надо удалить соответствующие колонки из оператора создания таблицы компаний. В-третьих, надо добавить в обе таблицы колонки `id_address` для внешнего ключа. Кстати, они будут NULL или NOT NULL? Поставим NULL, пусть не всегда вводят адрес. Да, надо ещё добавить комментарии для структурности, а то там потом не найдёшь нужное место глазами без глобального поиска.

В итоге маленький прежде файл распухает на глазах и приобретает новый вид.

```
-- Адреса
CREATE TABLE public.address(
      id_address serial,
      street varchar(50),
      postal_code varchar(10),
      city varchar(30),
      CONSTRAINT pk_address PRIMARY KEY (id_address)

);

-- Компании
CREATE TABLE public.companies(
      id_company serial,
      name varchar(50),
      id_address integer,
      CONSTRAINT pk_companies PRIMARY KEY (id_company)
);
ALTER TABLE public.companies
  ADD CONSTRAINT fk_address_companies
  FOREIGN KEY (id_address)
  REFERENCES public.address (id_address);

-- Контакты
CREATE TABLE public.contacts(
      id_contact serial,
      first_name varchar(50),
      last_name varchar(50),
      phone varchar(16),
      email varchar(120),
      id_company integer,
      id_address integer,
      CONSTRAINT pk_contacts PRIMARY KEY (id_contact)
);
ALTER TABLE public.contacts
  ADD CONSTRAINT fk1_companies_contacts
  FOREIGN KEY (id_company)
  REFERENCES public.companies (id_company);
ALTER TABLE public.contacts
  ADD CONSTRAINT fk_address_contacts
  FOREIGN KEY (id_address)
  REFERENCES public.address (id_address);
```

На четвёртом этапе оказывается, что один и тот же контакт может быть представителем двух и более компаний одновременно.

П1 делает достаточно большую правку модели малыми усилиями: удаление связи между контактами и компаниями, добавление ассоциативной таблицы «Компании контактов» связей «многие-ко-многим», дорисовка двух связей к ней из таблиц «Компании» и «Контакты», соответственно. Прежняя связь с адресом также уходит из таблицы контактов в «Компании контактов» (может и сохраняться в качестве личного адреса).

Чтобы убедиться в согласованности получившейся схемы БД, П1 перед сохранением нажимает кнопку проверки («Validate»).

П2 вынужденно начинает делать то, что на жаргоне называется «лопатить код», неизбежно внося в него человеческие ошибки, потому что **правка текста в одном месте** (добавление или удаление связей) **должна быть строго согласована с соответствующей правкой в других его местах** (добавление/удаление столбцов, ограничений, индексов, содержащих удалённые колонки и т.д.).

Теперь представьте типовую ситуацию - появился третий программист (П3) в подмогу одному из двух. Как вы думаете, П3 поймёт текущую постановку задачи быстрее глядя на модель, изображённую на рис. 51 или пристально смотря на вышеприведённый кусок SQL-кода?

Наш пример был игрушечным, а как изменится скорость реструктуризации схемы БД если таблиц станет 50, потом 100, 200, 1000? Какова вероятность внесения ошибки при ручном изменении SQL-скрипта по сравнению с перерисовкой элементов модели на графической схеме? Для меня ответы на все эти вопросы не только очевидны, но и многократно подтверждены практикой.

Показанная на примере суть работы на разных уровнях абстракции не изменится, если вы работаете не с реляционной, а с объектной моделью, генерирующую соответствующую ей схему БД. Для ручного подхода в этом случае процесс внесения изменений лишь усложнится за счёт необходимости поддержки отображения классов на таблицы.

Большие данные как состояние отрасли

Термин «большие данные» (big data) был извлечён из пыльных закромов маркетологами от компьютерных технологий несколько лет назад. Попробуем взглянуть, чего в этом явлении больше: нового или хорошо забытого старого.

Прежде всего, у человека с дефрагментированными мозгами, а сия процедура необходима из-за непрерывно поступающей обрывочной информации, должен возникнуть естественный вопрос: «Большие — это, простите, сколько?»

Во время написания этой главы словом «большие» оценивались объёмы, начиная примерно с 1-10 петабайт.

Хорошо, давайте заменим «большие данные» на «петабайтные данные». Суть фразы, как явствуют законы логики, не изменилась, но несколько прояснилась, что уже неплохо для начала.

Следующий не менее естественный вопрос, который должен возникать у человека с техническим или научным образованием: «А почему, собственно, 1 петабайт, а не 1 терабайт или 1 зетабайт?» Для ответа на него нам придётся переместиться на машине времени в недалёкое прошлое.

В 1975 году конференция по базам данных, представляющая широкое сообщество специалистов, ввела в обиход проблематику очень больших баз данных (Conference on Very Large Databases). Название с точки зрения маркетинга, было менее удачным, всего лишь скупая аббревиатура VLDB.

В 1979 году на поднимающейся волне интереса к большим базам данных или, точнее, платёжеспособного спроса на прогрессирующую элементную базу, была образована компания с говорящим названием Teradata. И уже в 1984 году на рынке появился её продукт Teradata database, предназначенный для массивно параллельной обработки (MPP) больших объёмов данных. Да, вы не ослышались, продукт появился именно для того, чтобы в 1984 году обрабатывать те «большие данные». А большими, в соответствии с аппаратными возможностями ЭВМ той эпохи, считались тогда данные порядка 1 терабайта.

Примерно к началу-середине 1990-х стоимость носителей информации стала подъёмной даже для средних предприятий. Вышли на рынок массово доступные продукты для анализа данных. Маркетинговым «кейсом» для ЛПР[19] было незабвенное: «Анализ выявил, что изменение местоположения прилавков с бананами увеличит их продажу на 146%». Поезд стал набирать ход, и к 2000 году средства анализа данных были включены в СУБД ведущих поставщиков, например, Microsoft OLAP. Sybase выпускает специализированную СУБД «IQ» со входным языком SQL, использующее колоночное хранение.

Следует признать, что база данных размером в 1 терабайт и сейчас, в 2014 году, вовсе не является маленькой. Транзакционные БД даже крупных предприятий со всеми сделками, контрактами, проводками, каталогами и прочим жизненно важным хозяйством измеряются десятками и сотнями гигабайт за период в несколько лет.

Из-за того, что сегодня вы можете купить себе домой жёсткий диск объёмом в 1 терабайт по цене поездки на такси из аэропорта в центр города, такие данные не перестали быть достаточно большими. Заполнить подобную БД осмысленной информацией и потом попытаться её анализировать с приемлемым временем отклика — задача, которая, по-прежнему, под силу только профессиональным аналитикам в содружестве со специалистами СУБД.

Чтобы не быть голословным, приведу примеры. База данных перемещений всех пассажиров парижского региона, при условии, что все они имеют именные магнитные карты, за год умещается в 1 (один) терабайт. В тот же один терабайт укладываются неагрегированные данные за 3-5 лет бухгалтерий всех основных компаний по добровольному медицинскому страхованию национального уровня (десятки миллионов застрахованных, десятки финансовых операций в месяц на человека) с 30-40 уровнями аналитики (измерений). Контора по веб-маркетингу умещает в БД размером около одного терабайта данные всей активности пользователей национальной службы (порядка 10 миллионов профилей) с примерно 30

[19] ЛПР — Лицо Принимающее Решения, аналог англ. VIP — Very Important Person

измерениями за 1-2 года. База данных сбора информации с устройств национальной электрической сети (сотни тысяч устройств, пакеты приходят каждые 10 минут) за 5 лет накапливает 3-4 терабайта.

Последний пример с датчиками является хорошей иллюстрацией, откуда берутся петабайты. Дело в том, что система проектировалась в середине 1990х годов, авторам пришлось достаточно плотно поработать над оптимизацией форматов хранения. С другой стороны, собиралась и предварительно обрабатывалась только та информация, которая нужна в центре. В мозге.

На этом принципе фильтрации информации для анализа работают все выжившие в результате эволюции живые организмы. Если человеческий мозг одновременно будет воспринимать всю полноту приходящих от рецепторов сигналов, то под давлением поступающих петабайтов произойдёт мгновенный крах всей системы. Поэтому, например, читая в метро книжку, вы не замечаете деталей происходящего события или разговора в нескольких метрах. А если книжка требует напряжения мысли, то не заметите и как у вас вытащат кошелёк.

При этом человеческий организм способен настроиться на приём более детальной информации. Сосредоточиться. Напрячься. Сконцентрироваться. Уловить на слух тонкий звук, запах, лёгкое прикосновение, едва заметную игру теней. Если надо.

Маркетинг нынешних «больших данных» для создания спроса и продвижения соответствующих продуктов и услуг в массы основан на противоположном принципе: «Мы не умеем быстро менять фокус детализации. Поэтому давайте будем собирать всю поступающую информацию на случай если она вдруг понадобится когда-нибудь...»

То есть, вместо гибкого механизма управления с фокусировкой на важных событиях по-прежнему предлагается склад информации, теперь не только «посмертной», но и всякой-разной. Чтобы на её основе искать закономерности, экстраполяции и делать выводы на будущее. Конечно, и человек способен анализировать своё поведение в прошлом, делая выводы. Но уже после того, как он научился оперативно реагировать на изменения

среды. И отпечатываются в памяти для анализа действительно важные события.

«Нагенерировать» данные — не проблема. Проблема «нагенерировать» осмысленные данные. Броуновское движение в капле воды «нагенерирует» и петабайты и экзабайты всего за несколько минут если выбрать соответствующую дискретность. Потом из полученных «больших данных» можно будет выводить астрологические законы природы. Описанная на молекулярном уровне структура кофейной гущи на блюдце также займёт петабайты.

Я намеренно не касаюсь специализированных БД, типа тех, что имеются у Google. Это как раз тот случай, когда реляционные подходы универсальных СУБД плохо подходят к моделируемым процессам. Специализированные решения были и будут всегда, просто потому, что они по некоторым параметрам, являющимся в данном конкретном случае ключевыми, эффективнее универсальных.

С другой стороны, теоретики и практики СУБД ещё с 1990-х ворчат на тему «засилья» реляционных БД. Несмотря на известные проблемы, РСУБД являются универсальными и решают свои задачи хранения и обработки данных с приемлемым качеством в большинстве случаев. Такая ситуация привела к некоторому «застою», когда тройка поставщиков СУБД имеет, во всех смыслах слова, более 90% всего рынка. Теоретикам и практикам оказалось не под силу вписаться в эту ситуацию и реализовать свои решения в сколько-нибудь массовом продукте. Все ограничивается нишевыми специализированными решениями.

Для таких специалистов «большие данные» открывают новые сцены и рынки. Перекрестим уже упоминавшийся выше принцип транзакционности ACID в BASE (Basically Available, Soft-state, Eventual consistency). Не будем упоминать, что распределенные БД массово существовали в 1980-90-х годах. По ключевым словам «репликация», «сервер репликации», «двухфазная фиксация» желающие смогут найти немало интересного. И что от распределённых архитектур всеми силами избавлялись при первой же возможности по причине гораздо более высоких расходов на разработку и

эксплуатацию по сравнению с централизованными решениями. Сформулируем принцип CAP: есть три качества распределенных систем — Consistency, Availability и Partition Tolerance, выбирайте любые два. Назовём этот принцип теоремой, которую, правда, никто всерьёз не будет доказывать. Зато он очень похож на другую управленческую «теорему» в софтостроении: «Быстро, качественно и дёшево — выберите два нужных критерия из трёх».

Одним словом, люди всерьёз пытаются ломать прежние тенденции.

Что имеем в итоге на сегодняшний день. Позитивные моменты:

- облегчение работы с объёмами данных, считавшимися большими 30 лет назад. Правда, в основном за счёт развития аппаратуры и универсальных РСУБД;
- некоторые подвижки в теоретических работах и их реализации;
- возможность сбора и анализа большего объёма информации в научной сфере;
- относительная общедоступность решений по обработке данных, считающихся большими сегодня (Hadoop, MapReduce и др.).

Не обошлось, конечно, и без негативных:

- на практике принцип сбора данных «на случай если они вдруг понадобятся в будущем» быстро превращает хранилище в помойку и кладбище;
- в научной сфере развернулись дискуссии на тему достоверности и собственно научности подходов, потому что при широком доступе к результатам экспериментов найденные в данных закономерности выдаются за подтверждённые гипотезы без объяснений природы связи. Самая настоящая астрология въехала в науку на колёсах телеги «больших данных». Например, корреляция между потреблением сыра «Моцарелла» на душу населения и числом получивших степень доктора наук в США за 2000-2009 годы —

целых 96 %[20], и, согласно цифрам, эти явления гораздо более связаны, нежели зависимость между наличием туч на небе и дождём.

- за исключением случаев необходимости использования специализированных СУБД, пользуясь некомпетентностью в области баз данных части разработчиков и руководителей проектов, ведётся достаточно агрессивная пропаганда против универсальных РСУБД;
- освоение бюджетов проектами с неизвестной рентабельностью с одной стороны, повышение расходов на мэйнфреймы для поддержки существующих систем 40-50-летней давности — с другой.

Что можно делать в такой ситуации? Наверное, наилучшим решением будет исходить из собственных потребностей. Тогда их придётся анализировать, планировать объёмы и удивляться, что во многих случаях можно обойтись вполне обычными данными, без «больших».

Ну, а если имеются неосвоенные бюджеты, то почему бы не собрать петабайт-другой данных. Вдруг он когда-нибудь понадобится?

[20] По данным USDA (per capita consumption of mozzarella cheese) и National Science Foundation (civil engineering doctorates awarded)

Программирование с испытаниями

«Обычно арьергард прежнего авангарда является авангардом нового арьергарда». С. Е. Лец

Типы соединений в SQL на примерах

Основная операция в SQL-запросах — *соединение*, представляющее собой последовательность множественных операций *пересечения*, согласно критериям, и *объединения* выбранных кортежей (строк) в результирующее отношение (таблицу). Для чего служат различные виды соединений таблиц проще всего запомнить на практических примерах.

Для иллюстрации возможностей соединений используются две связанные таблицы: контактные лица (`contacts`) и компании (`companies`).

```
CREATE TABLE companies (
  company_id INT NOT NULL,
  company_name VARCHAR(64) NOT NULL,
  phone VARCHAR(16) NULL,
  CONSTRAINT pk_companies
    PRIMARY KEY (company_id)
)

CREATE TABLE contacts (
  contact_id INT NOT NULL,
  contact_name VARCHAR(64),
  phone VARCHAR(16) NULL,
  company_id INT NULL,
  CONSTRAINT pk_contacts
    PRIMARY KEY (contact_id),
  CONSTRAINT fk_contact_company
    FOREIGN KEY (company_id)
    REFERENCES companies(company_id)
)
```

Заполним таблицы данными.

```
INSERT INTO companies
  VALUES (1, 'Рога и копыта', null)
INSERT INTO companies
  VALUES (2, 'НИИ ЧАВО', '322-223')

INSERT INTO contacts
  VALUES (1, 'Бендер Остап Сулейманович', null, 1)
INSERT INTO contacts
```

```
  VALUES (2, 'Гарин Петр Петрович', '322-223', null)
INSERT INTO contacts
  VALUES (3, 'Привалов Александр Иванович', '322-223', 2)
```

Наши исходные заполненные таблицы будут выглядеть следующим образом.

Таблица компаний (companies)

company_id	company_name	phone
1	Рога и копыта	NULL
2	НИИ ЧАВО	322-223

Таблица контактов (contacts)

contact_id	contact_name	phone	company_id
1	Бендер Остап Сулейманович	NULL	1
2	Гарин Петр Петрович	322-223	NULL
3	Привалов Александр Иванович	322-223	2

Теперь проделаем над таблицами выборки с использованием различных типов соединений и посмотрим на результаты, чтобы понять суть проводимых операций.

Обычное **эквисоединение**, оно же **внутреннее (INNER JOIN)** соединение.

```
SELECT contact_name, company_name
  FROM contacts INNER JOIN companies
    ON contacts.company_id = companies.company_id
  ORDER BY contact_name
```

Результат

contact_name	company_name
Бендер Остап Сулейманович	Рога и копыта
Привалов Александр Иванович	НИИ ЧАВО

Внешнее соединение слева (LEFT JOIN) включает все строки таблицы или результата, находящегося слева от оператора JOIN.

```
SELECT contact_name, company_name
FROM contacts LEFT OUTER JOIN companies
       ON contacts.company_id = companies.company_id
ORDER BY contact_name
```

Результат

contact_name	company_name
Бендер Остап Сулейманович	Рога и копыта
Гарин Петр Петрович	NULL
Привалов Александр Иванович	НИИ ЧАВО

Аналогичным образом, **внешнее соединение справа (RIGHT JOIN)** включает все строки таблицы или результата, находящегося слева от оператора JOIN. В примере соединение проводим по неключевому атрибуту — номеру телефона. На то нам и дана реляционная модель, чтобы мы не задумывались о необходимости существовании физических связей, ограничиваясь связями логическими.

```
SELECT contact_name, company_name
FROM contacts RIGHT OUTER JOIN companies
       ON contacts.phone = companies.phone
ORDER BY contact_name
```

Результат

contact_name	company_name
NULL	Рога и копыта
Гарин Петр Петрович	НИИ ЧАВО
Привалов Александр Иванович	НИИ ЧАВО

Выполним для полноты эксперимента ещё и внешнее соединение слева по тому же атрибуту.

```
SELECT contact_name, company_name
FROM contacts LEFT OUTER JOIN companies
       ON contacts.phone = companies.phone
ORDER BY contact_name
```

Результат

contact_name	company_name
Бендер Остап Сулейманович	NULL
Гарин Петр Петрович	НИИ ЧАВО
Привалов Александр Иванович	НИИ ЧАВО

Полное соединение (FULL JOIN) также проводим по неключевому атрибуту — номеру телефона.

```
SELECT contact_name, company_name
```

```
FROM contacts FULL OUTER JOIN companies
       ON contacts.phone = companies.phone
ORDER BY contact_name
```

Нетрудно убедиться, что итогом полного соединения является объединение результатов, полученных внешними соединениями слева и справа.

contact_name	company_name
NULL	Рога и копыта
Бендер Остап Сулейманович	NULL
Гарин Петр Петрович	НИИ ЧАВО
Привалов Александр Иванович	НИИ ЧАВО

Перекрёстное соединение (CROSS JOIN), оно же **декартово произведение** в терминах реляционной алгебры — каждая строка левой таблицы соединяется со всеми строками правой. Если в первой таблице имеется N строк, а во второй — M строк, то в результирующей таблице вы получите ровно N·M строк, поэтому будьте осторожны при использовании перекрёстного соединения на больших таблицах.

```
SELECT contact_name, company_name
FROM contacts CROSS JOIN companies
ORDER BY contact_name
```

Результат

contact_name	company_name
Бендер Остап Сулейманович	Рога и копыта
Бендер Остап Сулейманович	НИИ ЧАВО
Гарин Петр Петрович	Рога и копыта
Гарин Петр Петрович	НИИ ЧАВО
Привалов Александр Иванович	Рога и копыта
Привалов Александр Иванович	НИИ ЧАВО

Исходники и синхронизация структур

В «Дефрагментации» [13] мимоходом было упомянуто понятие *безысходного программирования* — программирования без исходных текстов программ. Не раз приходилось наблюдать, как вполне

квалифицированные программисты при работе с СУБД «внезапно» теряют навыки и совершают ошибки, впадая в «первородный грех»:

- считают ненастоящим или неполноценным программирование на процедурном расширении SQL и оформление соответствующих скриптов;
- пишут код процедур и триггеров непосредственно в БД;
- передают БД из разработки в тестирование методом резервного копирования и восстановления всей БД;
- как следствие, по ходу работы не имеют ни всей совокупности исходных текстов, ни истории их изменения.

Приступая к разработке программ на SQL и его процедурных расширениях, необходимо с самого начала отнестись к делу так же серьёзно, как к разработке программ на любых других универсальных языках, не имеющих отношения к СУБД. Основой работы по-прежнему являются файлы с расширением «.sql», содержащие **весь** программный код проекта включая сценарии создания базы данных «с нуля». Будучи положенными в депо контроля версий (SVN, Git и другие), эти файлы аналогично программам на Java, C++ или PHP, сохранят историю всех модификаций, позволяя вернуть некоторые изменения назад или создать независимую от основной ветку для нового выпуска версии продукта.

Некоторые поставщики СУБД, например Oracle, предоставляют разработчику развитые возможности структурирования программ на процедурном расширении PL/SQL, например, объединения в пакеты, которые можно развёртывать на множестве БД. В качестве основного инструмента поставляется интегрированный SQL Developer. Начиная с версии 4 инструментарий Oracle SQL Developer поддерживает и другие СУБД через JDBC-соединения. Однако, эта поддержка очень ограничена и в основном предназначена для миграции. Возможности Microsoft Visual Studio и отчасти даже административно-ориентированного SQL Server Management Studio также позволяют разрабатывать код для СУБД, но эти инструменты специализированы для SQL Server.

Кроме продуктов поставщиков СУБД существует немало коммерческих и условно-бесплатных универсальных интегрированных сред разработки, отражающих её специфику и поддерживающих разные СУБД: EMS SQL Manager, TOAD, DreamCoder, Database.NET и другие.

Если же говорить о свободно-распространяемых СУБД с открытым кодом, то в распоряжении разработчика зачастую имеется лишь развитая графическая консоль с возможностью запуска и отладки файлов SQL-скриптов. Например, pgAdmin для PostgreSQL, FlameRobin для Firebird или MyAdmin для MySQL. Для полноценной разработки проекта из множества исходных файлов их возможностей не хватает, но использование командных скриптов уровня операционной системы (shell, cmd, PowerShell и т. п.) позволяет решить многие проблемы.

В любом случае, следует заранее подумать об организации разработки программ слоя хранения данных. Минимально, разработчик развёртывает свои приложения в трёх средах:

- среда разработки и модульного тестирования;
- среда функционального тестирования и приёмки;
- среда эксплуатации.

С ростом сложности системы и вовлечения в процесс персонала, таких сред становится все больше, перечисленные разделяются, добавляются новые (предварительная приёмка, нагрузочное тестирование, опытная эксплуатация и другие). Но в минимальном варианте работы профессионала должны присутствовать хотя бы три перечисленных среды. Тем более, что сейчас технологии виртуализации позволяют достаточно просто управлять организацией изолированных сред.

Развёртывание в нескольких средах приводит разработчиков к необходимости иметь установочный пакет, который «с нуля» создавал бы не только каталог БД (таблицы, виды, процедуры и т. д.), но и наполнял базу начальными данными, пригодными для разработки и тестов. Впрочем, это касается не только СУБД-программирования, но и всей разработки.

Если в вашем софтостроительном процессе используется **непрерывная интеграция** (continuous integration), то разработка на уровне СУБД должна быть полностью туда включена.

Начать планирование использования исходников СУБД-программ следует с определения структуры директорий (папок) проекта и декомпозиции на модули. Общие рекомендации:

- отдельный поддиректорий для исходников слоя СУБД (слоя хранения), в нем могут располагаться файлы проектов;
- внутри располагаются поддиректории для сценариев создания каталога БД, для загрузки начальных данных и для исходников согласно их отношению к соответствующему функционалу системы, например, по одной на модуль.

Пример организации исходных файлов.

```
|-- SQL
    |-- catalog
        |-- create_db.sql
            create_tables.sql
            init_data.sql
    |-- core
        |-- doc_flow.sql
            grouping.sql
            localization.sql
            metadata.sql
            security.sql
            unit_tests.sql
    |-- accounting
        |-- accounts.sql
            periods.sql
            transactions.sql
            unit_tests.sql
    |-- CRM
        ...
    |-- MRP
    |-- sales
    |-- warehousing
```

В директории `catalog` располагаются файлы сценариев создания БД и наполнения начальными данными. Директорий `core` содержит реализацию

функционала общего назначения. Далее, в одноимённых прикладным подсистемам директориях располагаются процедуры, триггеры, функции и модульные тесты, `accounting` — бухгалтерия, CRM — отношения с клиентами, `sales` — управление сбытом и т.д.

Предположим, в проекте не предусмотрена покупка интегрированной среды для «базоданной» разработки, но планируется использовать инструментарий, поставляемый вместе с СУБД. Например, для PostgreSQL это будет pgAdmin.

В отсутствии полноценной среды разработки возникает очевидный вопрос: «Как управлять созданием БД из этого множества файлов?» Не менее очевидный ответ: «Создать командные файлы компоновки общего скрипта и включение его в общую систему сборки». Здесь имеется важная особенность: файлы SQL должны включаться в общий сценарий только в определённом порядке, соблюдая зависимости. Например, нельзя создать триггер раньше чем таблицу, к которой он относится. Разрешать многие зависимости приходится вручную, но при логичной организации исходников эта задача не представляет трудности, восходя на уровень зависимости функциональных модулей.

Для приведённого выше примера командный файл компоновки общего скрипта может быть размещён в корневой директории и выглядеть следующим образом.

```
#!bin/sh
out_file=db_install.sql

echo "Создание скрипта развертывания БД..."
echo "/* Пример сборки скрипта развертывания БД */">$out_file
for i in "catalog/create_db.sql"\
         "catalog/create_tables.sql"\
         "catalog/init_data.sql"\
         "core/doc_flow.sql"\
         "core/localization.sql"\
         "core/security.sql"\
         "core/grouping.sql"\
         "core/metadata.sql"\
         "core/unit_tests.sql"\
         "accounting/accounts.sql"\
```

```
        "accounting/periods.sql"\
        "accounting/transactions.sql"\
        "accounting/unit_tests.sql"; do
  echo "$i"
  echo "/* $i */" >>$out_file
  cat "./$i" >>$out_file
  echo >>$out_file
done

echo "Готово!"
```

Теперь мы можем запустить скомпонованный `db_install.sql` в любой среде и получить готовую к разработке или испытаниям базу данных.

```
$psql -f db_install.sql
```

Если необходимо установить несколько копий БД под разными именами на один и тот же СУБД-сервер, то следует воспользоваться параметризацией сценария.

В сценарии `create_db.sql` создание БД будет выглядеть так

```
CREATE DATABASE :dbname
```

При запуска сценария на выполнение требуется определить значение параметра `dbname`

```
$psql --set 'dbname=bench21' -c '\i db_install.sql'
```

Параметризация сценариев также поддерживается утилитой `sqlcmd`, входящей в поставку Microsoft SQL Server. В случае отсутствия такого функционала приходится задействовать макроподстановки, делая замены непосредственно утилитой обработки текста, например, `sed`.

Следует понимать, что параметризация в идеале должна касаться только входной точки в сценарий. Все остальные SQL-файлы разрабатываются в контексте соединения и БД «по умолчанию» или текущей (активной) БД. В противном случае вы можете потерять возможность прогонки и отладки отдельных файлов в консоли.

Общие рекомендации к разработке SQL-скриптов:

- избегайте явных зависимостей от имён базы данных, схем и пространств хранения таблиц (tablespace);

- осторожно используйте схемы (schema), это аналог не пространств имён (namespace) в универсальных языках, а, скорее, домена приложения (application domain). Следует понимать, что `crm.clients` и `sales.clients` — это две физически разных таблицы. В некоторых СУБД схемы все ещё привязаны к одноимённым пользователям и служат для реализации разделения доступа, а не модульности;

- не включайте в код опции физического хранения, таких как размеры страниц, хешей, процент наполняемости страниц, кластеризация и т. д. кроме обоснованных случаев и параметров «по умолчанию», подтверждённых администратором БД или экспертом по соответствующей СУБД;

- оформляйте процедуры, функции, виды и триггеры в стиле CREATE OR REPLACE, что позволит использовать одни и те же скрипты как для создания, так и для обновления каталога БД. Если такая возможность не поддерживается СУБД на уровне языка, то создание объекта должно предшествовать его удалению при условии существования IF EXISTS … DROP PROCEDURE;

- хотя бы минимально добавляйте назначение прав (GRANT) на создаваемые объекты фиксированному списку пользователей или группе. Имена таких пользователей или групп следует определить на этапе планирования развёртывания.

Пользуясь такой логикой и техническими приёмами вы сможете бесшовно интегрировать СУБД-разработку в цикл непрерывной интеграции. Новые сложности возникают, когда требуется «накатить обновления» в тех средах, где переустановка невозможна в принципе, прежде всего в эксплуатации.

Что касается процедур, функций и триггеров, то ничего менять не нужно. Подобно приведённому выше командному файлу, создаём другой (или параметризуем существующий), выполняющий компоновку отдельных скриптов в общий за исключением тех, где создаются база данных, таблицы и начальная инициализация. Если структуры не менялись, то запустив

результирующий скрипт, назовём его `db_update.sql`, на целевой БД мы обновим все процедурные объекты.

А если схема БД изменилась?

Тогда требуется аккуратно синхронизировать структуры, и, возможно, данные. Для решения этой задачи существует несколько подходов.

Ручное отслеживание изменений

Как обычно вносятся изменения в БД разработки? Если БД централизована, то этим занят администратор БД или выполняющий его роль. Тогда задача упрощается: вместо прямых изменений через графический интерфейс среды администрирования администратор БД пишет небольшой скрипт ALTER TABLE, выполняющий изменения, затем накатывает его на БД. Сам же файл изменений добавляется, например, в специальный директорий updates под именем, отражающую суть и содержащую информацию о дате обновления.

Теперь при компоновке `db_update.sql` следует включить в его начало все скрипты обновлений, следуя их хронологическому порядку — от самых ранних до недавних.

Если требуется дополнительная обработка данных до или сразу после изменения структуры, она также включается в SQL-файлы обновлений. Если же данные нужно изменить после обновления процедур, то аналогичным образом создаются SQL-скрипты обновлений в другом выделенном директории, либо с каким-нибудь признаком в имени файла, например `after_2014-01-25.sql`. Эти файлы должны быть включены в `db_update.sql` **после** всех файлов процедурных объектов.

Несколько сложнее организовать процесс при децентрализованной разработке, когда программист работает с локальной БД у себя на компьютере. Обычно, все изменения вносятся в графическом интерфейсе доступа к БД, «повозившись мышкой». Действительно, так может быть быстрее и удобнее. Но для отслеживания изменений приходится вводить аналогичную процедуру каждому разработчику: написание обновляющих

БД скриптов и размещение их в депо контроля исходников в заданном месте и с определёнными правилами именования.

Пример организации файлов обновлений.

```
|-- SQL
    |-- update
        |-- 2011-01-25.sql
            2011-01-26.1.sql
            2011-01-26.2.sql
            2011-01-26.after.sql
            2011-01-27.after.sql
```

Преимущество подхода — высокая надёжность, потому что сценарии внесения изменений тестируются уже на стадии разработки. Недостаток — зависимость от человеческого фактора (Вася забыл написать скрипт обновлений, но уже выгрузил исходники в депо).

Синхронизация версий

В этом варианте текущие изменения вносятся программистами или администратором любым удобным для них способом, в актуальном состоянии поддерживается лишь сценарий установки «с нуля».

В момент решения о выпуске версии компонуются два сценария `db_install.sql`: первый соответствует актуальной версии, второй — предшествующей (речь о версиях продукта, а не номере ревизии в депо исходников). Оба сценария создают свои БД. Затем проводится их сравнение с помощью одной из доступных на рынке утилит синхронизации БД. Я не буду приводить список, их действительно много, зачастую они специфичны для СУБД, поэтому поставщики обычно публикуют на сайте списки продуктов третьих фирм, решающих данную задачу. В конце концов, гугл по ключевым словам «database schema synchronization» выдаст множество ссылок. Следует обратить внимание, чтобы функционал утилиты синхронизации обеспечивал:

- выдачу результата изменений структур в виде SQL-сценария, производящего все нужные операции по обновлению схемы;
- определение разницы в самих данных и генерацию скрипта обновления (дополнения) данных.

Полученные SQL-сценарии можно включать в пакет обновления текущей версии до новой.

Преимущество подхода — децентрализованная разработка, отсутствие жёстких правил. Недостатки — дополнительный сложный этап интеграции, результат которого можно будет всесторонне проверить только на реальных тестовых базах; определение «дельты» для данных — нетривиальная задача не имеющая общего решения без дополнительных описаний концептуального уровня.

Использование метаданных

В этом подходе, тесно связанном с модель-ориентированной разработкой, в системе существует описание схемы данных, например в виде XML внутреннего формата. Сценарии создания БД генерируются по этим описаниям. Сценарии обновления могут быть сгенерированы в каждый момент на основании разницы описаний новой ревизии и текущей и/или на основании метаданных уровня СУБД (описания уровня таблиц, столбцов и т. д.).

При наличии скриптов загрузки данных, их обновления по-прежнему нужно программировать вручную, как в первом способе. Для автоматизации, вводятся метаописания загружаемых данных, например, в виде XML, позволяющие сравнить их состав.

Обновление проводится собственной утилитой, анализирующей метаописания и генерирующая специфичный для данной СУБД SQL, обновляющий схему и данные.

Преимущества подхода — изменения метаданных автоматически обновляют БД любой среды, начиная с БД разработчика, до текущей версии с **любой** предшествующей; возможность кросс-реализации сразу для нескольких целевых СУБД. Недостаток — сложные специфичные для продукта алгоритмы сравнений и генерации обновлений.

Некоторые особенности программирования

Параметризация запросов и SQL-инъекции

При работе с реляционной СУБД из клиентского приложения программисту приходится манипулировать классами и компонентами слоя доступа к данным (Data Access Layer), позволяющими внедрять SQL непосредственно в код вызовов. Упомянутые выше ОРП также работают поверх уровня доступа к данным, но не всегда логически отделены от него. Часто используемые компоненты и библиотеки разной степени абстракции: ADO, ADO.Net, JDBC, многочисленные xxxDAC (Data Access Components) для Delphi/Lazarus/FreePascal, ODBC, SQLAPI++, Qt Database Access для C/C++ и многие другие.

Пример простого запроса в Delphi с использованием компонентов UniDAC выглядит следующим образом.

```
with TUniQuery.Create(nil);
try
  Connection := UniConnection1;
  SQL.Text := 'SELECT * FROM clients WHERE name LIKE
''Иван%''';
  Open;
  while not EOF do
  begin
    // обработка текущей записи
    Next;
  end;
finally
  Free;
end;
```

Что делать, если поиск по фрагменту фамилии нужно сделать параметрическим? Нередко, программист пишет подобный приведённому код динамического формирования запроса.

```
SQL.Text := 'SELECT * FROM clients WHERE name LIKE ''' +
  SearchPattern + '''';
```

Такой подход к программированию запросов с параметрами хоть и может показаться простым и удобным, имеет ряд серьёзных недостатков и проблем.

1. Необходимо соблюдать формат данных для преобразования значений в строку. Например, использование конвертации `Date1.ToString` или `FloatToStr(Value2)` не является безопасным и зависит от региональных установок пользовательского компьютера и сервера.
2. Тело запроса постоянно меняется, что может оказать негативное влияние на производительность за счёт перекомпиляции и вероятного пропуска процедурного кэша на уровне СУБД.
3. Для строк необходимо учитывать внутренние кавычки и апострофы. Хотя стандарт SQL использует апострофы (одиночные кавычки) для строковых констант, некоторые СУБД для обеспечения совместимости принимают и двойные, что увеличивает число проверок. Внутренняя кавычка или апостроф могут по разному обозначаться в теле константы, например, двойным повторением или специальным предшествующим символом. Так, константа `'Жанна д'Арк'` должна быть в большинстве случаев преобразована в `'Жанна д''Арк'`, иначе СУБД вернёт синтаксическую ошибку.

Последний из приведённых пунктов является основой для хакерской атаки типа SQL-инъекция (SQL injection).

Предположим, что образец текста, напрямую вводимый пользователем, используется для поиска в запросе согласно приведённому выше примеру.

```
SQL.Text := 'SELECT * FROM clients WHERE name LIKE ''' +
  EditBox.Text + '''';
```

Злоумышленник может набрать в поле ввода например, такой текст.

```
Hello';UPDATE users SET password_hash =
0x83218ac34c1834c26781fe4bde918ee4 WHERE name='admin
```

Программа преобразует запрос в следующий вид.

```
SELECT * FROM clients WHERE name LIKE 'Hello';UPDATE users
SET password_hash = 0x83218ac34c1834c26781fe4bde918ee4 WHERE
name='admin'
```

Вместо одного запроса стало два. Первый, как и прежде, выводит список клиентов по фильтру названия. Но к нему «паровозиком» прицеплен второй, обновляющий значение хеша пароля администратора. Теперь хакер беспрепятственно может войти в систему с правами администратора, потому что кто-то в спешке или по незнанию написал небезопасный код. Аналогичным образом, даже проще, внедрить запрос можно в веб-приложении, через параметры URL или симуляцией отправки формы.

Чтобы избежать перечисленных недостатков и проблем безопасности возьмите за правило.

> При динамическом формировании SQL в программе всегда пользуйтесь функциями или объектами, позволяющими управлять параметрами.

Возвращаясь к самому первому запросу, безопасный и не ухудшающий производительность код на Delphi/FreePascal выглядит так.

```
with TUniQuery.Create(nil);
try
  Connection := UniConnection1;
  SQL.Text := 'SELECT * FROM clients WHERE name
LIKE :ClientName';
  ParamByName('ClientName').AsString := EditBox1.Text;
  Open;
  while not EOF do
  begin
    // обработка текущей записи
    Next;
  end;
finally
  Free;
end;
```

Аналогично в C#.NET.

```
using (var conn = new SqlConnection(MyConnectionString))
{
  conn.Open();
  SqlCommand cmd = new SqlCommand(
```

```
    "SELECT * FROM clients WHERE name LIKE (@ClientName)",
conn);
  cmd.Parameters.AddWithValue("@ClientName", editBox1.Text);
  SqlDataReader reader = cmd.ExecuteReader();
  while (reader.Read())
  {
    // обработка текущей записи
  }
}
```

Практически все фреймворки и библиотеки компонентов доступа к данным поддерживают возможности по управлению параметрами запросов, обратитесь к соответствующему разделу документации для справки.

Сравнение с неопределёнными (пустыми) значениями

Неопределённые (пустые) значения в SQL обозначаются, как NULL. Некоторые СУБД для совместимости поддерживают весьма странные операции сравнения, вроде `column1 = NULL`. Однако, вас не должно это подвигать к написанию подобного кода. Согласно стандарту, да и самой логике неопределённых значений, любая операция сравнения с таким значением выдаёт «ложь». Поэтому следуйте простому правилу.

> Для нахождения неопределённых значений всегда используйте только две операции сравнения IS NULL и IS NOT NULL.

Например:

```
/* Всегда возвращает пустой результат */
SELECT *
FROM persons
WHERE middle_name <> NULL

/* Возвращает контакты с отчеством */
SELECT *
FROM persons
WHERE middle_name IS NOT NULL
```

В некоторых СУБД поведение операции сравнения с неопределённым значением можно регулировать установкой опции SET ANSI_NULL (ON|OFF).

Неопределённым становится и значение в других операциях, например при слиянии строк или в математических вычислениях. Например, следующий запрос вернёт пустые строки для всех контактов, у которых имя не определено (IS NULL).

```
SELECT 'Здравствуйте, ' || first_name
FROM persons
```

Чтобы избежать появления в результате пустых строк, добавляйте соответствующее условие фильтрации.

```
SELECT 'Здравствуйте, ' || first_name
FROM persons
WHERE first_name IS NOT NULL
```

Аналогично обстоит дело с арифметикой и другими расчётами. Запрос ниже вернёт неопределённые значения возраста для всех контактов, у которых не заполнено поле «Дата рождения».

```
/* SQL Server */
SELECT datediff(year, birth_date, getdate()) AS age
FROM persons
```

Возможно, это то, что вы и ожидали получить, но подобное ожидание всегда должно основываться на знании механизмов работы с неопределёнными значениями.

Работа со строками

Обработка строк на SQL имеет некоторые особенности, специфичные для разных СУБД.

Начнём с конкатенации. Стандарт предписывает использование двойной вертикальной черты `||` для операции соединения двух строк. Однако, например, Microsoft SQL Server не поддерживает этот способ, обязуя применять знак сложения `+`.

```
/* PostgreSQL */
SELECT 'Hello,' || 'world!'
/* SQL Server*/
SELECT 'Hello,' + 'world!'
```

Данная проблема относится к обеспечению переносимости, поэтому для её решения потребуются уже рассмотренные приёмы, предусмотренные ещё на уровне проектирования.

Следующий пункт — понятие «пустая строка». Подчеркну, что это именно пустая строка, а не пустое значение NULL, имеющее семантику неопределённости. Между «значение неизвестно» и «значение отсутствует» есть большая смысловая разница.

Константа, обозначающая пустую строку, как правило, задаётся в виде двух подряд идущих апострофов. Ожидаемая длина пустой строки — ноль. Однако, так реализовано не везде.

```
length('') /* Возвращает 0 на PostgreSQL */
length('') /* Возвращает NULL на Oracle */
```

За тонкостями следует обращаться к документации, но если вы хотите сразу написать переносимый вариант SQL-запроса, то пригодится функция COALESCE или её более громоздкий аналог через CASE.

```
SELECT *
FROM persons
WHERE coalesce(lenght(middle_name), 0) = 0
```

Через CASE:

```
SELECT *
FROM persons
WHERE
  CASE
    WHEN lenght(middle_name) IS NULL THEN 0
    ELSE lenght(middle_name)
  END = 0
```

Конкатенация с пустой строкой также может иметь разное поведение. В Oracle фактически не различаются пустые строки и неопределённые значения. Несмотря на то, что длина пустой строки для Oracle это NULL, любое неопределённое значение трактуется как пустая строка.

```
/* Возвращает NULL на PostgreSQL*/
SELECT 'Тест' || NULL
/* Возвращает 'Тест' на Oracle*/
SELECT 'Тест' || NULL FROM dual
```

Весьма интересной особенностью является сравнение строк при наличии завершающих пробелов. Исторически, строки постоянной и переменной длины хранились в БД согласно заданному размеру, недостающие символы заполнялись пробелами. Для управления поведением хранения данных, соответствующая опция определена в некоторых СУБД, она управляется через SET ANSI_PADDING со значениям ON (по умолчанию) или OFF. При значении ON завершающие пробелы не обрезаются и наоборот.

В связи с особенностями хранения, сравнение строк проводилось без учёта завершающих пробелов, перекочевав на уровень стандарта SQL-92.

```
/* Оба запроса вернут одинаковые результаты */
SELECT * FROM persons WHERE first_name = 'Иван';
SELECT * FROM persons WHERE first_name = 'Иван     ';
```

В SQL-99 было сделано уточнение, что сравнение строк с завершающими пробелами должно зависеть от текущего порядка сопоставления символов (collation) соответствующего строкового типа. Однако, эта возможность реализована не везде, а если и реализована, то не всегда полно.

Например, в PostgreSQL в текущей на данный момент версии 9 все порядки сопоставления имеют по умолчанию опцию NO PAD, альтернативная опция PAD SPACE не реализована (см. раздел документации 34.10. collations). Это означает, что запрос ниже не вернёт результата.

```
SELECT * FROM persons WHERE first_name = 'Иван     ';
```

В Firebird 2.5 для обеспечения поведения NO PAD необходима создание пользовательских типов.

```
CREATE COLLATION collation1 FOR iso8859_1 FROM en_US NO PAD;
CREATE COLLATION collation2 FOR iso8859_1 FROM en_US PAD
SPACE;
CREATE DOMAIN domain1 varchar(20) character set iso8859_1
collate collation1;
CREATE DOMAIN domain2 varchar(20) character set iso8859_1
collate collation2;

/* Запрос не находит совпадений */
SELECT *
FROM persons
```

```
WHERE CAST(first_name AS TYPE OF domain1) = 'Иван     ';

/* Запрос возвращает результат, игнорируя пробелы */
SELECT *
FROM persons
WHERE CAST(first_name AS TYPE OF domain2) = 'Иван     ';
```

Также следует помнить, что длина строк в символах и байтах может быть разной, если используются типы постоянной длины или мультибайтные Unicode-кодировки. Ниже пример для PostgreSQL при используемой по умолчанию кодировке UTF-8.

```
/* Возвращает длины 4 и 10 */
SELECT length(CAST('Test' AS char(10))),
       octet_length(CAST('Test' AS char(10)));
/* Возвращает длины 4 и 8 */
SELECT length('Тест'),
       octet_length('Тест');
```

Работа с датами

Часто возникает вопрос о том, как задать константу типа даты или пары даты-времени. Программист пробует писать в запросах нечто похожее на следующий код.

```
SELECT * FROM orders WHERE created > '15/05/2014'
```

С большой вероятностью запрос проходит на локальном сервере без ошибок, но у клиента из другой страны вдруг возникает ошибка.

Проблема кроется в региональных форматах даты и времени, используемых по умолчанию. В примере использовался привычный для европейских стран формат «день/месяц/год», тогда как в Северной Америке используется «месяц/день/год». Также может иметь значение символ разделителя, и вместо косой черты использоваться точка или тире.

Чтобы застраховаться от ошибок подобного рода необходимо использовать международные форматы даты и времени, регламентируемые стандартом ISO 8601. В этом случае для даты можно выбрать фиксированный формат «ггггммдд», а для времени «ггггммддТчч:мм:сс».

Переписанный надлежащим образом запрос будет выглядеть так.

```
SELECT * FROM orders WHERE created > '2014-05-15'
```

Некоторую сложность представляют собой проверки попадания значения в заданный интервал дат. Оператор BETWEEN определяет закрытый интервал, то есть граничные значения включаются в результат. Например, если вы хотите выбрать заказы за неделю, начиная с заданной даты, то нельзя просто добавить 7 дней и написать запрос.

```
SELECT *
FROM orders
WHERE created BETWEEN
        '2014-05-15' AND
        TIMESTAMP '2014-05-15' + INTERVAL '7 days'
```

Если тип колонки `created` включает только дату, то прибавлять нужно не 7, а 6 дней. Но если тип содержит ещё и время с точностью до секунд, то такой приём не сработает, потому что все заказы седьмого дня не попадут в выборку. Выходом в такой ситуации является прибавление 7 дней, с последующим вычитанием одной секунды.

```
SELECT *
FROM orders
WHERE created BETWEEN
       '2014-05-15' AND
       TIMESTAMP '2014-05-15' + INTERVAL '7 days' - INTERVAL
'1 seconds'
```

Заметим, что данный код является непереносимым, так как использует специфичные для PostgreSQL функции работы с датами и временим. Например, в SQL Server аналогичная функция называется `dateadd`. Ну вот, ещё одна проблема! Но решить её можно достаточно просто: не кодируйте константы в запросе, используйте параметрические запросы, упомянутые выше, а значения дат вычисляйте непосредственно в программе — принцип сравнения останется прежним.

Облегчить понимание логики сравнения периодов поможет следующий пример.

```
SELECT *
FROM (
   SELECT '2014-02-03' AS d1
   UNION
   SELECT '2014-02-03T23:59:59' AS d1
   UNION
```

```
   SELECT '2014-02-04' AS d1
   UNION
   SELECT '2014-02-04T00:00:01' AS d1
   UNION
   SELECT '2014-02-05' AS d1
   UNION
   SELECT '2014-02-06' AS d1
   UNION
   SELECT '2014-02-06T00:00:01' AS d1
   UNION
   SELECT '2014-02-07' AS d1
) AS t
WHERE t.d1 BETWEEN '2014-02-04' AND '2014-02-06'
ORDER BY 1
```

Результат

```
d1
------------------
2014-02-04
2014-02-04T00:00:01
2014-02-05
2014-02-06
```

Генерация идентификаторов записей

Использование суррогатных идентификаторов в качестве ключей приводит к необходимости генерировать неповторяющиеся значения. Физическое упорядочивание таблицы в кластер по первичному ключу добавляет рекомендацию к использованию возрастающих последовательностей таких значений. Наилучшим вариантом с точки зрения физической организации будет упорядоченное множество натуральных чисел.

Последовательности и генераторы

Большинство СУБД предоставляют программисту возможности генерации идентификаторов посредством последовательностей (sequences). Стандарт SQL-2003 явным образом вводит это понятие. Однако, например, разработчики SQL Server реализовали соответствующий функционал только в версии 2012. Ещё одна иллюстрация разницы между двумя мирами.

Пример для PostgreSQL.

```
/* Создание последовательности */
CREATE SEQUENCE seq_orders START WITH 1;
/* Получение следующего значения */
SELECT nextval('seq_orders');
/* Получение текущего значения (без приращения) */
SELECT currval('seq_orders');
```

Как лучше организовать работу с последовательностями? Существует несколько основных вариантов:

- установка значения по умолчанию;
- реализация триггера вставки;
- получение и установка значений в приложении.

Установка значения по умолчанию состоит в создании последовательности и назначении соответствующей колонке функции, возвращающей следующее значение. Колонка таким образом превращается в автоинкрементную.

```
CREATE TABLE products (
    id_product integer DEFAULT nextval('seq_products'),
    ...
);
```

Определив выражение для значения по умолчанию, можно осуществлять вставку в таблицу, не указывая значения для колонки идентификатора.

Если СУБД не поддерживает возможность указания функций типа `nextval` в качестве значению по умолчанию или же одновременно требуется иметь возможность вставки записей с явно указанными значениями, то необходимо использовать триггер события `before insert`.

```
/* Пример для Firebird */
CREATE TRIGGER orders_bi_trigger FOR orders BEFORE INSERT
AS
BEGIN
 IF (NEW.id_order IS NULL) THEN
  NEW.id_order = GEN_ID(seq_orders, 1);
END;

/* Пример для Oracle */
```

```
CREATE OR REPLACE TRIGGER orders_bi_trigger
  BEFORE INSERT
  ON orders
  FOR EACH ROW
  WHEN (new.id_name is null)
BEGIN
  SELECT seq_orders.nextval INTO :new.id_name FROM DUAL;
END orders_bi_trigger;
```

Реализация логики управления величинами в приложении может быть удобна в программировании пользовательского ввода или при необходимости знать ключ только что вставленной записи, не обращаясь дополнительно к СУБД.

Например, пользователь вводит данные в таблицы, организованные по принципу «целое-части», «заказ-позиции заказа». Добавление строки в буфер позиций заказа требует указания идентификатора заказа в полях колонки внешнего ключа. Однако, если сам заказ ещё не создан, то его идентификатор неизвестен. Приходится использовать локальные значения, например отрицательные целые числа, чтобы отличить их от идентификаторов уже существующих объектов, затем при сохранении меняя их на реальные. Подобной процедуры можно избежать, если запросить СУБД следующий номер в момент создания записи в буфере приложения.

Данный способ имеет некоторый недостаток по причине неизбежно образующихся «дыр» в последовательностях, поскольку не все буфера сохраняются, пользователь может просто отменить ввод.

Существуют модификации способов установки идентификаторами в приложении.

- Метод диапазонов. Сервер выделяет приложению диапазон номеров, которое оно использует не обращаясь за новыми значениями.
- Метод «верх-низ» (HiLo). Идентификатор разбивается на две части, старшие разряды раздаются приложениям сервером, младшие заполняются локально. Представляет собой частный случай диапазона, задаваемого неявным образом.

Кроме последовательностей, некоторые СУБД, такие как DB2, MySQL, SQL Server и Sybase, поддерживают колонки автоинкрементных целочисленных типов.

```
CREATE TABLE cities
(
 id_city  int IDENTITY(1,1) PRIMARY KEY,
 city_name nvarchar (50),
)
```

Данный метод является аналогом создания последовательности и выбором описанного выше метода установки значения по умолчанию. Разница в том, что следующее значение может быть получено только в результате вставки записи в таблицу.

```
INSERT INTO cities (city_name) VALUES ('Китеж');
SELECT @@identity; /* выводит полученное значение счетчика */
```

Большинство СУБД также позволяют работать с суррогатными ключами типа UUID (GUID). Основное преимущество данного метода в том, что значения представляют собой глобально уникальные идентификаторы, которые могут генерироваться как СУБД, так и приложением, без риска конфликтов их дублирования в распределённой БД. Однако, обеспечить монотонно возрастающую последовательность, рекомендуемую для таблиц-кластеров, в таком случае очень непросто. Если же осуществлять генерацию UUID только средствами СУБД, то имеется возможность упорядочивания. Например, функция `NEWSEQUENTIALID()` из арсенала SQL Server гарантирует монотонное увеличение всех возвращаемых значений.

Общее преимущество перечисленных методов генерации — изменение счётчика вне контекста текущей транзакции. Если бы получение нового значения входило бы в транзакцию, то каждая вставка потенциально могла заблокировать другие до момента своего завершения.

Тем не менее, если встроенные реализации последовательностей не подходят для использования в приложении или по каким-то причинам требуется соблюсти транзакционность, существуют методы самостоятельной реализации. Перечислим некоторые из способов, примеры кода приведены для SQL Server.

Таблица счётчиков

В этом способе используется одна таблица на все счётчики. Получение нового значения в транзакции может привести к блокировке всей таблицы (подробнее см. главу «Транзакции, изоляция и блокировки»).

```
CREATE TABLE counters (
  name varchar(128),
  value int,
  CONSTRAINT PK_COUNTERS PRIMARY KEY (name)
);

CREATE PROCEDURE GetNextValue(
  @counter varchar(128),
  @value int out)
AS
BEGIN
  UPDATE counters
  SET @value = value = value + 1
  WHERE counter = @counter
END
```

Если использовать переносимый ANSI SQL, то код получения нового значения изменится.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION

UPDATE counters
SET value = value + 1
WHERE counter = 'table name'

SELECT value
FROM counters
WHERE counter = 'table name'

COMMIT
```

Использование identity

Данный способ может пригодиться, если СУБД позволяет определять автоинкрементные колонки, но не поддерживает последовательности. Недостатком может являться необходимость определения новой таблицы на каждый счётчик или использовать её как глобальный генератор для всех

идентификаторов. Получение нового значения в транзакции не блокирует работу других процессов.

```
CREATE TABLE counter_myname (
  value int identity(1, 1),
  foo bit,
  CONSTRAINT PK_COUNTERS PRIMARY KEY (value)
)
GO

CREATE PROCEDURE GetNextMyNameValue(@value int out)
AS BEGIN
  SET NOCOUNT ON;
  IF @@trancount > 0
    SAVE TRANSACTION tnx_GetNextMyNameValue
  ELSE
    BEGIN TRANSACTION tnx_GetNextMyNameValue
  INSERT INTO counter_myname (foo) VALUES (0);
  SET @value = @@IDENTITY
  ROLLBACK TRANSACTION tnx_GetNextMyNameValue
END
```

Транзакции, изоляция и блокировки

Понятие транзакции является основополагающим в области баз данных, поэтому оно было введено в самом начале книги. Хотя практические примеры данной главы относятся к реляционным СУБД, транзакции не являются их исключительной прерогативой.

Транзакционные файловые системы, такие как NTFS и ext4, давно сменили прежние, и теперь FAT32 можно встретить разве что на сменных картах памяти (flash cards) и USB-ключах. В стандартизированных объектных средах, например в CORBA, определена специальная служба транзакций (Transactional Service), позволяющая проводить атомарные операции. Абстрагирующие от слоя хранения ОРП имеют свои интерфейсы работы с транзакциями, чаще всего, основанные на средствах целевой СУБД. Используя шаблон «Единица работы» (Unit of work) программист должен быть уверен: пакет изменений, отосланный веб-службе, будет принят или отменён целиком.

Любые попытки создать сколько-нибудь сложное приложение без использования транзакций обречены на провал, вызванный

рассогласованными данным, ошибками и потерями. Поэтому программистам, в отсутствии соответствующих служб, пришлось бы реализовывать транзакционные механизмы вручную.

Одна из причин широкого распространения реляционных СУБД в том, что они предоставляют стандартизованные возможности управления транзакциями. Не нужно более изобретать свой сложный велосипед, требующий квалифицированных системных программистов. Однако, и научиться пользоваться имеющимся арсеналом с требуемой эффективностью тоже непросто.

Независимо от модели данных, не поддерживающая транзакции СУБД является неполноценной и имеет серьёзные технические и функциональные ограничения по области применения.

При разработке программ на SQL и его процедурных расширениях транзакция включает в себя один и более операторы языка, оперирующих с данными. В однопользовательском режиме проблем использования механизма транзакций не возникает. Но стоит двум процессам одновременно приступить к чтению и изменению пересекающегося множества записей, как всё меняется. Могут ли пользователи видеть неподтвержденные изменения друг друга? Что происходит при попытке одновременной записи? Является ли повторно считанная запись идентичной первой?

Эти и другие вопросы имеют вполне конкретные и предсказуемые ответы, основанные на понятиях **уровней изоляции транзакций**, обеспечивающих целостность данных при их одновременной обработке множеством процессов (пользователей). Вначале мы кратко перечислим особенности уровней изоляции согласно стандарту ANSI SQL-92.

Уровни SQL-92

Незавершённое (черновое) чтение (read uncommitted)

Представляет собой минимальный уровень изоляции, гарантирует только физическую целостность при записи данных на уровне поля строки.

Процессы-читатели могут видеть и считывать изменённые данные незавершённой транзакции процесса-писателя.

Например, один пользователь изменил количество товара в заказе. При сохранении произошла ошибка и транзакция откатилась. Другой пользователь между этими событиями прочитал изменённое, но неподтвержденное значение количества и учёл его в сводке.

Подтверждённое чтение (read committed)

На этом уровне процессы-читатели уже не могут считывать изменённые данные незавершённой транзакции, но процессы-писатели способны изменять прочитанные другими транзакциями данные.

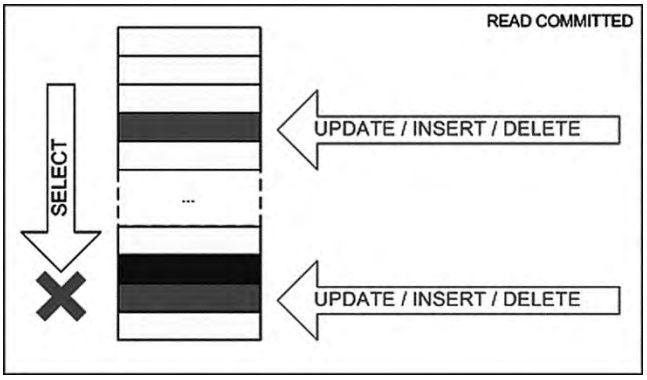

Рис.54. Процессы на уровне подтверждённого чтения

Теперь, в той же ситуации если первый пользователь изменяет количество товара в заказе, и транзакция откатывается по ошибке, другой пользователь не может в это время прочитать ещё не подтверждённое значение количества.

Возникает проблема следующего уровня. Пусть второй пользователь рассчитывает сумму заказа по его позициям `SELECT SUM()`, но при повторном выполнении того же запроса в рамках одной транзакции получает иное значение. Дело в том, что первый пользователь изменил количество по одной из позиций в промежутке между этими двумя запросами.

Повторяемое чтение (repeatable read)

Уровень гарантирует, что повторное чтение данных вернёт одни и те же значения в течении всей транзакции.

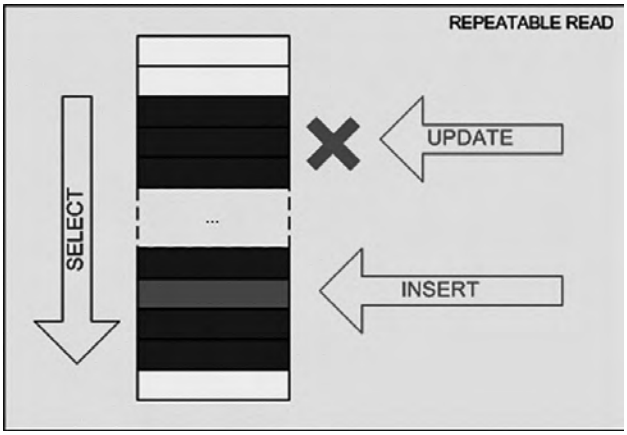

Рис.55. Процессы на уровне повторяемого чтения

Однако неизменность чтения касается только существующих данных. Процессы-писатели могут вставлять новые записи, видимые после подтверждения читателям и имеющие статус «фантома». Повторное чтение новых записей также называется «фантомным».

На уровне повторяемого чтения пользователь, рассчитывающий сумму заказа по его позициям получит одинаковые значения в одной транзакции, даже если в промежутке между запросами другой пользователь попытается изменить величины. Но если второй пользователь добавит новую позицию в заказ, то расчётная сумма опять окажется разной.

Упорядоченное чтение (serializable)

Упорядоченное чтение — максимально возможный уровень изоляции, гарантирующий полную неизменность обрабатываемых данных (прочитанных или модифицированных) до завершения транзакции со стороны других процессов.

Возвращаясь к приведённым примерам, пользователь, несколько раз считывающий сумму по позициям заказа, гарантированно получит одинаковое значение в случае любых действий других пользователей.

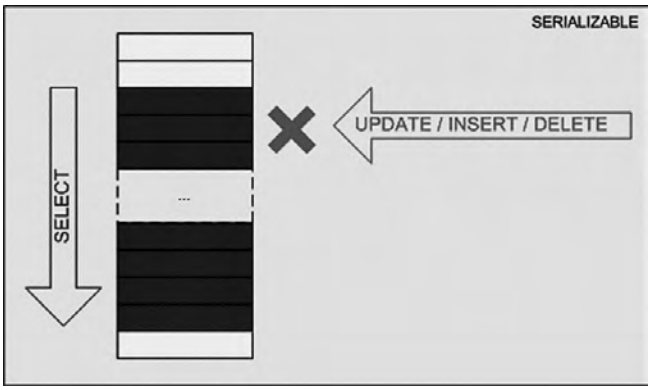

Рис.56. Процессы на уровне упорядоченного чтения

Кроме стандартных уровней некоторые СУБД вводят дополнительно и свои собственные. Например, Microsoft SQL Server, начиная с версии 2005, поддерживает уровень мгновенного снимка (snapshot), являющегося по сути не самостоятельным уровнем, а механизмом разведения процессов-читателей и процессов-писателей. Процессы-читатели не ждут завершения транзакций писателей, а считывают версию данных по состоянию на момент начала своей транзакции.

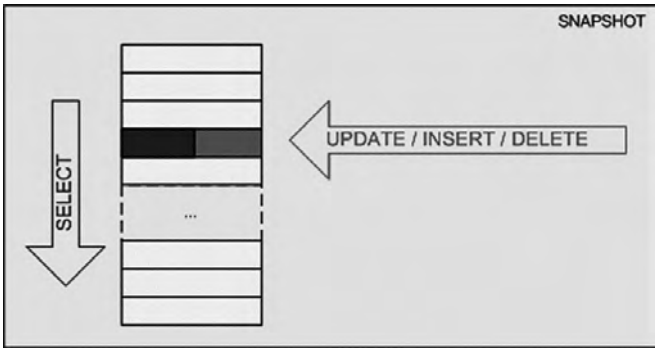

Рис.57. Процессы при использовании мгновенного снимка

Перечисленные уровни ANSI — это только спецификация, реализация которой ложится на плечи разработчиков конкретных СУБД. Остановимся на основных подходах, использующихся в современных СУБД в качестве внутреннего механизма обеспечения изоляции.

Блокировки

Блокировки пришли в СУБД из традиционного системного программирования, где семафоры и критические секции служат целям синхронизации доступа процессов к ресурсам. Действительно, если необходимо блокировать запись от чтения другими процессами, почему бы не использовать для этого штатные средства?

Однако, между блокировками системных ресурсов и записями таблиц базы данных имеются и некоторые важные отличия:

- наименьшей единицей хранения является страница, а не запись;
- количество страниц в больших таблицах велико, поэтому управление блокировками только на уровне страниц может быть неэффективным с точки зрения производительности.

Если процесс читает данные на уровне подтверждённого чтения, то блокировку можно снимать сразу после окончания чтения страницы. А если уровень — повторяемое или упорядоченное чтение? Тогда с большой вероятность придётся расставлять блокировки на все прочитанные страницы и поддерживать их до конца транзакции. При миллионах страниц для этого потребовались бы значительные системные ресурсы, прежде всего оперативная память и процессорное время.

Поэтому блокировки в СУБД имеют следующие принципиальные отличия:

- им присуща разная степень грануляции;
- обладают возможностью динамической эскалации и деэскалации грануляции;
- классифицируются по типам, имеющих разную степень совместимости между собой.

Грануляция блокировок формально начинается с уровня страницы, но некоторые СУБД поддерживают деэскалацию до уровня записи, если это возможно. Например, обновление записи со строковым полем переменной

длины может вызвать необходимость реорганизации памяти на странице, поэтому блокировка будет соответствующего уровня.

Конкретные единицы грануляции можно узнать из руководства к используемой вами СУБД, чаще всего эскалация проходит через блоки (extents) и поднимается до уровня таблицы. В некоторых случаях, например, при восстановлении данных или изменении схемы, может блокироваться база данных целиком.

Динамическая эскалация осуществляется СУБД автоматически по достижению определённого порога, определяемого статическими параметрами конфигурации и/или динамически в соответствии с наличествующими ресурсами и ценой операции.

Например, запуск подсчёта суммы по колонке на уровне повторяемого чтения неминуемо приведёт к эскалации блокировок от уровня страниц до таблицы. С другой стороны, операция выборки из маленькой таблицы с сотней строк может быть настолько быстра и дешева в терминах ресурсов СУБД, что блокировка уровня таблицы будет наложена сразу независимо от уровня изоляции транзакций.

Важную роль играют типы блокировок и их совместимость. Номенклатура типов зависит от реализации и может быть найдена в соответствующем разделе документации. Например, блокировки делятся на:

- разделяемые (совмещаемые) и монопольные (эксклюзивные);
- уже наложенные и с намерением наложения;
- чтение и модификацию.

Перечисленные признаки могут комбинироваться, например, «разделяемая блокировка чтения». Обозначим типы блокировок следующими символами.

Табл. 17. Обозначение некоторых типов блокировок

Тип блокировки	**Обозначение**
Разделяемая	S
Монопольная	X
Намерение	I
Модификация	U

Определим совместимость наиболее распространённых комбинаций блокировок (на практике типов и комбинаций может быть гораздо больше). Несовместимость на практике означает, что транзакция будет ожидать освобождения блокировки.

Табл. 18. Совместимость типов блокировок

	S	**U**	**X**	**IS**	**IU**	**IX**
S	Да	Да	Нет	Да	Да	Нет
U	Да	Нет	Нет	Да	Да	Нет
X	Нет	Нет	Нет	Нет	Нет	Нет
IS	Да	Да	Нет	Да	Да	Да
IU	Да	Нет	Нет	Да	Да	Да
IX	Нет	Нет	Нет	Да	Да	Да

Что полезного можно извлечь из таблицы совместимостей?

Как правило, СУБД предоставляет возможность мониторинга блокировок во время транзакций. Наиболее простой и доступный способ — набрать в SQL-консоли соответствующую команду или запрос, не закрывая транзакцию. В SQL Server этому служит системная хранимая процедура sp_locks, в PostgreSQL — системная таблица pg_locks.

```
BEGIN TRANSACTION;
UPDATE orders SET status = 1 WHERE id_order = 64952;
/* вывод текущих блокировок */
SELECT t.relname,   /* название таблицы */
       l.mode,      /* режим (тип) блокировки */
       l.locktype,  /* тип блокируемого объекта */
       l.page,      /* идентификатор страницы */
       l.tuple,     /* идентификатор строки */
       l.pid        /* идентификатор соединения */
FROM pg_locks l INNER JOIN pg_stat_all_tables t
       ON l.relation = t.relid
WHERE t.relname = 'orders';
COMMIT;
```

Во-первых, анализируя создаваемые запросами блокировки можно представлять, насколько разные запросы будут совместимы друг с другом и предусмотреть потенциальные конфликты. Во-вторых, если конфликты обнаруживаются во время тестов, или, что хуже, в эксплуатации, то мониторинг блокировок поможет понять их причину.

Более подробно приёмы исследования и анализа блокировок можно найти в соответствующих разделах документации конкретных СУБД.

Взаимные блокировки процессов (deadlock)

Взаимные блокировки — это тупиковые ситуации, когда две транзакции уже блокируют ресурсы, запрашиваемые ими перекрёстно.

Рассмотрим подробнее пример, изображённый на рис. 58. Пусть две транзакции одновременно обрабатывают один и тот же заказ. Транзакция №1 меняет количество товара, затем пересчитывает общую сумму и записывает её в документ. В противоположность первой, транзакция №2 вначале меняет предполагаемую дату доставки в самом документе, затем помечает товары, уже доступные на складе.

Если представить последовательность действий на ленте времени, то получится следующий порядок действий.

1. Транзакция №1 блокирует (тип блокировки — UX) строку товара N из заказа M на запись.
2. Транзакция №2 блокирует (UX) заказ M на запись.
3. Транзакция №1 ожидает блокировки (IX) заказа M на запись.
4. Транзакция №2 ожидает блокировки (IX) строки товара N из заказа M на запись.
5. Тупиковая ситуация, транзакции взаимно блокированы.

В состоянии взаимных блокировок транзакции могут «зависнуть» навечно. Поэтому все СУБД, использующие механизм блокировок, умеют распознавать тупиковые ситуации и разрешать их путём принудительной отмены одной из транзакций. При этом выбор транзакции«жертвы» может быть даже случайным. Отмена транзакции сопровождается генерацией ошибки высокого приоритета, в некоторых случаях достаточной для закрытия всего соединения.

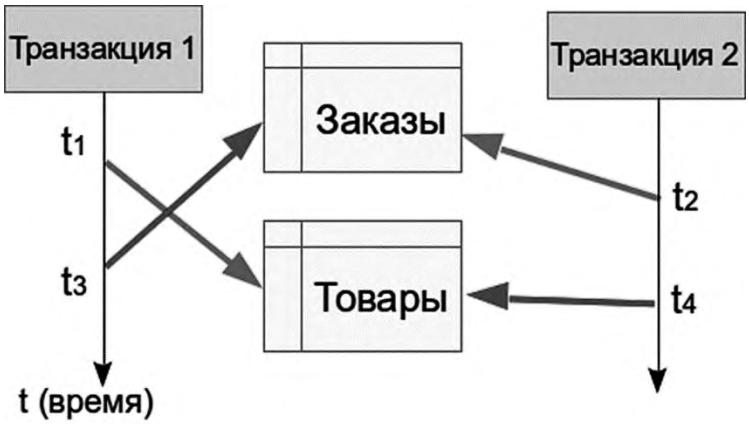

Рис.58. Взаимные блокировки

Из этого следует, что ошибка взаимной блокировки является серьёзным сбоем в работе транзакционной системы, поэтому следует придерживаться минимальных правил для предупреждения подобной ситуации:

- транзакции должны быть как можно более короткими;
- порядок блокировки ресурсов в транзакциях должен быть одинаков.

Действительно, если в приведённом примере транзакции №1 и №2 блокировали бы записи в таблицах в одном и том же порядке, то ситуации тупика не возникло бы.

Но жизнь богата ситуациями. Протестировать на возможность возникновения взаимных блокировок программный код большой системы, написанный многими авторами в разное время — задача, мягко говоря, непростая. Поэтому кроме следования правилам, необходимо предусмотреть обработку ошибки непосредственно в приложении.

Как правило, ошибка взаимной блокировки имеет специфичный для СУБД код, отловив который, требуется реализовать обработку ситуации сбоя:

- оповестить пользователя, что изменения не были зафиксированы;
- восстановить соединение с СУБД в случае его потери;
- повторить попытку.

Например, код программы на C#, обрабатывающий ситуация при работе с Microsoft SQL Server, может быть следующим.

```
int attempts = 0;
while (attempts < MaxAttempts)
{
  using (SqlConnection connection =
           new SqlConnection(connectionString))
  {
    connection.Open();
    SqlTransaction tx =
connection.BeginTransaction("Orders");
    try
    {
      // выполняем запросы в транзакции
      SqlCommand cmd = connection.CreateCommand();
      cmd.Connection = connection;
      cmd.Transaction = tx;
      cmd.CommandText = "UPDATE orders ...";
      cmd.ExecuteNonQuery();
      cmd.CommandText = "UPDATE order_items ...";
      cmd.ExecuteNonQuery();
      tx.Commit();
    }
    catch (SqlClient.SqlException ex)
    {
      if (ex.Number == 1205) // код ошибки взаимной
блокировки
        attempts++;
      else
        throw;
    }
    catch (Exception ex)
    {
      try
      {
        tx.Rollback();
      }
      catch (Exception ex2)
      {
        // обработка возможной ошибки отката транзакции
        // например, при закрытии соединения
        Console.WriteLine("Ошибка отката: {0}\n{1}",
          ex2.GetType(), ex2.Message);
      }
```

```
    }
  }
}
```

При соблюдении правил к программированию операций с БД в транзакции, ошибки взаимной блокировки достаточно редки. Средства мониторинга промышленных СУБД позволяют отслеживать ошибки разных типов, накапливая по ним статистику. Если в результате тестов или опытной эксплуатации обнаружится, что тупиковые ситуации — частый случай, необходимо произвести ревизию программного кода, являющегося источником сбоя.

Версии данных

Недостатком подхода блокировок является ожидание читающих транзакций освобождения наложенных пишущими транзакциями ограничений и возможность чтения несогласованных данных на низких уровнях изоляции. Наиболее выпукло это проявляется при программировании отчётности, непосредственно используя информацию из БД оперативного контура информационной системы.

Альтернативный подход решает проблему разведения «читателей» и «писателей» без риска возникновения конфликтов. Соответствующий механизм основан на хранении версий данных на моменты начала соответствующих транзакций.

Рассмотрим обобщённый пример работы механизма версий без привязки к конкретным СУБД.

1. Транзакция T1 стартует.
2. Транзакция T2 стартует и считывает запись N из таблицы заказов.
3. Транзакция T1 изменяет запись N. Создаётся версия записи N на момент считывания. В зависимости от объёмов изменений, создаётся так называемая обратная версия (back version), представляющая собой полную копию, либо только разностная версия.

4. T2 второй раз считывает запись N. СУБД выдаёт информацию обратной версии, не заставляя T2 ожидать подтверждения изменений T1.
5. Транзакция T3 стартует.
6. T1 подтверждает изменения и завершается. Так как в системе есть транзакции, стартовавшие до момента подтверждений изменений записи N, обратная версия сохраняется.
7. Транзакция T4 стартует и считывает запись N. СУБД выдаёт текущую версию, так как T4 стартовала после подтверждения записи T1.
8. T3 считывает запись N. СУБД выдаёт информацию обратной версии, так как T3 стартовала до подтверждения изменений T1.
9. T2 третий раз считывает запись N. СУБД также выдаёт информацию обратной версии.
10. T2 и T3 завершаются. Поскольку в системе нет активных транзакций, стартовавших до момента подтверждения изменений записи N, её обратная версия удаляется.

Как можно заметить, даже при единственной пишущей транзакции механизм реализации версий весьма непрост. Если же модификаций несколько, то СУБД приходится поддерживать не одну и несколько версий записи. Это потребует дополнительных системных ресурсов. Транзакция, производящая пакетные изменения над всеми записями таблицы, рискует к моменту завершения создать её полную копию.

По этой причине одним из недостатков версионных СУБД, по сравнению с блокировочными, является повышенная требовательность к системным ресурсам при прочих равных условиях. Например, в СУБД Oracle для реализации мгновенных снимков данных используются так называемые сегменты отката (rollback segments). Физическая архитектура расположения соответствующих файлов и администрирование сегментов, описанные на многих страницах соответствующих руководств, могут сильно повлиять на производительность системы в целом.

Проявления эффектов изоляции

Рассмотрим практические примеры поведения запросов на разных уровнях изоляции. В качестве подопытной выступит СУБД Microsoft SQL Server версии 2005 или выше.

Создадим базу данных с названием `test`, в ней — таблицы для тестовых данных и заполним их. Предположим, что мы собираем поступающую с датчиков информацию в таблицу `DevicesData`. Колонка `DeviceId` содержит идентификатор устройства, а колонка `Value` — последнее полученное значение.

```
USE test
CREATE TABLE DevicesData (
  DeviceId int not null,
  Value    int not null,
  CONSTRAINT PK_DevicesData PRIMARY KEY (DeviceId)
)
```

Для уверенного проявления эффектов таблица должна быть достаточно велика. Пусть общее количество датчиков будет равно одному миллиону, заполним их текущие значения нулями.

```
TRUNCATE TABLE DevicesData;
DECLARE @n int;
SET @n = 999;
DECLARE @List TABLE (n int);
WHILE @n >= 0 BEGIN
  INSERT INTO @List (n) VALUES (@n);
  SET @n = @n – 1;
END;
INSERT INTO DevicesData (DeviceId, Value)
SELECT A.n * 1000 + B.n, 0
FROM @List A CROSS JOIN @List B;
```

Проверки производим в SQL-консоли. Если вы привыкли работать со средой SQL Server Management Studio, просто откроем два окна для запросов к нашей базе данных `test`.

Перед каждым прогоном следует заново инициализировать таблицу устройств следующим SQL-скриптом.

```
DELETE FROM DevicesData WHERE DeviceId > 999999
UPDATE DevicesData SET value = 0
```

Подтверждённое чтение (read committed)

Установим для каждого процесса уровень изоляции READ COMMITTED. В первом окне запустим процесс-писатель, который меняет значение поля Value у двух случайным образом выбранных записей в таблице. Первое значение увеличивается на единицу, второе уменьшается на ту же единицу. Так как изначально все значения колонки Value в таблице равны нулю, значит их сумма также будет равна нулю. Следовательно, и после завершения каждой транзакции сумма значений полей Value в таблице будет оставаться неизменной и равной нулю.

Процесс №1 (писатель)

```
SET NOCOUNT ON
USE test
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
DECLARE @Id int
WHILE 1 = 1
BEGIN
  SET @Id = 500000 * rand() /* случайный ID датчика */
  BEGIN TRANSACTION
    /* изменяем первое значение */
    UPDATE DevicesData
    SET Value = Value + 1
    WHERE DeviceId = @Id
    /* изменяем второе значение */
    UPDATE DevicesData
    SET Value = Value – 1
    WHERE DeviceId = 500000 + @Id;
  COMMIT /* подтверждаем изменения */
  WAITFOR DELAY '00:00:00.100' - задержка на 100 миллисекунд
END
```

Во втором окне запустим процесс-читатель, подсчитывающий сумму значений поля Value.

Процесс №2 (читатель)

```
SET NOCOUNT ON
USE test
SET TRANSACTION ISOLATION LEVEL READ COMMITTED
SELECT SUM(Value) FROM DevicesData
```

Нетрудно убедиться, что выбранный уровень не обеспечивает логической целостности: если при запущенном процессе №1 в другом окне вручную запустить несколько раз процесс №2, то возвращаемые выборкой значения периодически будут отличаться от нуля. Если же процесс №1 прервать, то процесс №2 будет стабильно показывать нулевую сумму.

Повторяемое чтение (repeatable read) и моментальный срез

Чтобы избежать подобной проблемы, повысим уровень до повторяемого чтения, установив REPEATABLE READ. Повторив наш предыдущий эксперимент, легко убедиться в этом: процесс-читатель всегда возвращает нулевую сумму. Если же посмотреть накладываемые сервером блокировки (EXEC sp_lock), то можно увидеть многочисленные разделяемые блокировки уровня страниц (page shared lock). По сути, на запись блокируется вся таблица.

Процесс №2 (читатель)

```
USE test
SET NOCOUNT ON
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
  SELECT SUM(Value) FROM DevicesData
  EXEC sp_lock
COMMIT
```

Моментальный срез позволяет решить проблему блокировок, другим транзакции могут изменять данные. Если установить в процессах уровень SNAPSHOT и повторить эксперимент, то исчезнут многочисленные разделяемые блокировки уровня страниц, препятствующие записи данных. Таким образом, моментальный срез представляет собой очень хорошее средство для генерации отчётов или экспорта данных, поскольку гарантирует для читателя непротиворечивость и целостность данных в каждый момент времени, но при этом не препятствует работе других пишущих транзакций.

Вспомним, что повторяемое чтение гарантирует: при повторной выборке одних и тех же данных внутри транзакции мы получим одни и те же их значения. Уровень препятствует другим транзакциям менять уже прочитанные значения, однако допускается вставка новых фантомных записей.

Для воспроизведения проблемы немного изменим код.

Процесс №1 (читатель)

```
USE test
SET NOCOUNT ON
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
  /* первое чтение возвращает 0 */
  SELECT SUM(Value) FROM DevicesData WHERE DeviceId > 999000
  WAITFOR DELAY '00:00:03' /* в паузе запускаем "Процесс 2"
*/
  /* второе чтение возвращает 111 */
  SELECT SUM(Value) FROM DevicesData WHERE DeviceId > 999000
COMMIT
```

Процесс №2 (писатель)

```
USE test
SET NOCOUNT ON
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
  INSERT INTO DevicesData (DeviceId, Value)
  VALUES (1000000, 111)
  WAITFOR DELAY '00:00:00.100'
COMMIT
```

Избегая блокировок уровня таблицы, запустим читателя («Процесс 1») на небольшом диапазоне записей, например, на тысяче, а в паузе между двумя чтениями этого диапазона запустим «Процесс 2», добавляющий новую запись. В результате первая выборка вернёт 0, а вторая — 111.

Чтобы симулировать чтение фантома следует немного модифицировать программу процесса №2, предварительно удалив запись с `DeviceId = 1000000`.

```
USE test
SET NOCOUNT ON
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ
BEGIN TRANSACTION
  INSERT INTO DevicesData (DeviceId, Value)
  VALUES (1000000, 111)
  WAITFOR DELAY '00:00:03'
ROLLBACK
```

Изменив уровень до SNAPSHOT, можно увидеть, что чтение в обеих выборках корректно возвращает нуль, однако запись все равно будет добавлена конкурирующей транзакцией. Если цель — не получение среза данных для отчёта, а обеспечение логической целостности при работе нескольких процессов-писателей, то уровень моментального среза нам не поможет. Например, не обеспечивается добавление записи по ранее вычисленному условию, как в приведённом ниже случае.

Процесс №1 (писатель)

```
SET NOCOUNT ON
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
DECLARE @SumVal int
BEGIN TRANSACTION
  SELECT @SumVal = SUM(Value)
  FROM DevicesData
  WHERE DeviceId > 999000
  WAITFOR DELAY '00:00:03' /* в паузе запускаем процесс 2 */
  IF (@SumVal) = 0
    INSERT INTO DevicesData (DeviceId, Value)
    VALUES (1000000, 111)
COMMIT
```

Процесс №2 (писатель)

```
SET NOCOUNT ON
SET TRANSACTION ISOLATION LEVEL SNAPSHOT
BEGIN TRANSACTION
  INSERT INTO DevicesData (DeviceId, Value)
  VALUES (1000000, 111)
  WAITFOR DELAY '00:00:00.100'
COMMIT
```

Результат выполнения процесса №1 при запущенном в паузе процессе №2.

```
Msg 2627, Level 14, State 1, Line 18
Violation of PRIMARY KEY constraint
'PK__DevicesData__51BA1E3A'. Cannot insert duplicate key in
object 'dbo.DevicesData'.
The statement has been terminated.
```

Упорядоченное чтение (serializable) или полная изоляция

Уровень SERIALIZABLE гарантирует, что все затронутые в транзакции данные не будут изменены другими транзакциями. На этом уровне появление фантомов исключается, поэтому становятся возможными такие сложные конкурентные операции, как, например, добавление записей в диапазон значений ключа с предварительной проверкой целостности.

На практике такая обработка требуется, например, в учётных системах, когда условием добавления операции, в частности списания товара со склада или денег со счета, является предварительная проверка положительности его остатка по сумме всех предыдущих операций.

Если повторить наш предыдущий пример с уровнем SERIALIZABLE, то ошибка добавления записи возникнет уже в процессе №2.

Толстые транзакции

Если в транзакции выполняется достаточно большое число коротких и быстрых SQL-операторов или небольшое число, но длинных и долгих, она называется «толстой».

На практике в OLTP толстой транзакцией может быть даже одиночный оператор, выполняющийся несколько секунд, здесь следует оценить, какой объем данных (строк и таблиц) он при этом обрабатывает.

Оборотной стороной идеальной изоляции является установка СУБД большого числа блокировок записей, быстро переходящих в блокировки уровня таблиц. Соответственно, весьма вероятно, что толстая транзакция на время своего выполнения **полностью заблокирует работу других процессов**.

По этим причинам использовать уровень упорядоченного чтения нужно только если это действительно необходимо. В остальных случаях будет достаточно более низких уровней. По умолчанию, как правило, СУБД устанавливает уровень подтверждённого чтения. С другой стороны, необходимо стремиться к минимизации толстых транзакций в приложении, как постоянного источника проблем многопользовательской обработки.

Загрузка данных

Говоря о загрузке данных в БД, будем подразумевать, что их объем значителен. Действительно, если вы вносите в БД одиночный документ с несколькими десятками позиций, то качество программирования этой операции не будет особо влиять на время отклика. Хотя, признаюсь, приходилось встречаться с шедеврами софтостроения и на этом поле: документ-спецификация в виде XML поступал на обработку программы на C#, распихивающей его по десятку объектов с последующим сохранением через слой проекции Hibernate. Трассировка запросов обнаружила в этом процессе несколько тысяч коротких запросов к СУБД, оборачивающихся в итоге 30-секундной задержкой на каждом документе.

Ситуация становится качественно иной, когда требуется за определённое ограниченное время загрузить в БД данные достаточно большого объёма. На практике, при неумелом подходе проблемы могут начаться уже с нескольких сотен тысяч записей, хотя такой объем ни в коем случае нельзя назвать большим.

Как вы уже знаете, реляционные СУБД и SQL — это мир транзакций. Даже если вы не пишете явно в коде `begin transaction`, то все равно каждый отдельный оператор SQL представляет собой неявную одиночную транзакцию. Из этого факта вытекают общие рекомендации по загрузке данных:

- использовать средства СУБД для пакетной загрузки вне контекста транзакции;
- объединять одиночные операторы SQL в транзакцию с размером пакета от десятков единиц до сотен тысяч (определяется экспериментальным путём);
- отключать на время загрузки ограничения ссылочной целостности (кроме первичных ключей, как правило) и удалять индексы.

Почему «тормозят» одиночные транзакции?

Транзакционность в СУБД реализуется, как правило, посредством журнала и/или версий данных. То есть СУБД по запросу `INSERT INTO`

`my_table` не пишет данные прямо в физическую структуру хранения таблицы, а вначале заносит операцию в журнал. По окончанию подтверждённой транзакции данные переносятся в таблицу, запись в журнале удаляется. На самом деле, внутренний механизм гораздо более сложен и зависит от способов реализации журнала, настроек режима восстановления (recovery) и многих других факторов, поэтому каждому программисту должно быть предельно ясно: **операция вставки записи в таблицу реляционной СУБД не имеет ничего общего со вставкой аналогичной записи в файл на диске**.

Пакетная загрузка

Пакетная загрузка — наиболее быстрый способ, однако у него есть существенный недостаток: игнорируются ограничения ссылочной целостности. Поэтому если проводится загрузка сырых данных, потенциально содержащих ошибки, то следует их предварительно подготовить, либо принять риски внесения в БД противоречивой информации с необходимостью пост-обработки.

В целом, пакетная загрузка более характерна для интерактивных аналитических приложений, чем для транзакционных.

Если вы решили использовать этот подход, прежде всего следует обратиться к функционалу СУБД, к той его части, что обеспечивает пакетную загрузку данных (bulk copy). Рассмотрим загрузку вне контекста транзакции на примере возможностей SQL Server. Правильнее будет называть эти возможности *операциями с минимальным журналированием* (minimal logging), потому что даже самая простая пакетная загрузка пишет в журнал некоторую необходимую для отмены операции информацию.

Утилита командной строки `bcp` позволяет загрузить в таблицу данные из предварительно отформатированного файла в формате CSV (текстовый файл с разделителями). С помощью той же утилиты можно совершить и обратную операцию: выгрузить данные из таблицы или вида в файл на диске. Использование этой утилиты целесообразно в интерактивном режиме работы с БД или при невозможности интегрировать пакет загрузки в общий поток обработки данных иначе как через интерфейс командной строки.

Аналогичный функционал загрузки предоставляет оператор Transact SQL BULK INSERT, имеющий множество опций.

```
BULK INSERT orders
FROM 'C:\data\orders.csv'
WITH (
  FIRSTROW = 1,
  FORMATFILE='C:\data\orders.xml'
);
```

Основные рекомендации по ускорению вставки:

- если возможно по условиям задачи, очистить таблицу TRUNCATE TABLE;
- предварительное удаление всех некластерных индексов (кроме первичного ключа, если он кластерный, соответственно);
- данные в файле должны быть отсортированы по кластерному ключу (например, если это автоинкрементный целочисленный идентификатор, то в порядке возрастания номеров);
- использование опции TABLOCK.

BULK INSERT — основное «оружие» программиста на Transact SQL в вопросах загрузки из текстовых CSV-файлов. Однако, если входной формат отличается, то можно использовать другой оператор OPENROWSET с опцией пакетной загрузки.

```
INSERT INTO cities(id_city, postal_code, name)
SELECT id_city, postal_code, name
FROM OPENROWSET(
       'Microsoft.Jet.OLEDB.4.0',
       'Excel
8.0;Database=C:\data\cities.xls;HDR=Yes;IMEX=1',
       'SELECT DISTINCT
          [Num City] as id_city,
          [Postal Code] AS postal_cide,
          [Title] AS name
        FROM [Data$]'
     ) AS src
```

Однако, OPENROWSET используется в запросах, в примере — в качестве источника для INSERT. Значит он будет выполняться в контексте

транзакции этого оператора со всеми вытекающими последствиями. Чтобы минимизировать журналирование при вставке типа INSERT SELECT необходимо:

- удалить все некластерные индексы;
- отключить все ограничения целостности в таблице (внешние ключи, уникальность, проверки значений...);
- включить флаг трассировки 610;
- сортировать входные данные в соответствии с кластерным ключом таблицы;
- использовать опцию TABLOCK.

Написанный с учётом рекомендаций запрос выглядел бы следующим образом.

```
DROP INDEX IX1_cities_name ON cities;
ALTER TABLE cities DROP CONSTRAINT UC_cities_postal_code;
ALTER TABLE cities NOCHECK CONSTRAINT ALL;
DBCC TRACEON(610);

INSERT INTO cities(id_city, postal_code, name) WITH (TABLOCK)
SELECT id_city, postal_code, name
FROM OPENROWSET(
       'Microsoft.Jet.OLEDB.4.0',
       'Excel
8.0;Database=C:\data\cities.xls;HDR=Yes;IMEX=1',
       'SELECT DISTINCT
          [Num City] as id_city,
          [Postal Code] AS postal_cide,
          [Title] AS name
        FROM [Data$]'
     ) AS src
ORDER BY 1;
```

В этом примере операция вставки будет записывать в журнал только минимально необходимую информацию.

Есть и другой случай, когда OPENROWSET BULK может быть полезен: вставка в таблицу содержимого файла на диске, например, двоичного.

```
CREATE TABLE dbo.filestore (
```

```
  id      int PRIMARY KEY,
  name    nvarchar(255),
  content varbinary(max)
)
GO

INSERT INTO dbo.filestore (id, name, content)
SELECT 1 AS id, 'oembios.bin' AS name, src.BulkColumn AS
content
FROM OPENROWSET(BULK N'C:\WINDOWS\system32\oembios.bin',
SINGLE_BLOB) AS src
```

Данный приём используется только на относительно небольших файлах размером порядка единиц мегабайт. При увеличении объёмов следует использовать функционал файловых потоков (filestream), введённый в версии SQL Server 2008.

Программист вне рамок Transact SQL, использующий средства фреймворка .NET, имеет в своём распоряжении класс `System.Data.SqlClient.SqlBulkCopy`, выполняющий пакетную вставку данных в таблицу.

```
DataTable dt = new DataTable();
ReadCSVFileIntoDataTable("c:\data\orders.csv", dt);
using (SqlBulkCopy copy = new SqlBulkCopy(connection))
{
  copy.DestinationTableName = "cities";
  copy.ColumnMappings.Add(0, 0);
  copy.ColumnMappings.Add(1, 1);
  copy.ColumnMappings.Add(2, 2);
  copy.WriteToServer(dt);
}
```

Наконец, SQL Server предоставляет программисту службу интеграции SQL Server Integration Services. Это полноценная среда визуального программирования с возможностями вставки кода на Visual Basic и C#. Для простых задач типа перекачки между источником и приёмником или конвертации БД из старой структуры в новую интерфейс достаточно интуитивен и понятен даже опытному пользователю MS Access. Но с усложнением требований приходится заниматься полноценным программированием и параметризацией приложений-пакетов, что влечёт за

собой необходимость освоения среды разработки и исполнения, изучения фреймворка, окружения, основных приёмов и подводных камней.

К недостаткам Integration Services следует отнести риски по совместимости пакетов между версиями. Во-первых, каждая версия среды разработки BIDS (Business Intelligence Development Studio), входящая в поставку SQL Server, совместима только со своей средой. И менять в BIDS 2008 пакеты, сделанные в BIDS 2005, невозможно. Во-вторых, при переходе с версии 2000 на 2005 совместимость была прервана полностью: скомпилированные пакеты прежнего DTS (Data Transformation Server) все ещё могли выполняться в среде SSIS 2005, но работать с соответствующими проектами на уровне исходников в BIDS 2005 было невозможно, требовалась их переделка.

Компрессия таблиц

Важным фактором скорости вставки является компрессия таблиц. Начиная с версии 2008 SQL Server позволяет задать тип сжатия данных таблицы, построчный и постраничный. В первом случае осуществляется оптимизация размера строки согласно хранимым в ней значениям колонок. Например, для колонки с целочисленным типом в строке со значением поля «257» вместо четырёх байт для хранения будет выделено два.

Постраничная компрессия больше походит на привычное всем сжатие файлов архиватором: размер уменьшается, но увеличивается загрузка процессора и время доступа на запись за счёт переупаковки данных.

Поэтому при пакетной вставке в сжатую таблицу следует исходить из объёма загружаемых данных по отношению к уже имеющимся в таблице. Для пустой таблицы компрессию следует отключить на время загрузки. Если же объем вставки по количеству строк не превышает 10-20% от размера таблицы, то следует сохранять таблицу сжатой. Разумеется, это увеличит время загрузки по сравнению с несжатой таблицей, но полная переупаковка многократно превысит эти накладные расходы.

Для более точной оценки порога, за которым переупаковка таблицы при пакетной загрузке становится невыгодной, следует проводить тестирование на структурах и данных, максимально приближенных к эксплуатационным.

Загрузка в других СУБД

PostgreSQL поддерживает аналог INSERT BULK в виде многофункциональной команды COPY, при этом возможна как загрузка, так и выгрузка данных средствами SQL.

```
-- загрузка
COPY orders FROM '/home/usr1/data/orders .csv';
-- выгрузка
COPY orders TO '/home/usr1/data/orders2.csv' (DELIMITER '|');
-- выгрузка с одновременным сжатием
COPY orders TO PROGRAM 'gzip > /home/usr1/data/orders2.gz';
```

Не все СУБД поддерживают пакетную загрузку явным образом, но найти средства для минимизации журналирования можно в большинстве случаев. Например, для Firebird можно использовать функциональность внешних таблиц.

В файле конфигурации firebird.conf следует включить соответствующую опцию.

```
ExternalFileAccess = Full
```

Создаём внешнюю таблицу на основе текстового файла

```
CREATE TABLE BCP_DATA EXTERNAL FILE 'c:\data\data.txt' (
  ID CHAR(10) NOT NULL,
  NAME CHAR(100),
  CRLF CHAR(2)  -- CHAR(1) for Linux
);
COMMIT;
```

Выгружаем данные во внешнюю таблицу (файл data.txt должен быть пуст).

```
INSERT INTO BCP_DATA(ID, NAME, CRLF)
SELECT ID, NAME, ascii_char(13) || ascii_char(10) FROM
MY_REAL_TABLE;
```

После выполнения команды файл data.txt будет заполнен данными в формате с колонками фиксированной длины.

Вставка из внешней таблицы, тем не менее, проводится в контексте транзакции, но затрагивает только таблицу-приёмник.

```
INSERT INTO MY_REAL_TABLE(ID, NAME)
SELECT ID, NAME FROM BCP_DATA;
```

Вставка в толстой транзакции

Если пакетная вставка не может быть использована, программисту остаётся построчная передача данных от источника к приёмнику. Часто встречающаяся ошибка в этом случае — отсутствие явных транзакций.

Как уже было сказано, вставка строки в таблицу является неявной транзакцией, поэтому, например, перекачивая миллион даже из текстового файла строк в таблицу реляционной СУБД вы неявно проведёте миллион транзакций, что резко снизит производительность вашей программы.

Выходом из ситуации является объединение вставок в пакет под одной транзакцией. Размер пакета следует подбирать экспериментально, он может достигать и сотен тысяч записей. Предварительно, данные кешируются в клиентском наборе данных (DataSet).

Пример загрузки в среде Delphi/Lazarus с использованием библиотеки компонентов доступа к различным СУБД UniDAC.

```
procedure BulkLoad(Source, Target: TUniTable);

const
  BatchSize = 10000; // размер пакета в одной транзакции

  procedure CommitBatch;
  var
    Tnx: TUniTransaction;
  begin
    Tnx := TUniTransaction.Create(nil);
    try
      try
        Tnx.AddConnection(Target.Connection);
        Target.UpdateTransaction := Tnx;
        Tnx.StartTransaction;
        Target.ApplyUpdates; // вставка порции в таблицу СУБД
        Tnx.Commit;
        Target.CommitUpdates; // очистка кеша
      except
        Tnx.Rollback;
        raise;
      end;
    finally
      Target.UpdateTransaction := nil;
```

```
      FreeAndNil(Tnx);
    end;
  end;

var
  i, CurrRecNo: integer;
begin
  Source.First;
  CurrRecNo := 0;
  while not Source.EOF do
  begin
    Inc(CurrRecNo);
    Target.Append;
    for i := 0 to Source.Fields.Count - 1 do
       Target.Fields[i].Value := Source.Fields[i].Value;
    if CurrRecNo mod BatchSize = 0 then
      CommitBatch;
    Source.Next;
  end;
  if Target.UpdatesPending then
    CommitBatch;
end;
```

По сравнению с одиночными вставками записей, толстая транзакция обеспечивает даже не в разы, а на порядки (в 100-1000 раз) большее быстродействие. Оборотной стороной толстой транзакции является потенциальная блокировка работы других пользователей. Поэтому для интенсивно используемых транзакционных БД размер пакета не должен превышать 10-100 записей, обеспечивая быстрое выполнение пакета.

Все вышесказанное также касается операций обновления и удаления данных.

РСУБД и неполно структурированные данные

В главе «Неполно структурированные данные и высокая нагрузка» был описан метод проектирования, позволяющей средствами реляционной СУБД получить функционал, отсутствие которого может быть причинами выбора других решений из области NoSQL.

К концу 1990х годов XML стал фактическим стандартом для систем электронного обмена данными[21] (СОД), шлюзов между информационными системами разных уровней. Возможность работы с XML была введена основными производителями прежде всего для поддержки интеграции и обработки XML-документов средствами РСУБД. Например, SQL Server способен даже реализовать полноценную веб-службу исключительно средствами языка Transact SQL.

Современные РСУБД обладают широкими возможностями хранения и обработки неполно структурированных данных (НСД), прежде всего в виде XML. Программист получает возможность не только совмещать сильные стороны обоих подходов, но и обрабатывать НСД, используя множественный подход и соединения с физическими таблицами.

Поддержка XML

Рассмотрим на примерах возможности обработки XML, предоставляемые СУБД PostgreSQL (текущая версия 9).

Создадим две таблицы, заказов (`orders`) и продуктов (`products`), но, в отличие от обычного реляционного подхода, будем хранить часть атрибутов в виде XML. Для продуктов в качестве НСД выступает спецификация, которая может содержать разнообразную информацию, а для заказов — его переменный состав.

```
CREATE TABLE products (
  id_prod integer NOT NULL,
  code    varchar(10) NOT NULL,
  name    varchar(50) NOT NULL,
  spec    xml NOT NULL,
  CONSTRAINT pk_products PRIMARY KEY (id_prod)
);

CREATE TABLE orders
(
  id_order integer NOT NULL,
  created  date NOT NULL,
  doc_data xml NOT NULL,
  CONSTRAINT pk_orders PRIMARY KEY (id_order)
```

[21] В англоязычной среде используется аббревиатура EDI — Electronic Data Interchange

```
);
```

Таблицы получились несвязными явным образом, поскольку требуемая колонка внешнего ключа отсутствует. Первая плата за гибкость — СУБД не гарантирует целостность, заказы могут содержать ссылки на несуществующие продукты. Но при необходимости можно реализовать соответствующую поддержку триггером.

Заполним таблицы тестовыми данными. Как можно видеть, сохраняемый XML не имеет строгой схемы и потому относится к НСД.

```
/* Инициализация таблицы продуктов */
INSERT INTO products (id_prod, code, name, spec)
SELECT 1, 'P01', 'Мука пшеничная в/с',
  XMLPARSE(DOCUMENT convert_from(
  '<packs>
    <pack>
      <weight>1</weight>
      <unit>кг</unit>
    </pack>
    <pack>
      <weight>50</weight>
      <unit>кг</unit>
    </pack>
   </packs>', 'utf-8')::xml)
UNION ALL
SELECT 2, 'P02', 'Дрожжи',
  XMLPARSE(DOCUMENT convert_from(
  '<packs>
    <pack>
      <weight>100</weight>
      <unit>г</unit>
    </pack>
    <expired>10</expired>
   </packs>', 'utf-8')::xml)
UNION ALL
SELECT 3, 'P03', 'Сахар',
  XMLPARSE(DOCUMENT convert_from(
  '<packs>
    <pack>
      <weight>10</weight>
      <unit>кг</unit>
    </pack>
    <color>белый</color>
   </packs>', 'utf-8')::xml);
```

```
/* Инициализация таблицы заказов */
INSERT INTO orders (id_order, created, doc_data)
SELECT 1, date '2014-02-20',
  XMLPARSE(DOCUMENT convert_from(
  '<items>
     <product>
       <id>1</id>
       <quantity>50</quantity>
       <units>кг</units>
       <price>45</price>
     </product>
     <product>
       <id>2</id>
       <quantity>300</quantity>
       <units>г</units>
       <price>80</price>
     </product>
    </items>', 'utf-8')::xml)
UNION ALL
SELECT 2, date '2014-02-21',
  XMLPARSE(DOCUMENT convert_from(
  '<items>
     <product>
       <id>3</id>
       <quantity>100</quantity>
       <units>кг</units>
       <price>70</price>
     </product>
    </items>', 'utf-8')::xml)
UNION ALL
SELECT 3, date '2014-02-22',
  XMLPARSE(DOCUMENT convert_from(
  '<items>
     <product>
       <id>1</id>
       <quantity>80</quantity>
       <units>кг</units>
       <price>50</price>
     </product>
    </items>', 'utf-8')::xml);
```

Рассмотрим подробнее, какие возможности даёт использование XML в запросах. PostgreSQL использует стандартизованный язык XPath, уже упоминавшийся в главе «Иерархическая модель». Выражение XPath в

PostgreSQL возвращает значение в виде одномерного массива из XML-элементов.

```
SELECT name,
       xpath('//pack/weight/text()', spec) AS pack_weight
FROM products
```

Результат:

```
name                    pack_weight
character varying(50)   xml[]
---------------------------------
Мука пшеничная в/с      {1,50}
Дрожжи                  {100}
Сахар                   {10}
```

Для преобразования одномерного массива в реляционный вид используется встроенная функция unnest. Например следующий оператор вернёт таблицу из одной колонки со значениями массива. В терминах линейной алгебры, проводится операция транспонирования вектора-строки в столбец.

```
SELECT unnest(ARRAY['Синий', 'Красный', 'Зелёный']) AS color
```

Результат:

```
color
text
-----------
Синий
Красный
Зелёный
```

Другая полезная функция-предикат — xpath_exists, возвращающая истину в случае нахождения заданного пути в XML-документе или ложь в противоположном случае. Поскольку документы не ограничены схемой, использование проверки наличия соответствующих элементов будет частой необходимостью.

```
SELECT name, xpath_exists('//color', spec)
FROM products
```

Результат:

```
name                    xpath_exists
character varying(50)   boolean
---------------------------------
```

```
Мука пшеничная в/с      f
Дрожжи                  f
Сахар                   t
```

Теперь попробуем применить на практике полученное представление о работе XML-функций.

Запрос №1. Вывести все продукты, у которых специфицирован максимальный срок реализации.

```
SELECT *
FROM products
WHERE xpath_exists('//expired', spec)
```

Запрос №2. То же, но с выводом числового значения срока (в сутках). Обратите внимание, что требуются несколько операций: конвертация значений XML-массива в текстовый и целочисленный типы, преобразование в скаляр.

```
SELECT id_prod,
  unnest(xpath('//expired[1]/text()', spec)::text[]::int[])
FROM products
WHERE xpath_exists('//expired', spec)
```

Ещё одна плата за гибкость: из-за отсутствия схемы данных, мы не можем гарантировать что значение срока истечения годности в спецификации будет единственным. Поэтому средствами XPath явно выбираем только первый подходящий элемент.

Другой вариант — выбирать первый элемент массива уже от полученного результата. При большом размере массива он может быть менее производительным, чем отсечение элементов на уровне XPath-запроса.

```
SELECT id_prod,
       (xpath('//expired/text()', spec))[1]::text::int
FROM products
WHERE xpath_exists('//expired', spec)
```

Запрос №3. Выбрать все заказы, включающие продукты, у которых в спецификации не указаны сведения о сроке истечения годности. Здесь запрос немного посложнее и состоит из двух выборок, соединяемых по ключу.

```
SELECT o.id_order, p.id_prod, p.name
FROM
```

```
  (SELECT id_prod, name
   FROM products
   WHERE NOT xpath_exists('//expired', spec)
  ) p
  INNER JOIN
  (SELECT id_order,
     unnest(xpath('//items/product/id/text()',
                  doc_data)::text[]::int[]) AS id_prod
   FROM orders
  ) o
  ON p.id_prod = o.id_prod
```

Другой вариант использует вложенный подзапрос.

```
SELECT o.id_order, p.id_prod, p.name
FROM orders o
     INNER JOIN products p
     ON p.id_prod = ANY(
       xpath('//items/product/id/text()',
             o.doc_data)::text[]::int[])
WHERE NOT xpath_exists('//expired', p.spec)
```

Как мы помним, соответствие значения внешнего ключа в XML-документе и ключа продукта не гарантировано, несуществующие значения будут просто пропущены. Для их отображения надо использовать внешнее соединение и проводить корректировку рассогласованных данных.

Запрос №4. Рассчитать общую сумму покупки по каждому заказу.

```
SELECT id_order, sum(price * qty)
FROM
  (SELECT id_order,
    unnest(xpath('//items/product/price/text()',
                 doc_data)::text[]::int[]) AS price,
    unnest(xpath('//items/product/quantity/text()',
                 doc_data)::text[]::int[]) AS qty
  FROM orders) o
GROUP BY id_order
```

Очередная плата за гибкость: в отсутствии схемы остаётся лишь надеяться, что значение элемента <price> может быть преобразовано в целочисленное. В противном случае выполнение запроса прервётся по ошибке.

Индексация поиска

За рамками примеров остался вопрос индексации НСД. Пока таблица содержит небольшое количество документов, выполнение запросов, осуществляющих последовательное сканирование таблицы и проверку каждого документа.

Таблица заказов из примера содержит всего три строки. Если заполнить её данными, объёмом в несколько сотен тысяч строк, то время отклика при выполнении тех же запросов будет исчисляться уже не миллисекундами, а единицами и десятками секунд.

В рамках примера ограничимся программированием небольшого SQL-скрипта, заполняющего таблицы продуктов и заказов. В то же время пример заполнения таблицы тестовыми данными одним единственным оператором SELECT показывает, насколько язык развит и высок по уровню. Напишите тот же код на Яве или C# и сравните по числу операторов и строк.

```
TRUNCATE TABLE products;
INSERT INTO products (id_prod, code, name, spec)
SELECT id,
       'P' || (id)::int AS code,
       md5(random()::text) AS name,
       xmlparse(document (
         '<packs><pack><weight>' ||
         trunc(random() * 100)::text ||
         '</weight></pack></packs>')::xml)
FROM (SELECT *
      FROM generate_series(1, 1000) AS id
     ) AS seq;


TRUNCATE TABLE orders;
INSERT INTO orders (id_order, created, doc_data)
SELECT id,
       date '2014-01-01' + trunc(random() * 100)::int,
       xmlparse(document (
       '<items>' ||
       '<product><id>' || trunc(random() * 500 + 1)::text ||
       '</id>' ||
       '<quantity>' || trunc(random() * 700)::text ||
       '</quantity>' ||
```

```
       '<price>' || trunc(random() * 100)::text || '</price>'
||
       '</product>' ||
       '<product><id>' || trunc(random() * 500 + 501)::text
||
       '</id>' ||
       '<quantity>' || trunc(random() * 300)::text ||
       '</quantity>' ||
       '<price>' || trunc(random() * 200)::text || '</price>'
||
       '</product>' ||
       '</items>')::xml)
FROM (SELECT *
      FROM generate_series(1, 100000) AS id
     ) AS seq;
```

Если выполнить на заполненной базе относительно простой запрос, то можно констатировать увеличение времени отклика до нескольких секунд.

```
SELECT *
FROM orders
WHERE (xpath('//items/product/id/text()',
             doc_data))[1]::text::int = 123
```

Добавив перед запросом команду EXPLAIN, можно увидеть, что план запроса состоит из последовательного сканирования таблицы `orders` и фильтрации по XPath-выражению, выполняемую над документом.

```
Seq Scan on orders  (cost=0.00..4500.00 rows=500 width=40)
  Filter: ((((xpath('//items/product/id/text()'::text,
doc_data, '{}'::text[]))[1])::text)::integer = 123)
```

В общем случае, ситуацию можно поправить построением XPath-индексов по наиболее часто используемым выражениям. Для приведённого XPath-запроса индекс будет аналогом индекса по внешнему ключу.

```
CREATE INDEX ix1_orders_id_prod_xml
ON orders USING btree (
   ((xpath('//items/product/id/text()',
           doc_data))[1]::text::int)
)
```

Перезапустив запрос можно увидеть, что время выполнения сократилось на порядки. В моём частном случае величины равнялись 1868 и 72 миллисекунды, соответственно, что составляет оптимизацию по времени в 26 раз. Абсолютные значения будут зависеть от мощности

компьютера, но относительная разница в 20-30 раз воспроизводится независимо от аппаратной конфигурации.

План выполнения подтверждает использование созданного индекса.

```
Bitmap Heap Scan on orders  (cost=12.14..1259.81 rows=500
width=40)
  Recheck Cond: ((((xpath('//items/product/id/text()'::text,
doc_data, '{}'::text[]))[1])::text)::integer = 123)
  ->  Bitmap Index Scan on ix1_orders_id_prod_xml
(cost=0.00..12.02 rows=500 width=0)
        Index Cond:
((((xpath('//items/product/id/text()'::text, doc_data,
'{}'::text[]))[1])::text)::integer = 123)
```

Решена ли проблема?

Не решена, но локализована. Использовать данный индекс для других типов выражений невозможно, тогда как введение колонки, хранящей НСД в виде XML, было сделано в соответствии с пожеланием хранить любую информацию с заранее неизвестными атрибутами. Если добавится новая информация, перед программистом опять встанет дилемма: создавать новый индекс или оставить пользователя в ожидании перед монитором.

Следует понимать, что выбор между индексаций, вызывающей увеличение размера БД и замедление её модификаций, не зависит от модели данных, используемой в основе СУБД. Алгоритмы и способы быстрого поиска — это проблематика теоретического программирования, а не результат «магических действий» администратора БД. Более подробно о построении индексов см. главу «Производительность SQL-запросов».

Поддержка JSON

Начиная с версии 9.3, СУБД PostgreSQL обладает встроенным типом `json`, позволяющем хранить НСД в ранее упомянутом формате JSON.

Как и в случае XML, можно создать колонку соответствующего типа НСД.

```
CREATE TABLE products (
  id_prod integer NOT NULL,
  code    varchar(10) NOT NULL,
  name    varchar(50) NOT NULL,
  spec    json NOT NULL,
```

```
  CONSTRAINT pk_products PRIMARY KEY (id_prod)
);
```

Заполним таблицу данными, аналогичным взятым из примера для XML.

```
INSERT INTO products (id_prod, code, name, spec)
SELECT 1, 'P01', 'Мука пшеничная в/с',
  '{"packs": [
      {"weight": 1, "unit": "кг"},
      {"weight": 50, "unit": "кг"}]
   }'::json
UNION ALL
SELECT 2, 'P02', 'Дрожжи',
  '{"packs": [{"weight": 100, "unit": "г"}],
    "expired": 10
   }'::json
UNION ALL
SELECT 3, 'P03', 'Сахар',
  '{"packs": [{"weight": 10, "unit": "кг"}],
    "color": "белый"
   }'::json
```

В отличие от XPath, основными приёмами работы по извлечению данных будут определённые для JSON-значений операторы.

Табл. 19. Основные операторы для работы с JSON

Оператор	**Тип и назначение операнда правой части**	**Что делает?**	**Пример и результат**

->	Целое (номер элемента)	Возвращает элемент массива по номеру	`SELECT '[1,2,3]'::json->2 AS value`
			`value`
			`json`
			`------`
			`3`

->	Текст (имя элемента)	Возвращает значение элемента по имени	`SELECT '{"a":1,"b":2}'::json->'b' AS value`

Оператор	Тип и назначение операнда правой части	Что делает?	Пример и результат
			value json ------ 2
->>	Целое (номер элемента)	Возвращает элемент массива по номеру в виде текста	SELECT '[1,2,3]'::json->>2 AS value
			value text ------ 3
->>	Текст (имя элемента)	Возвращает значение элемента по имени в виде текста	SELECT '{"a":1,"b":2}'::json->>'b' AS value
			value text ------ 2
#>	Массив строк (путь)	Возвращает элемент по заданному пути	SELECT '{"a":[1,2,3],"b":[4,5,6]}' ::json#>'{a,2}' AS value
			value json ------ 3
#>>	Массив строк (путь)	Возвращает элемент по заданному пути в виде текста	SELECT '{"a":[1,2,3],"b":[4,5,6]}' ::json#>>'{a,2}' AS value
			value text ------ 3

Часто использующаяся в запросах функция json_array_elements возвращает элементы массива, транспонированные в виде колонки.

```
SELECT json_array_elements(
  '[{"weight": 100}, {"weight": 50}]') AS value
```

Результат:

```
value
json
---------------
{"weight": 100}
{"weight": 50}
```

Запрос №1. Вывести все продукты, у которых специфицирован максимальный срок реализации.

```
SELECT *
FROM products p
WHERE (spec#>'{expired}') IS NOT NULL
```

Запрос №2. То же, но с выводом числового значения срока.

```
SELECT *, (spec#>>'{expired}')::int
FROM products p
WHERE (spec#>'{expired}') IS NOT NULL
```

Запрос №3. Вывести все спецификации упаковок товара в колонки «Вес» и «Единицы измерения».

```
SELECT *,
  (json_array_elements(spec#>'{packs}')#>>'{weight}'
    )::int AS "Вес",
  json_array_elements(spec#>'{packs}')#>>'{unit}'
    AS "Единица изм."
FROM products p
```

Для часто используемых путей извлечения из документа значений можно строить индексы.

```
CREATE INDEX ON products((spec#>>'{expired}'));
```

Выводы

Современные реляционные СУБД представляют достаточно широкие возможности по работе с неполно структурированными данными (НСД), прежде всего с XML. В меньшей степени поддерживается JSON.

Возможность индексации НСД позволяет оптимизировать время поиска в БД относительно большого объёма. При этом проблемы достижения оптимума между производительностью поиска, операциями модификации и размером БД являются общей проблемой независимо от используемой модели данных.

Следует понимать, что работа со встроенными типами XML или JSON даже средствами РСУБД не делает этот подход реляционным. По прежнему, программист оперирует понятиями соответствующей модели НСД.

Использование НСД в рамках РСУБД позволяет комбинировать подходы, опираясь на сильные стороны каждого из них.

Для использования НСД в условиях, приближенных к высокой нагрузке, необходимо выносить в основную таблицу все значимые атрибуты, по которым ведётся поиск.

Постраничные выборки

Тема постраничных выборок с громким плеском, пеной и пузырями всплыла в связи с ростом разработки веб-приложений. Интерфейс пользователя в веб несколько отличается от традиционного оконного, прежде всего техникой постраничного вывода длинных списков вместо их прокрутки. Поисковики в Интернет выдают сотни тысяч результатов с темами, подобной «как вывести записи с 100 по 150-ю». При этом мало кто задумывается о практической целесообразности вывода на экран результата выборки с возможностью постраничного просмотра, если она содержит многие сотни и тысячи записей.

Добавлю несколько рекомендаций веб-программистам.

- Ограничивайте размер выборки, за исключением специально оговорённых случаев. Нет смысла возвращать все 10500 строк, если пользователь ищет в БД населения города по фамилии Смирнов. Не стоит огорчать пользователя и нагружать СУБД бесполезными запросами с пролистыванием. Покажите вопрошающему первые несколько десятков записей и попросите уточнить запрос ввиду большого количества результатов. Если сгруппировать результаты в

иерархию, например, по именам или годам рождения с соответствующим интерфейсом, то можно выводить и большее число записей. Вспоминайте, как часто вы сами ходите дальше первой страницы поиска в Яндексе или Google.

- Если запрос все-таки содержит достаточное количество строк, чтобы показывать результирующую таблицу с функцией пролистывания, то кэшируйте результат. Объект «Набор данных» уровня сессии для корпоративного приложения с десятками-сотнями соединений пользователей будет работать быстрее, чем повторные запросы к СУБД с целью выкачать очередную страницу из 20 записей. В веб-приложениях число соединений может быть велико, но тем ценнее производительность кэширования, только механизмы усложняются.

Как писал классик, «величайшей ошибкой было бы думать», что постраничные выборки появились именно в веб-эпоху. Наиболее востребованная для них задача — передача данных пачками по N записей, где N может быть достаточно велико. В этом контексте мы и рассмотрим соответствующие средства, предоставляемые СУБД. Но все сказанное для веб-приложений тоже пригодится.

Средства постраничной выборки можно разделить на следующие группы:

- ограничения, выполняемые на уровне ядра СУБД. Это конструкции, являющиеся расширением синтаксиса SQL-запросов, воспринимаются СУБД однозначно и соответствующим образом учитываются на стадии оптимизации. Это наиболее быстрый способ в большинстве случаев, но могут быть и исключения, о них будет сказано ниже;

- ограничения на уровне пользовательских запросов и функций. Сюда входят разнообразные способы использования порядка следования записей в запросе и отсечения «лишних» записей результат запроса, включая предварительные выборки во временную таблицу;

- ограничения серверных курсоров. Итог запроса не возвращается клиенту, на сервере открывается курсор, который позволяет построчно пролистывать результирующее множество данных;

- ограничения клиентских курсоров. Итог запроса целиком закачивается в DataSet клиента, вся дальнейшая навигация происходит в памяти клиентского приложения (им может быть и веб-сервер, и сервер приложений и служба).

Обзор способов постраничной выборки

Способ 1. Ограничения на уровне синтаксиса запроса

К ограничениям уровня синтаксиса SQL запросов относятся, например, возможности MySQL.

```
SELECT *
FROM orders
ORDER BY created ASC
LIMIT 101, 120  /* вывод строк с 101 по 120-ю */
```

Аналогично работают ограничения в PostgreSQL с несколько изменённой логикой.

```
SELECT *
FROM orders
ORDER BY created ASC
LIMIT 20 OFFSET 100  /* вывод строк с 101 по 120-ю */
```

Может показаться странным, но до версии 2012 Microsoft SQL Server не имел в своём арсенале возможностей ограничивать выборки на уровне ядра СУБД, указывая соответствующие критерии в теле запроса. И вот, наконец, у разработчиков этой передовой во многих отношениях СУБД, что называется «дошли руки».

```
SELECT *
FROM orders
ORDER BY created ASC
OFFSET 100 ROW FETCH NEXT 20 ROWS ONLY /* вывод строк с 101
по 120-ю */
```

Как вы могли заметить, во всех примерах имеется конструкция ORDER BY, хотя в случае MySQL и PostgreSQL она не является обязательной. Причина тому родом из теории множеств. Результат SQL-запроса — это неупорядоченное множество записей. Даже если дважды выполненный запрос вернёт одно и то же множество, порядок следования записей в нем не гарантирован. На практике, он зависит от физического размещения

записей в БД, которое может меняться при модификациях данных. Поэтому, возьмите за правило.

> Выборки с ограничениями размера должны сопровождаться упорядочиванием результатов при помощи ORDER BY.

Почему вышеприведённые конструкции являются в большинстве случаев наиболее эффективными с точки зрения производительности?

Явное задание критериев ограничения учитывается при составлении плана запроса. Если на уровне пользовательских функций отсеивается уже результат, то на уровне ядра СУБД оптимизатор может решить, например, прервать внутреннее соединение двух таблиц на 100-й записи.

Способ 2. Использование функций ранжирования

Тем не менее, встречаются случаи, когда ограничить выборку на уровне ядра не является корректным с логической точки зрения. Это многочисленные аналитические запросы с ранжированием и группировками, когда целью стоит, например, «выбрать группы с 10-й по 20-ю». Те же функции можно использовать и для ограничения на уровне строк.

```
WITH ordered_sales AS (
  SELECT
    *,
    row_number() OVER (
      ORDER BY id_product, id_customer, sale_date
    ) AS row_num
    FROM sales
)
SELECT *
FROM ordered_sales
WHERE row_num BETWEEN 101 AND 120;
```

Оборотная сторона такого способа — меньшее быстродействие по сравнению с основным.

Способ 3. Временная таблица

Если к выбранной странице или к их совокупности с 1 по N предполагается обращаться многократно, например, используя в

последующих соединения с другими таблицами, то может пригодиться способ на основе временной таблицы.

```
/* создание структуры временной таблицы */
SELECT * INTO #s FROM sales WHERE 1 = 0;
/* добавляем колонку - номер строки, первичный ключ */
ALTER TABLE #s ADD row_num INT NOT NULL IDENTITY(1, 1)
PRIMARY KEY;
/* заполняем данные */
INSERT INTO #s
SELECT TOP 120 *
FROM sales
ORDER BY id_product, id_customer, sale_date;
/* выборка последней страницы */
SELECT * FROM #s
WHERE row_num BETWEEN 101 and 120;
/* выборка третьей страницы */
SELECT * FROM #s
WHERE row_num BETWEEN 41 and 60;
```

Недостаток способа — необходимость предварительно выбирать все записи до N-й.

Способ 4. Использование SELECT TOP

Может встретиться СУБД, у которой реализовано только ограничение общего числа выводимых строк результата. Тогда решением может служить пересечение двойное использование этой функции со взаимно обратной сортировкой.

```
SELECT *
  FROM
  (
    SELECT TOP 20 *
    FROM
      (SELECT TOP 120 *
         FROM sales
         ORDER BY id_product ASC, id_customer ASC, sale_date
ASC
      ) t1
    ORDER BY id_product DESC, id_customer DESC, sale_date
DESC
  ) t2
  ORDER BY id_product, id_customer, sale_date
```

Способ 5. Серверный курсор

Если необходимо выбирать большие пачки записей из длинной таблицы, то основной способ (1) может оказаться менее производительным, чем серверный курсор, несмотря на большее время первоначальной загрузки. Ниже — пример для SQL Server, подробную документацию можно найти в MSDN по заголовку «Cursor Stored Procedures (Transact-SQL)».

```
DECLARE @handle int, @rows int;
EXEC sp_cursoropen
  @handle OUT,
  'SELECT * FROM sales ORDER BY id_product, id_customer,
sale_date',
    1, /* 0x0001 - тип курсора (keyset-driven cursor) */
    1, /* только чтение */
    @rows OUT; /* общее количество записей */

EXEC sp_cursorfetch
  @handle,
  16,   /* отсчет по абсолютному номеру строки */
  100,  /* пропускаем первые 100 строк */
  20    /* выбираем 20 последующих */

EXEC sp_cursorclose @handle; /* закрываем курсор */
```

Способ 6. Стандартный ANSI SQL и переносимость

Данный способ представляет собой скорее академический интерес, показывая, что даже на минимальном уровне совместимости со стандартом задачу можно единообразно решить на любой СУБД. На практике, его быстродействие с ростом N падает как N^2 (зависимость типа $O(n^2)$), что не позволяет рекомендовать его для использования без дополнительных обоснований.

```
SELECT o.*
FROM orders o
WHERE
  (SELECT count(1)
   FROM orders o1
   WHERE o1.product_code <= 0.product_code
  ) BETWEEN 101 AND 20
  ORDER BY o.product_code ASC
```

Однако, если рассматривать расширенный стандарт ANSI SQL, то в него входит функция `row_number()`, реализованная во многих СУБД. Соответственно, вышеприведённый способ №2 также может быть отнесён к категории переносимых между СУБД методов.

Тестирование способов постраничной выборки

Приведём подробное описание испытаний, чтобы читатели смогли понять суть подходов к такого рода практическим проверкам. В качестве подопытной системы используется Microsoft SQL Server 2012, но аналогичные тесты могут быть проведены на любой другой СУБД с небольшими изменениями SQL-сценариев.

Пусть имеется две таблицы: клиенты и продажи. Клиенты расположены в 7 странах. Необходимо выбирать информацию о продажах пачками по N записей относящиеся ко всем клиентам заданной страны продажи. В сценарии задан фильтр по Италии (IT), но при относительно равномерном распределении данных, использующихся в тесте, выбор не является существенным. Результаты прогонов заносятся в специально созданную для этих целей таблицу.

Конфигурация оборудования в данном испытании также не важна, поскольку проводится лишь относительное сравнение разных способов на одной и той же БД. Разумеется, на более быстром сервере ждать итогов придётся меньше, а объёмы данных можно нарастить.

Вначале создадим БД и таблицы, это будет первым SQL-сценарием в серии.

```
SET NOCOUNT ON;
CREATE DATABASE test_paging
GO
ALTER DATABASE test_paging SET RECOVERY SIMPLE
GO
ALTER DATABASE test_paging MODIFY FILE
(NAME=N'test_paging', SIZE=1024MB, MAXSIZE=UNLIMITED,
FILEGROWTH=512MB )
GO
ALTER DATABASE test_paging MODIFY FILE
```

```
(NAME=N'test_paging_log', SIZE=512MB, MAXSIZE=UNLIMITED,
FILEGROWTH=256MB)
GO

USE test_paging
GO

CREATE TABLE dbo.customers (
  id_customer    int           NOT NULL,
  country_code   nchar(2)      NOT NULL,
  name           nvarchar(255) NULL,
  street_address nvarchar(100) NULL,
  city           nvarchar(40)  NULL,
  postal_code    nvarchar(15)  NULL,
  CONSTRAINT pk_customers
    PRIMARY KEY CLUSTERED (id_customer)
)
GO
CREATE INDEX IX1_COUNTRY_CODE ON dbo.customers (country_code
ASC)
GO

CREATE TABLE dbo.sales (
  id_product  int      NOT NULL,
  id_customer int      NOT NULL,
  sale_date   datetime NOT NULL,
  qty         int      NOT NULL,
  CONSTRAINT pk_sales
    PRIMARY KEY CLUSTERED (id_product, id_customer,
sale_date),
  CONSTRAINT fk1_orders_customers FOREIGN KEY(id_customer)
    REFERENCES dbo.customers (id_customer)
)
GO

CREATE TABLE dbo.results (
  method int NOT NULL,
  offset int NOT NULL,
  page_size int NOT NULL,
  attempt int NOT NULL,
  duration_msec int NOT NULL,
  CONSTRAINT pk_results
    PRIMARY KEY CLUSTERED (method, offset, page_size,
attempt)
)
```

```
GO
```

Второй сценарий заполняет таблицы псевдослучайным образом сгенерированными данными.

```
USE test_paging
GO
CREATE VIEW dbo.rand2 AS
  SELECT rand(abs(convert(int, convert(varbinary, newid()))))
AS rand_value;
GO
SET NOCOUNT ON;

DECLARE
  @max_products_count int = 1000,
  @max_customers_count int = 10000,
  @max_sales_count int = 10000000,
  @min_date datetime = '20100101',
  @max_qty int = 1000;
DECLARE
  @max_days int = datediff(day, @min_date, getdate()),
  @i int;

TRUNCATE TABLE dbo.sales;
DELETE FROM dbo.customers;

PRINT 'Insert customers...'
SET @i = 1;
DECLARE @id_country int;
BEGIN TRANSACTION
WHILE @i <= @max_customers_count BEGIN
  SET @id_country = floor((SELECT rand_value * 7 FROM
rand2));
  INSERT INTO dbo.customers (id_customer, country_code, name,
street_address, city, postal_code)
  SELECT
    @i,
    CASE @id_country
      WHEN 0 THEN 'ES'
      WHEN 1 THEN 'FR'
      WHEN 2 THEN 'GE'
      WHEN 3 THEN 'IT'
      WHEN 4 THEN 'NL'
      WHEN 5 THEN 'RU'
      WHEN 6 THEN 'UK'
    END AS country_code,
```

```
    'Name' + convert(nvarchar(16), floor((SELECT rand_value
FROM rand2) * @max_customers_count + 1)),
    'Street ' + convert(nvarchar(16), floor((SELECT
rand_value FROM rand2) * 100000 + 1)),
    'City' + convert(nvarchar(16), floor((SELECT rand_value
FROM rand2) * 1000 + 1)),
    convert(nvarchar(16), floor((SELECT rand_value FROM
rand2) * 100000 + 1));
  IF @i % 1000 = 0 BEGIN
    COMMIT TRANSACTION
    BEGIN TRANSACTION
  END
  SET @i = @i + 1;
END
COMMIT TRANSACTION
PRINT 'Finished'

PRINT 'Insert sales data...'
SET @i = 1;
BEGIN TRANSACTION
WHILE @i <= @max_sales_count BEGIN
  INSERT INTO dbo.sales (id_product, id_customer, sale_date,
qty)
  SELECT
    floor((SELECT rand_value FROM rand2) *
@max_products_count + 1),
    floor((SELECT rand_value FROM rand2) *
@max_customers_count + 1),
    dateadd(second, @i + floor((SELECT rand_value * 10 FROM
rand2)), @min_date),
    floor((SELECT rand_value FROM rand2) * @max_qty + 1)
  ;
  IF @i % 1000 = 0 BEGIN
    COMMIT TRANSACTION
    BEGIN TRANSACTION
  END
  IF @i % 100000 = 0
    PRINT convert(nvarchar(16), @i) +  ' processed';
  SET @i = @i + 1;
END
COMMIT TRANSACTION
PRINT 'Finished'
GO

DROP VIEW dbo.rand2
```

```
GO

SELECT count(1) AS sales_count, c.country_code
FROM dbo.sales s INNER JOIN dbo.customers c ON s.id_customer
= c.id_customer
GROUP BY c.country_code
ORDER BY c.country_code
```

Третий сценарий создаёт хранимые процедуры, по одной на каждый из испытываемых способов.

```
USE test_paging
GO

IF EXISTS(SELECT 1 FROM sys.views WHERE object_id =
OBJECT_ID(N'dbo.test_sales_data'))
  DROP VIEW dbo.test_sales_data
GO
CREATE VIEW dbo.test_sales_data
AS
SELECT
  s.id_customer,
  s.id_product,
  s.qty,
  s.sale_date,
  c.country_code,
  c.name
FROM
  dbo.sales s
  INNER JOIN dbo.customers c
    ON s.id_customer = c.id_customer
WHERE
  c.country_code = 'IT'
GO

IF EXISTS(SELECT 1 FROM sys.procedures WHERE object_id =
OBJECT_ID(N'dbo.test_paging_m1'))
  DROP PROCEDURE dbo.test_paging_m1
GO
CREATE PROCEDURE dbo.test_paging_m1
  @offset int,
  @page_size int
AS
BEGIN
  SET NOCOUNT ON;
```

```
  SELECT * FROM dbo.test_sales_data
  ORDER BY id_product, id_customer, sale_date
  OFFSET @offset - 1 ROW FETCH NEXT @page_size ROWS ONLY
END
GO

IF EXISTS(SELECT 1 FROM sys.procedures WHERE object_id =
OBJECT_ID(N'dbo.test_paging_m2'))
  DROP PROCEDURE dbo.test_paging_m2
GO
CREATE PROCEDURE dbo.test_paging_m2
  @offset int,
  @page_size int
AS
BEGIN
  SET NOCOUNT ON;
  WITH ordered_sales AS (
    SELECT
      *,
      row_number() OVER( ORDER BY id_product, id_customer,
sale_date) AS row_num
    FROM dbo.test_sales_data
  )
  SELECT *
    FROM ordered_sales
    WHERE row_num BETWEEN @offset AND @offset + @page_size -
1;
END
GO

IF EXISTS(SELECT 1 FROM sys.procedures WHERE object_id =
OBJECT_ID(N'dbo.test_paging_m3'))
  DROP PROCEDURE dbo.test_paging_m3
GO
CREATE PROCEDURE dbo.test_paging_m3
  @offset int,
  @page_size int
AS
BEGIN
  SET NOCOUNT ON;
  -- Temporary table
  SELECT * INTO #s FROM dbo.test_sales_data WHERE 1 = 0;
  ALTER TABLE #s ADD row_num INT NOT NULL IDENTITY(1, 1)
PRIMARY KEY;
```

```
  INSERT INTO #s
  SELECT TOP (@offset + @page_size - 1) * FROM
dbo.test_sales_data
  ORDER BY id_product, id_customer, sale_date;

  SELECT * FROM #s
  WHERE row_num BETWEEN @offset and @offset + @page_size - 1;
END
GO

IF EXISTS(SELECT 1 FROM sys.procedures WHERE object_id =
OBJECT_ID(N'dbo.test_paging_m4'))
  DROP PROCEDURE dbo.test_paging_m4
GO
CREATE PROCEDURE dbo.test_paging_m4
  @offset int,
  @page_size int
AS
BEGIN
  SET NOCOUNT ON;

  SELECT *
  FROM
  (
    SELECT TOP (@page_size) *
    FROM
      (SELECT TOP (@offset + @page_size - 1) *
         FROM dbo.test_sales_data
         ORDER BY id_product ASC, id_customer ASC, sale_date
ASC
      ) t1
    ORDER BY id_product DESC, id_customer DESC, sale_date
DESC
  ) t2
  ORDER BY id_product, id_customer, sale_date
END
GO

IF EXISTS(SELECT 1 FROM sys.procedures WHERE object_id =
OBJECT_ID(N'dbo.test_paging_m5'))
  DROP PROCEDURE dbo.test_paging_m5
GO
CREATE PROCEDURE dbo.test_paging_m5
  @offset int,
  @page_size int
```

```
AS
BEGIN
  SET NOCOUNT ON;

  DECLARE @handle int, @rows int;
  EXEC sp_cursoropen
    @handle OUT,
    'SELECT * FROM dbo.test_sales_data ORDER BY id_product,
id_customer, sale_date',
    1,
    -- 16 - 0x0010 - Fast forward-only cursor
    -- 1 - 0x0001 - Keyset-driven cursor
    -- 8 - 0x0008 - Static cursor
    --select cast(0x0008 as int)
    1, -- Read-only
    @rows OUT; -- Contains total rows count

  EXEC sp_cursorfetch
    @handle,
    16,        -- Absolute row index
    @offset,   -- Fetch from row
    @page_size -- Rows count to fetch

  EXEC sp_cursorclose @handle;
END
GO
```

Четвёртый сценарий производит запуск тестов, ведя запись хронологии и результатов в соответствующей таблице. В сценарии заданы повторения по 4 попытки для каждого из 5 способов. Параметром @offset_count можно регулировать

```
USE test_paging
GO
SET NOCOUNT ON;

DECLARE
  @attempt_count int = 4,
  @methods_count int = 5,
  @offset_count int = 9,
  @method int,
  @offset int,
  @page_size int,
  @test_proc_name sysname,
  @started_at datetime,
```

```
  @i int,
  @j int;

TRUNCATE TABLE dbo.results;

SET @method = 1;
WHILE @method <= @methods_count BEGIN
  SET @test_proc_name = N'dbo.test_paging_m' +
convert(nvarchar(2), @method);

  SET @offset = 1;
  SET @page_size = 100;
  SET @i = 1;
  WHILE @i <= @offset_count BEGIN
    SELECT @offset =
      CASE
        WHEN @i = 1 THEN 1
        WHEN @i = 2 THEN 100
        WHEN @i = 3 THEN 1000
        WHEN @i = 4 THEN 10000
        WHEN @i >= 5 THEN 100000 * (@i - 4)
      END;
    DBCC DROPCLEANBUFFERS;
    DBCC FREEPROCCACHE;
    SET @j = 1;
    WHILE @j <= @attempt_count BEGIN
      SET @started_at = getdate();
      EXEC @test_proc_name @offset = @offset, @page_size =
@page_size;
      INSERT INTO dbo.results (method, offset, page_size,
attempt, duration_msec)
      SELECT @method, @offset, @page_size, @j,
datediff(millisecond, @started_at, getdate());
      SET @j = @j + 1;
    END;
    SET @i = @i + 1;
  END;
  SET @method = @method + 1;
END;
GO
```

Запуск этого сценария следует выполнять из командной строки, чтобы не тратить время на заполнение графического представления таблиц, возвращаемых запросами. Например, так.

```
sqlcmd.exe -S localhost -d test_paging -E -i 04_Test.sql -o
Test.log
```

Подробнее, смотрите опции запуска утилиты `sqlcmd` в документации,

Наконец, самый простой заключительный скрипт, выводящий результаты: отдельно для первой попытки и усреднённые по совокупности всех последующих. Первая таблица, соответственно, даёт значения времени выполнения «холодных» запросов (при очищенном кэше СУБД), вторая — среднее время из трёх (по умолчанию) попыток при наличии данных и скомпилированного запроса в кэше.

```
SELECT *
FROM
(
  SELECT method, offset, page_size, duration_msec
  FROM dbo.results
  WHERE attempt = 1
) p
PIVOT
(
  AVG(duration_msec)
  FOR method IN ([1], [2], [3], [4], [5])
) pvt

SELECT *
FROM
(
  SELECT method, offset, page_size, duration_msec
  FROM dbo.results
  WHERE attempt > 1
) p
PIVOT
(
  AVG(duration_msec)
  FOR method IN ([1], [2], [3], [4], [5])
) pvt
```

При воспроизведении тестов на относительно слабом компьютере уровня рабочей станции, получаются следующие результаты (размер страницы N = 100 записей).

Табл. 20. Результаты прогона «холодных» запросов, миллисекунды

offset	1	2	3	4	5
1	6690	6693	6763	6906	35033
100	7260	6790	7190	7083	34736
1000	9230	8996	8570	8663	34653
10000	9330	8916	8570	7200	35433
100000	14946	15126	16156	15180	35103
200000	21120	20216	22863	21126	35736
300000	25223	23210	26063	24970	35473
400000	29923	25690	31336	29846	34933
500000	29326	28240	31526	30300	34750

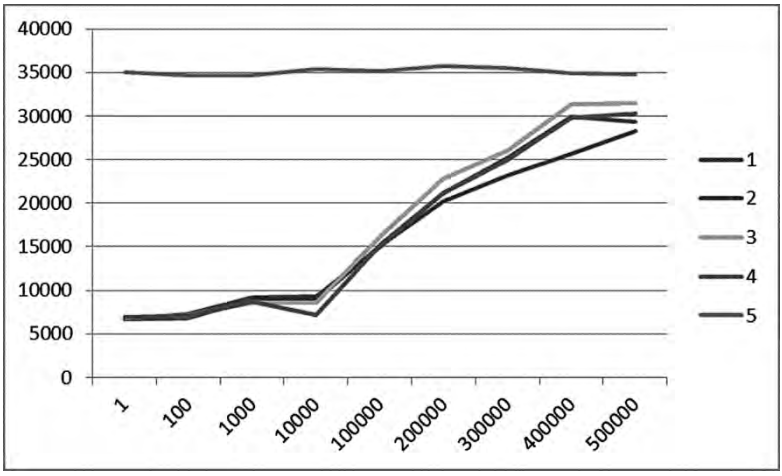

Рис.59. Графики результатов прогона «холодных» запросов

Табл. 21. Результаты прогона «горячих» запросов, среднее время, миллисекунды

offset	1	2	3	4	5
1	7	9	23	13	6669
100	16	12	24	16	6617
1000	33	34	48	37	7048
10000	140	210	243	131	6662
100000	1037	2084	1711	1182	6632
200000	3291	4206	4477	3218	6678
300000	4754	6340	5997	4923	6640
400000	1634	8407	2981	2006	6612
500000	1642	10641	3232	2196	6717

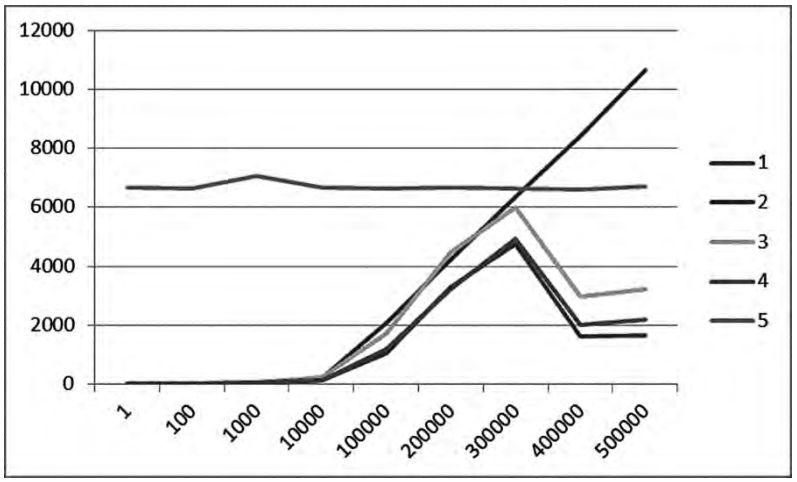

Рис.60. Графики результатов прогона «горячих» запросов

Выводы

Испытания на примере MS SQL Server 2012 показывают, что штатный способ №1 ограничения размера выборки на уровне синтаксиса SQL-запроса является одним из наиболее быстрых.

Между тем, единственным способом, показавшим стабильно одинаковое время отклика независимо от параметров выборки, является метод серверного курсора №5.

SQL и модульное тестирование

Место модульного тестирования в системе испытаний

Важнейший этап разработки программной системы — её испытание на соответствие требованиям. Требования к системе распределяются по уровням детализации, и, значит, соответствующие им тесты имеют различную форму и ответственных за их создание. На самом нижнем уровне располагаются модульные тесты, для которых также используется жаргонное словечко «юнит-тест», калька с английского термина unit test.

Табл. 22. Уровни испытаний и требований

Уровень	Тип требований	Источник, документ	Вид теста	Ответственный
Система	Требования к системе	Техническое задание	Системный тест (system test)	Главный инженер/ конструктор
Подсистема (компонент, пакет)	Требования к подсистеме	Технический проект	Интеграционный тест (integration test)	Ведущий инженер-программист
Модуль	Требования к модулю	Спецификация модуля	Модульный тест (unit test)	Инженер-программист

Как следует из таблицы, ответственным за создание модульных тестов является инженер-программист, разрабатывающий данный модуль. Модульные тесты относятся к категории «белый ящик», потому что в отличие от ящика чёрного, внутреннее устройство испытуемого объекта известно.

Рекламируемые в последние годы методики разработки «от тестов» (TDD - Test Driven Development) базируются на обязательном создании модульных тестов ещё до написания собственно кода. Преследуется цель 100 %го покрытия тестами кода приложения, где под покрытием имеется в виду отношение числа тестируемых функций/методов модуля к общему их числу. Разумеется, чем больше модульных тестов и больше покрытие, тем выше степень надёжности отдельных модулей. Однако подобная методика имеет и видимые недостатки:

- корректное выполнение модульных тестов не гарантирует соответствие системы требованиям на вышестоящих уровнях;
- сложность разработки модульных тестов сравнима со сложностью разработки тестируемого кода. Таким образом, общее время программирования увеличивается в 2-3 раза.

В самом деле, работоспособность отдельных компонентов компьютера вовсе не гарантирует, что, собранный вашими руками из купленной россыпи деталей, он заработает сразу и с нужной производительностью. Нельзя также быть уверенным, что все приложения будут корректно

функционировать после установки. Именно поэтому поставщики компьютеров и ПО предлагают клиентам совместимые конфигурации, прошедшие системные испытания.

Особенности разработки на процедурных расширениях SQL

В традиционных средах программирования, таких как C++, Delphi, Java, C# и др., модульное тестирование — обычная практика: имеются соответствующая инфраструктура, интегрированная со средой разработки, библиотеки и стандартные методики создания и прогона тестов вручную или в автоматическом режиме.

Программисты приложений баз данных, разрабатывающие много серверного кода на SQL и его процедурных расширениях, оказываются в худшем положении: поставщики СУБД, как правило, не включают в комплект не только инструменты для модульного тестирования, но и порой даже простые средства отладки и трассировки. Следует сказать, что отсутствие пошаговой отладки не является в данной области критичным фактором. Далеко не все СУБД поддерживают и подобие модульности для разрабатываемых хранимых процедур и функций. Задача создания инфраструктуры, инструментов и методики для модульного тестирования ложится на плечи программиста.

Пример задачи для модульного теста

Рассмотрим пример разработки с использованием модульных тестов, реализованный для Microsoft SQL Server (версия 2008 и выше). Общие принципы подхода не привязаны к СУБД, поэтому вы сможете использовать его и в других случаях с минимальными изменениями.

Положим, на базе имеющейся ежедневной статистики продаж продукции в магазинах требуется составить недельные прогнозы продаж на заданный период методом простой экстраполяции. Упрощённая схема данных будет выглядеть следующим образом.

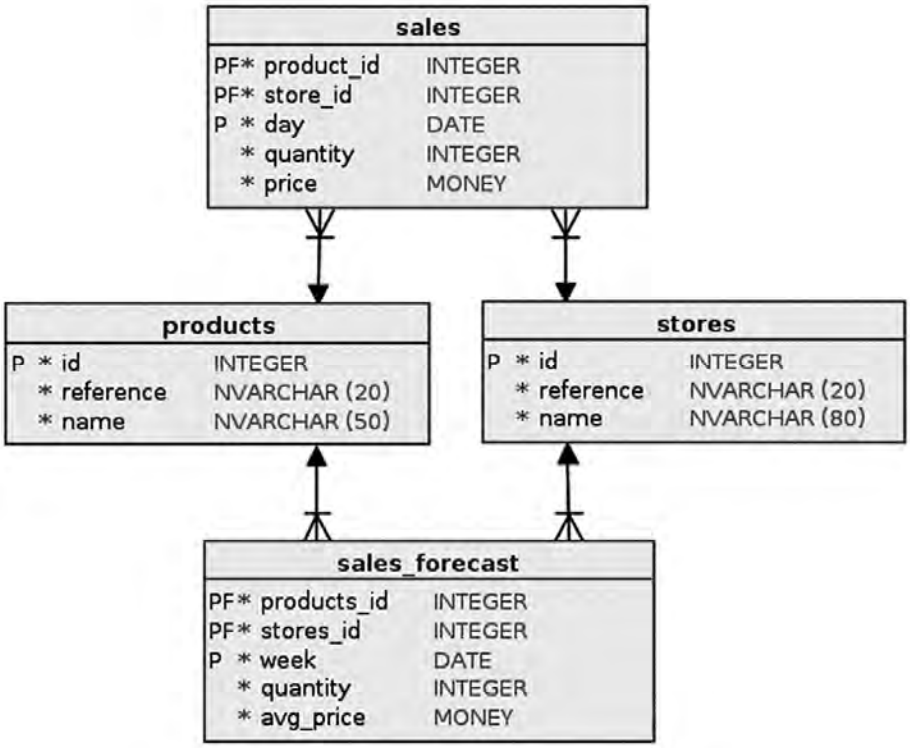

Рис.61. Схема данных для примера модульного теста

В таблицах `products` и `stores` хранятся список товаров и данные о магазинах. В таблице `sales` ведётся статистика продаж: количество и цена — в разрезе «продукт—магазин—день». А в таблицу `sales_forecasts` нужно внести данные прогноза продаж: количество и среднюю цену.

Функция расчёта упрощена: на входе имеем даты начала и окончания будущего периода и начальную дату прошлого. На базе этих сведений и будет сделан линейный прогноз. Данные по продажам консолидируются с уровня дней до недель, причём их количество суммируется, а цена вычисляется средняя.

Исходные тексты программ примера находится в приложении к книге, здесь же мы приведём только основные элементы. Хранимая процедура `sales_forecast_init` (исходный файл `SalesForecast.sql`), производит вычисления и заполняет таблицу прогноза.

```
CREATE PROCEDURE dbo.sales_forecast_init
```

```
   @first_week     date,
   @last_week      date,
   @first_week_ref date
AS
BEGIN
  SET NOCOUNT ON;

  DELETE FROM sales_forecast
  WHERE week BETWEEN @first_week AND @last_week;

  INSERT INTO sales_forecast
    (store_id,
     product_id,
     week,
     quantity,
     avg_price)
  SELECT s.store_id,
         s.product_id,
         w.start_date,
         sum(s.quantity),
         avg(s.price)
  FROM sales AS s
       LEFT OUTER JOIN
       dbo.get_weeks_list(@first_week, @last_week) AS w
         ON s.day BETWEEN
           dateadd(ww,
                   datediff(ww, @first_week, w.start_date),
                   @first_week_ref)
           AND
           dateadd(ww,
                   datediff(ww, @first_week, w.start_date) +
1,
                   @first_week_ref)
  GROUP BY s.store_id,
           s.product_id,
           w.start_date;
END;
```

Функция `utils_get_weeks_list()` возвращает таблицу-список недель между двумя заданными датами. Неделя задаётся датой первого дня (понедельника). Например, вызов функции

```
utils_get_weeks_list('20080204', '20080218')
```

возвратит таблицу из трёх строк:

week_num	start_date
1	2008-02-04
2	2008-02-11
3	2008-02-18

Теперь требуется создать модульные тесты для проверки не только нашей основной процедуры, но и вспомогательных функций.

Создаём специализированный макроязык

Хотя в столь простом случае можно обойтись исключительно средствами самого Transact SQL, решение получилось бы достаточно громоздким. Например, нам понадобятся стандартные процедуры проверок типа `assert`, генерирующие ошибку, если величина не равна/равна/больше/меньше заданной. Но в Transact SQL при вызове процедур нельзя напрямую передавать им значения, полученные из SQL-запросов или других функций. Следовательно, придётся всякий раз объявлять и инициализировать локальные переменные, а затем передавать их в процедуру проверки. Избежать этих и многих других неудобств поможет макропрограммирование.

Определим несколько макросов, которые будем использовать в тексте процедур на Transact SQL. Перед трансляцией процедуры в СУБД исходный текст проходит предварительную обработку макропроцессором, макросы раскрываются, получается чистый SQL. Подобный принцип активно применяется в языках семейства Си/C++. По своей сути и назначению же макропрограммирование позволяет обойти ограничения любого языка и создать поверх него собственный предметный язык более высокого уровня для эффективного решения частных задач.

В качестве макропроцессора используется `GNU m4`, входящий в стандартный инструментарий для любого Linux-окружения. Версия m4 для Windows включена в пример.

Исходные файлы SQL (с макросами) имеют расширения `.sqm`, но это ограничение не строгое, а введённое лишь с целью различать исходники на SQL с внедрёнными макросами от чистого Transact SQL. Обработанный

макропроцессором файл SQM превращается в SQL после запуска трансляции из командной строки.

```
D:\sqlunit\SalesTest>m4 -I .\Include
UtilsTest.sqm >UtilsTest.sql
```

Текст процедуры модульного теста функции `utils_get_weeks_list()` с использованием макросов будет выглядеть более наглядно, чем если бы мы остались в рамках Transact SQL.

```
include(Common.sqh)
include(MSSQL.sqh)
include(SqlUnit.sqh)

DeclareProcedure(dbo.test_utils_get_weeks_list)
AS
BEGIN
   SET NOCOUNT ON;

   DECLARE @start_date   date,
           @end_date     date,
           @weeks_to_add int;

   SELECT @start_date   = getdate(),
          @weeks_to_add = 7;
   SELECT @end_date = dateadd(ww, @weeks_to_add,
@start_date);

   SQLUnit_AreEquals(
      int,
      {SELECT count(*)
       FROM dbo.utils_get_weeks_list(@start_date,
@end_date)},
      @weeks_to_add + 1);

   SQLUnit_AreEquals(
      int,
      {SELECT max(week_num)
       FROM dbo.utils_get_weeks_list(@start_date,
@end_date)},
      @weeks_to_add + 1);
END;
```

Если посмотреть на результат раскрытия макроса `SQLUnit_AreEquals` в сгенерированном файле `UtilsTest.sql`, то можно увидеть кусок

рутинного кода, не подлежащего факторизации и оформлению в виде хранимой процедуры или функции.

```
DECLARE
    @Val18 int,
    @Val19 int,
    @ValStr18 nvarchar(255),
    @GivenValStr18 nvarchar(255);
SELECT
    @Val18 = (SELECT count(*) FROM
dbo.utils_get_weeks_list(@start_date, @end_date)),
    @Val19 = (@weeks_to_add + 1);
IF (@Val18 IS NULL OR NOT(@Val18 = @Val19))
BEGIN
    SELECT @ValStr18 = convert(nvarchar(255), @Val18),
        @GivenValStr18 = convert(nvarchar(255), @Val19);
    raiserror('Error. Value is "%s". Expected "%s".
Expression: SELECT count(*) FROM
dbo.utils_get_weeks_list(@start_date, @end_date)', 16, 1,
@ValStr18, @GivenValStr18);
END;
```

Как можно увидеть, четыре строки макроса проверки целочисленных результатов двух выражений на равенство раскрываются в 14 строк чистого Transact SQL! А ведь подобных проверок в теле рядовой процедуры теста используется в среднем около десятка.

Теперь взглянем на исходный текст модульных тестов в файле `SalesForecastTest.sqm`. В процедуре `test_sales_forecast_setup` производится заполнение таблиц временными данными для теста.

```
DeclareProcedure(dbo.test_sales_forecast_setup)
AS
BEGIN
   SET NOCOUNT ON;

   /* Fill test data */
   INSERT INTO products(id, reference, name)
   SELECT newid(), 'TEST#prod1', 'Product 1'
   UNION
   SELECT newid(), 'TEST#prod2', 'Product 2'
   UNION
   SELECT newid(), 'TEST#prod3', 'Product 3'
```

```
   INSERT INTO stores(id, reference, name)
   SELECT newid(), 'TEST#store1', 'Store1 Msk'
   UNION
   SELECT newid(), 'TEST#store2', 'Store2 SPb';


   INSERT INTO sales (store_id, product_id, [day], quantity,
price)
   SELECT s.id, p.id, d.start_date,
          convert(int, ((SELECT rand_value FROM rand2) *
1000)),
          (SELECT rand_value FROM rand2) * 100.00
   FROM products p
        CROSS JOIN stores s
        CROSS JOIN
          dbo.utils_get_days_list(TestWeek11, TestWeek12) d
   WHERE p.reference LIKE 'TEST#%' AND
         s.reference LIKE 'TEST#%'

   SQLUnit_MoreThan(
      {int},
      {SELECT count(*)
       FROM sales
            INNER JOIN stores ON sales.store_id = stores.id
            INNER JOIN products
              ON sales.product_id = products.id
       WHERE stores.reference LIKE 'TEST#%' AND
             products.reference LIKE 'TEST#%' AND
             day BETWEEN TestWeek11 AND TestWeek12
      },
      {0},
      {Sales table has no data});
END;
```

В процедуре test_sales_forecast_teardown, производящей очистку, мы удаляем эти данные.

```
DeclareProcedure(dbo.test_sales_forecast_teardown)
AS
BEGIN
   SET NOCOUNT ON;

   DELETE FROM sales_forecast
   WHERE product_id IN
     (SELECT id FROM products WHERE reference LIKE 'TEST#%');
```

```
   DELETE FROM sales
   WHERE product_id IN
     (SELECT id FROM products WHERE reference LIKE 'TEST#%');

   DELETE FROM products WHERE reference LIKE 'TEST#%';
   DELETE FROM stores WHERE reference LIKE 'TEST#%';
END;
```

Термины *setup* (инициализация) и *teardown* (очистка) — стандарты де-факто в модульном тестировании, поэтому мы их не изменяем. Сам тест проводится в процедуре `test_sales_forecast_init`, также использующей макросы.

```
DeclareProcedure(dbo.test_sales_forecast_init)
AS
BEGIN
   SET NOCOUNT ON;

   DECLARE @first_week     date,
           @last_week      date,
           @first_week_ref date;
   SELECT
      @first_week     = TestWeek21,
      @last_week      = TestWeek22,
      @first_week_ref = TestWeek11;

   EXEC dbo.sales_forecast_init
      @first_week     = @first_week,
      @last_week      = @last_week,
      @first_week_ref = @first_week_ref;

   SQLUnit_MoreThan(
      int,
      {SELECT count(*)
       FROM sales_forecast
            INNER JOIN stores
              ON sales_forecast.store_id = stores.id
            INNER JOIN products
              ON sales_forecast.product_id = products.id
       WHERE stores.reference LIKE 'TEST#%' AND
                products.reference LIKE 'TEST#%' AND
                week BETWEEN @first_week AND @last_week
      },
      {0},
      {Sales forecasts has no data});
```

```
   /* Провера расчета на случано выбранную неделю */
   DECLARE @store_id      uniqueidentifier,
           @product_id    uniqueidentifier,
           @week_to_check date,
           @ref_week      date;

   SELECT @store_id = id FROM stores WHERE reference =
'TEST#store1';
   SELECT @product_id = id FROM products WHERE reference =
'TEST#prod2';
   SELECT @week_to_check =
            dateadd(ww,
                    datediff(ww, @first_week, @last_week) *
                    (SELECT rand_value FROM rand2),
                    @first_week);
   SELECT @ref_week =
            dateadd(ww,
                    datediff(ww, @first_week,
@week_to_check),
                    @first_week_ref);
   SQLUnit_AreEquals(
      int,
      {SELECT quantity
       FROM sales_forecast
       WHERE store_id = @store_id AND
             product_id = @product_id AND
             week = @week_to_check
      },
      {SELECT SUM(quantity)
       FROM sales
       WHERE store_id = @store_id AND
             product_id = @product_id AND
             [day] BETWEEN @ref_week AND
               dateadd(ww, 1, @ref_week)
      },
      {Invalid quantity});

   SQLUnit_AreEquals(
      int,
      {SELECT avg_price
       FROM sales_forecast
       WHERE store_id = @store_id AND
             product_id = @product_id AND
             week = @week_to_check
```

```
      },
      {SELECT AVG(price)
       FROM sales
       WHERE store_id = @store_id AND
             product_id = @product_id AND
             [day] BETWEEN @ref_week AND
               dateadd(ww, 1, @ref_week)
      },
      {Invalid average price});
END;
```

Запуск теста выполняет процедура `test_sales_forecast_all`.

```
DeclareProcedure(dbo.test_sales_forecast_all)
AS
BEGIN
   SET NOCOUNT ON;
   EXEC dbo.test_sales_forecast_setup;
   EXEC dbo.test_sales_forecast_init;
   EXEC dbo.test_sales_forecast_teardown;
END;
```

Теперь, имея единую точку входа для запуска всех тестов, мы можем автоматизировать процесс с помощью обычного командного файла `run_tests.cmd` и утилиты выполнения SQL из командной строки, поставляемой с любой СУБД. Для MS SQL Server это уже упоминавшаяся `sqlcmd.exe`.

```
@echo off

rem установка переменных окружения
call "%~d0%~p0..\set_env.cmd"

echo.Testing utils module...
sqlcmd -b -r 0 -E -S %SERVER_NAME% -d %DATABASE_NAME% -Q
"EXEC dbo.test_utils_all" -o %OUTPUT_FILE%
if errorlevel 1 goto batch_failed

echo.Testing sales forecast module...
sqlcmd -b -r 0 -E -S %SERVER_NAME% -d %DATABASE_NAME% -Q
"EXEC dbo.test_sales_forecast_all"  -o %OUTPUT_FILE%
if errorlevel 1 goto batch_failed

goto all_done
```

```
:batch_failed
echo Test FAILED
exit /b 1

:all_done
echo Test OK
exit /b 0
```

Запускаем командный файл в консоли Windows и видим непосредственный итог наших тестов. Возврат статуса выполнения (0 или 1 при ошибке) позволяет встроить запуск модульных тестов в систему их автоматического выполнения в рамках постоянной интеграции.

Остановиться и оглянуться

Обратите внимание, процедура `test_sales_forecast_init` проводит весьма простую проверку: мы сверяем цифры по одному товару и одному магазину за единственную неделю, выбранную случайным образом из заданного диапазона. При этом её текст даже с применением лаконичных макросов растягивается на 80 строк. Без использования — на 110 строк согласно сгенерированному SQL-файлу. Если ещё учесть примерно 80 строк предварительной инициализации и очистки, то получается очень много, ведь текст собственно тестируемой процедуры `sales_forecast_init` занимает всего 33 строки! Соотношение объёма основного кода к тестирующему получается около одного к четырём-пяти.

> Существует рабочее эмпирическое правило: если тест получился короче тестируемой процедуры, значит он недостаточно её тестирует.

Данная картина типична для разработки с использованием модульных тестов, где отношение объёма тестируемого кода к тестам примерно один к двум-четырём. Отсюда и неизбежные дополнительные затраты времени, превышающие создание собственно кода приложения. Однако труд не пропадёт даром: надёжность вашего модуля возрастёт, а процесс поиска и предупреждения ошибок будет систематичен.

Окончательный выбор оптимального покрытия, соотношения между объёмом программного кода модулей и модульных тестов, будет зависеть от

многих факторов, в первую очередь, от степени критичности приложения и стоимости простоя в эксплуатации.

Производительность SQL-запросов

Общие рекомендации

Прежде чем приступать к следующим главам, посвящённым выявлению проблем, хотелось бы дать несколько общих рекомендаций по программированию на SQL, с целью минимизации самой возможности их появления.

Уже упоминалась особенность реляционных СУБД — множественная обработка. Забудьте про переменные, циклы, про указатель this. «This» теперь указывает не на объект, а на множество, и операции будут проводиться одновременно со всеми элементами множества. Если в традиционном программировании для изменения элементов списка по условию надо писать цикл, то в SQL все решается одним оператором.

Проведите несколько вечеров с книжкой «Введение в SQL» [3], чтобы на несложных примерах понять всю мощь множественного подхода к обработке данных.

Не используйте курсоры и навигационный подход. Практика показывает, что любую задачу, использующую курсоры, можно перепрограммировать без него. Производительность при этом возрастает на порядки.

Для функциональной декомпозиции используйте процедуры и функции, принимающие и/или возвращающие множества. В противном случае вам придётся программировать курсоры, вызывающие на каждую строку соответствующую функцию.

```
/* Такая реализация может привести
   к навигационным подходам и курсорам */
CREATE PROCEDURE check_qty(@qty integer, @id_sku integer)
AS
BEGIN
  SET NOCOUNT ON;
  IF @qty > (SELECT qty FROM stock WHERE id_sku = @id_sku)
    RAISERROR('Не хватает запаса!', 11, 1)
END
```

```
GO

/* Реализация на основе множественного подхода */
CREATE TYPE t_requirement_list AS TABLE (
  id integer,
  qty integer)
GO

CREATE PROCEDURE check_qty(
  @quantities t_requirement_list READONLY
)
AS
BEGIN
  SET NOCOUNT ON;
  IF EXISTS(SELECT 1
            FROM stock s
                 INNER JOIN @quantities q ON s.id_sku = q.id
            WHERE q.qty > s.qty)
    RAISERROR('Не хватает запаса!', 11, 1)
END
GO
```

Используйте предикат EXISTS() вместо SELECT COUNT(1), потому что для проверки существования извлекается только первый элемент, а подсчёт количества просканирует все выбранные строки.

```
/* Плохой код */
SELECT *
FROM sales s
WHERE (SELECT COUNT(1)
       FROM sales s2
       WHERE s.id_customer <> s2.id_customer AND
             s.id_product = s2.id_product AND
             s.qty = s2.qty) > 0
/* Так будет быстрее */
SELECT *
FROM sales s
WHERE EXISTS(SELECT 1
             FROM sales s2
             WHERE s.id_customer <> s2.id_customer AND
                   s.id_product = s2.id_product AND
                   s.qty = s2.qty)
```

Вместо связанных подзапросов старайтесь использовать обычные соединения. Не все оптимизаторы запросов могут отловить эту ситуацию.

```
/* Связанный подзапрос может привести
   к вложенным циклам в плане выполнения */
SELECT *
FROM sales s
WHERE
  qty < (SELECT AVG(qty)
         FROM sales s2
         WHERE s2.id_product = s.id_product)

/* Соединение может выполняться разными
   и более быстрыми способами */
SELECT s.*
FROM sales s
     INNER JOIN
     (
       SELECT AVG(qty) AS avg_qty, id_product
       FROM sales s2
       GROUP BY id_product
     ) s3
       ON s.id_product = s3.id_product
WHERE s.qty < s3.avg_qty
```

Анализ плана выполнения запроса

Как в общем случае понять, эффективен ли запрос? Если в среде разработки, где данные далеки от реальных по объёмам, запрос будет выполняться быстро, это не гарантирует проблемы при передаче очередной версии на тестирование.

Объективная оценка основывается на анализе плана выполнения запроса на самых ранних этапах. Будет лучше, если объёмы данных в среде разработки будут хотя бы одного порядка с предполагаемыми. Если это по каким-либо причинам невозможно, остановитесь на генерации тестовых данных разработчика порядка сотен тысяч строк — это объёмы, когда разница между плохим и хорошим планами будет уже заметна.

Как правило, СУБД предоставляет программисту возможность анализа планов двух видов:

- оценочный план, создаваемый оптимизатором до выполнения запроса;
- реальный план, получаемый по итогам запроса.

Наличие двух планов объясняется возможностями оптимизатора корректировать план по ходу выполнения. Например, если результат выборки подзапроса с фильтром на основании плотности значений в колонке оценивался в несколько тысяч записей, а в реальности было выбрано менее сотни, то дальнейшие способы соединения могут быть пересмотрены.

Реальный план сохраняется в кэше СУБД. Он будет заново использован при следующем запуске запроса, увеличивая производительность в общем случае.

Инструментарий СУБД может иметь функцию графического представления плана выполнения запроса в виде дерева, корнем которого является результирующий набор данных. Если план невелик, то удобнее просматривать его в виде масштабируемого изображения, но когда SQL-запрос сложен, включает в себя десятки таблиц и подзапросов, найти нужный фрагмент становится трудно.

Функция выдачи плана в текстовом виде присутствует, как правило, всегда, но разбираться в ней труднее.

Возьмём для примера БД из главы «Постраничные выборки», предварительно заполненную данными, и выполним несложный запрос, подсчитывающий число продаж, количество товара в которых меньше среднего в 5 и более раз.

```
SELECT s.*
FROM sales s
     INNER JOIN (
       SELECT AVG(qty) / 5 AS avg_qty, id_product
       FROM sales s2
       GROUP BY id_product
     ) s3
       ON s.id_product = s3.id_product
WHERE s.qty < s3.avg_qty
```

Поскольку запрос достаточно простой, графическая интерпретация плана будет удобной для чтения.

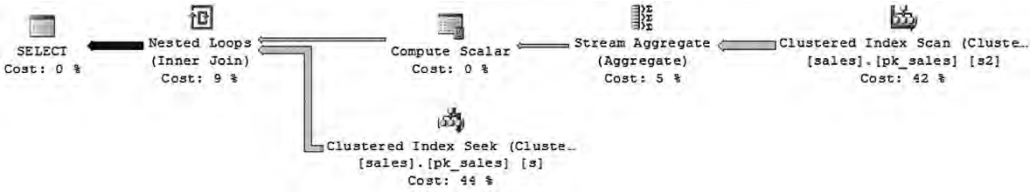

Рис.62. Схема плана выполнения запроса

Для получения текстового представления плана необходимо добавить перед запросом команду установки опции его показа (команда специфична для SQL Server).

```
SET SHOWPLAN_TEXT ON
GO
```

Сокращённый фрагмент текста плана выполнения того же запроса выглядит следующим образом.

```
StmtText
------------------------------------------------------------------
--
  |--Nested Loops(Inner Join, OUTER REFERENCES...
       |--Compute Scalar(DEFINE...
       |    |--Stream Aggregate(GROUP
BY:([s2].[id_product])...
       |         |--Clustered Index Scan(...sales
       |--Clustered Index Seek(OBJECT:(...sales
```

Из плана видно, что конечный результат получается в результате вложенного цикла (Nested Loops) по промежуточной выборке, подсчитывающей сгруппированное по товарам среднее количество их продаж (Compute Scalar, Stream Aggregate). В цикле на каждую запись промежуточной выборки осуществляется поиск по кластерному индексу (Clustered Index Seek). В свою очередь, промежуточная выборка сканирует всю таблицу sales, организованную в виде кластера (Clustered Index Scan).

Можно ли признать данный план эффективным? Для этого нужно знать, какие элементы плана потенциально могут быть узким местом.

Прежде всего, следует ориентироваться на оценку стоимости операций, выставляемую СУБД каждому элементу плана выполнения запроса. К сожалению, в текстовом виде у SQL Server эта информация отсутствует,

поэтому придётся прибегнуть к анализу графического представления или его «исходника» в виде XML.

```
SET SHOWPLAN_TEXT OFF
SET SHOWPLAN_XML ON
GO
```

Оптимизатор СУБД оценивает стоимость в процентах от общей. Соответственно, необходимо обратить внимание на максимальные значения.

Кроме стоимости, под подозрение попадают элементы следующих типов:

- table scan — последовательное сканирование таблицы, данные которой организованы в виде «кучи»;
- clustered index scan — сканирование таблицы, организованной в кластер;
- index scan — сканирование индекса;
- hash join — соединение с предварительным хэшированием.

В нашем примере наиболее дорогими этапами выполнения запроса является сканирование таблицы продаж (42 %) и поиск по кластерному индексу (44 %).

Из текста запроса видно, что предварительная выборка не содержит условий фильтрации, поэтому её полное сканирование для вычисления агрегата avg() является нормальным поведением и не может быть улучшено непосредственным образом.

Второй дорогостоящий элемент — поиск по кластерному индексу. На самом деле, поиск по кластеру является наиболее быстрым способом нахождения записи. Но в нашем случае соединение производится по единственной колонке `id_product`, входящий в составной кластерный индекс. Зная, что оптимизатор использует метод вложенного цикла, можно попытаться улучшить производительность, создав дополнительный индекс по колонкам `id_product` и `qty`.

Вначале замеряем время выполнения запроса в текущих условиях. Для этого в опциях SQL-консоли (Query options) нужно включить «SET STATISTIC TIME» и «SET STATISTIC IO».

Перед выполнением запроса рекомендуется сбросить кэш планов и данных, чтобы не вносить в сравнения возможные разночтения.

```
DBCC DROPCLEANBUFFERS
DBCC FREEPROCCACHE
```

Для существующего запроса получаем результат:

```
Table 'sales'. Scan count 1001, logical reads 110204,
physical reads 0, read-ahead reads 0, lob logical reads 0,
lob physical reads 0, lob read-ahead reads 0.
 SQL Server Execution Times:
   CPU time = 4236 ms,  elapsed time = 14858 ms.
```

Теперь проверим первую гипотезу, строим дополнительный индекс со всеми опциями по умолчанию.

```
CREATE INDEX ix1_sales ON dbo.sales (id_product, qty)
```

Перезапускаем запрос. Во-первых, можно увидеть, что план немного изменился. Вместо Clustered Index Seek, теперь используется обычный Index Seek (Nonclustered), который в общем случае медленнее. Однако его стоимость по указанной выше причине (составной кластерный индекс) оказывается в разы ниже и падает до 12 %. Теперь наиболее дорогим элементом остаётся сканирование таблицы — целых 67 %.

Замер времени и ввода/вывода показывает небольшое улучшение.

```
Table 'sales'. Scan count 1001, logical reads 35549, physical
reads 0, read-ahead reads 1, lob logical reads 0, lob
physical reads 0, lob read-ahead reads 0.
 SQL Server Execution Times:
   CPU time = 3244 ms,  elapsed time = 13113 ms.
```

Стоит ли радоваться улучшению? Не всегда. Надо понимать, что за полученный выигрыш придётся платить: за счёт нового индекса размер БД увеличился, а скорость вставки в таблицу снизилась. Локальное улучшение потенциально может ухудшить характеристики других запросов.

Избежать такой неоднозначной ситуации в случае достаточно сложной системы, разрабатываемой коллективом, можно двумя основными

способами, не исключающими, а, скорее, взаимодополняющими друг друга:

- привлекать к работе администратора БД, который имеет общее видение работы всех приложений с СУБД и может на экспертном уровне оценить последствия добавлений новых индексов и других изменений;
- разработать систему автоматизированных тестов, прогон которых показывает возможную регрессию на отдельных операциях.

Поиск узких мест

Даже если автоматизированная информационная система находится в эксплуатации продолжительное время, в один не самый прекрасный день может наступить момент, когда пользователи начнут жаловаться на увеличение времени отклика. У системы, переданной в опытную эксплуатацию, вероятность такого поворота дел ещё выше, особенно при недостаточном внимании к нагрузочным тестам (о них в следующей главе).

Как найти узкое место?

Прежде всего, следует определить, является ли таким узким местом собственно база данных, являющаяся конечным пунктом назначения запросов пользователей. Для такой диагностики промышленные СУБД обладают более или менее развитыми средствами мониторинга и трассировки.

В идеальном случае задержку можно стабильно воспроизвести. Тогда средства трассировки оказываются наиболее подходящими задаче. Весьма удобным инструментом является SQL Server Profiler, позволяющий в реальном масштабе времени наблюдать и регистрировать запросы, отфильтрованные по самым разнообразным критериям, например, по названию приложения или имени хост-машины.

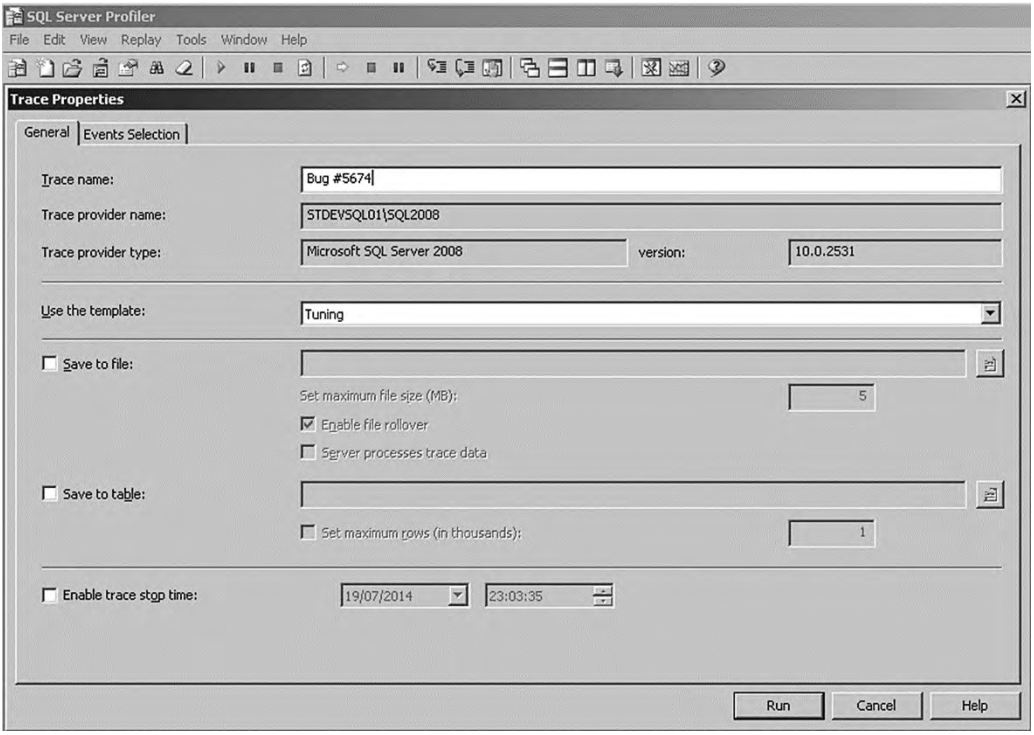

Рис.63. Установка опций запуска трассировки

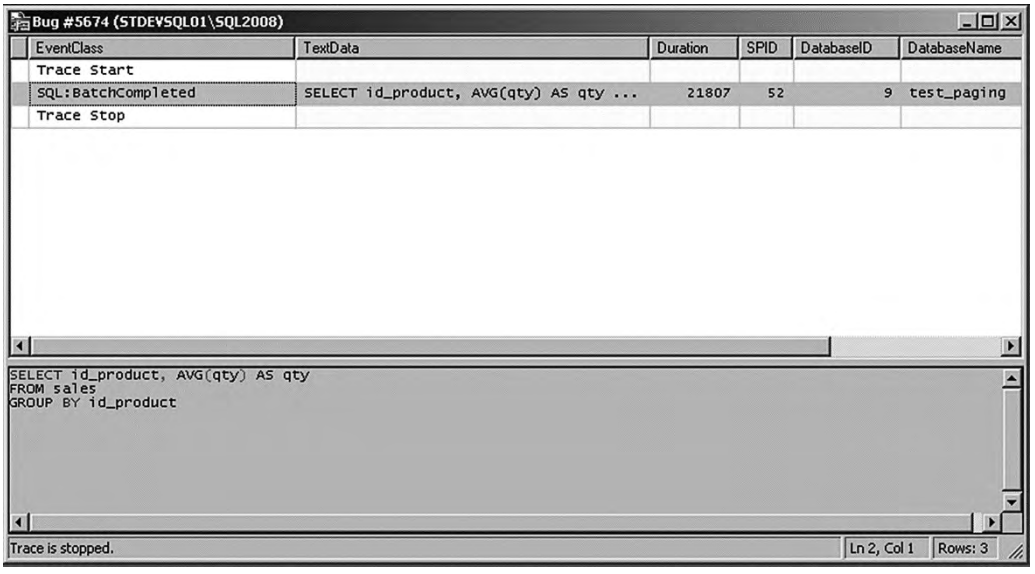

Рис.64. Результат трассировки средствами SQL Server Profiler

Итак, проблему можно воспроизвести в тестовой или эксплуатационной среде, запускаем утилиту трассировки с соответствующими настройками, на рабочем месте пользователя, на сервере пакетной обработки или ещё где-то на фронтальном уровне запускаем процесс и смотрим результат.

Из результата трассировки можно непосредственно определить время выполнения запроса или их серии, вызывающие нарекания по быстродействию. Если показанная цифра является гораздо меньшей, чем время отклика системы, то проблема кроется не в СУБД, а в вышестоящих слоях. Если же зафиксированное время соотносится с неприемлемо долгим ожиданием реакции системы на пользовательском уровне, то вам крупно повезло, с первой попытки найдено узкое место!

Теперь — дело техники, надо вычленить из трасс запросы, вызывающие проблемы, и провести их оптимизацию или переделку в режиме модульного тестирования и профилирования.

При условии, что запрос написан правильно с точки зрения программирования на SQL, наиболее частые причины его медленного выполнения таковы:

- недостаточно оперативной памяти на сервере или недостаточное количество памяти выделено для СУБД;
- не хватает статистики. Проверьте опции, касающиеся автоматического сбора и автоматического создания статистики по колонкам таблиц, если необходимо, пересоздайте её в ручном режиме;
- не хватает полезных индексов; проверьте в планах выполнения запросов использование индексов и в случае обнаружения прямого сканирования таблиц (table scan, cluster index scan) или хэшированных соединений (hash join) попробуйте добавить новые индексы сообразно условиям соединений или фильтрации запросов;
- не хватает полезных индексированных видов. Некоторые СУБД поддерживают индексированные виды, представляющие собой материализованное представление — слепок данных результата

некоторого запроса. Если в другом запросе частично или полностью может быть использован такой материализованный вид, то общее время выполнения снизится;

- проблемы физической архитектуры. Проверьте производительность дискового массива на характерных операциях чтения/записи (здесь может оказаться полезной утилита iometer). Проверьте, используется ли массив монопольно сервером СУБД или же доступ разделяется с другими серверами, возможно, виртуальными;
- отсутствует секционирование данных. При больших объёмах таблиц и запросах, затрагивающих лишь её небольшие фрагменты, следует подумать о секционировании.

К сожалению, такая ситуация является не самой частой на практике. Гораздо чаще задержки отклика системы являются спонтанными, воспроизвести их в одиночном режиме невозможно, а испытатель или приёмщик со стороны клиента описывает ситуацию как «в общем, система тормозит».

Не следует отчаиваться, нам помогут таблицы мониторинга, хранящие статистические данные о выполнении запросов в системе. Хорошая СУБД сама заботится о сборе информации, необходимой для диагностики своих проблем.

Общий список видов и табличных функций мониторинга в SQL Server называется Dynamic Management Views and Functions, по этим словам информацию можно найти в MSDN. Перечень содержит несколько десятков функций, поэтому ниже мы рассмотрим только несколько наиболее часто употребляемых в диагностике.

Если «система тормозит», то логично начать поиск с наиболее медленных и наиболее «прожорливых» запросов. Частично эти множества могут и пересекаться, но, в общем случае, медленный запрос может загружать только процессор, а «прожорливый», напротив, заставлять шуршать диски массива на полную катушку.

Для выявления кандидатов на оптимизацию нам понадобятся всего несколько запросов.

```
/* Статвыборка 1: запросы, использующие процессорное время */
SELECT TOP 100
  qt.text,
  qs.execution_count,
  qs.total_worker_time / 1000000 AS total_worker_time_sec,
  CASE qs.total_worker_time
    WHEN 0 THEN NULL
    ELSE round(
      (CAST(qs.execution_count AS float) /
            qs.total_worker_time * 1000000)
      , 3)
  END AS avg_queries_per_sec,
  qs.total_elapsed_time
FROM sys.dm_exec_query_stats qs
     CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) qt
     CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
ORDER BY total_worker_time DESC
```

В первую очередь, следует взглянуть на первую сотню запросов, наиболее использующих процессорное время (total_worker_time). Одновременно следует учитывать и число запусков этого запроса (execution_count) и среднюю производительность за секунду (avg_queries_per_sec). Например, если некоторый запрос находится в лидерах, но при этом он выполняется часто и его производительность не вызывает сомнений, то исключите его из списка подозреваемых.

Для прояснения картины та же статистическая выборка №1 может быть отсортирована по соответствующим колонкам или дополнительно отфильтрована, например «все запросы с производительностью ниже 100 в секунду и числом запусков не менее 1000».

Отсортировав ту же выборку по колонке execution_count, мы получим наиболее частые запросы. Здесь также следует обратить внимание на производительность.

Теперь настало время вытащить на свет наиболее «прожорливые» создания программистских рук. Прежде всего, замедления происходят при обращении к дисковой подсистеме.

```
/* Статвыборка 2: запросы, приводящие к обращениям к дискам
*/
SELECT TOP 100
  qt.text,
  qs.execution_count,
  qs.total_physical_reads,
  qs.total_physical_reads / qs.execution_count
    AS avg_physical_reads,
  qs.max_physical_reads,
  (qs.total_elapsed_time - qs.total_worker_time) / 1000
    AS total_non_cpu_time_msec
FROM sys.dm_exec_query_stats qs
     CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) qt
     CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
ORDER BY total_physical_reads DESC
```

Общее количество физических чтений (total_physical_reads), распределенное по числу запусков (execution_count) надо сопоставлять с их максимумом (max_physical_reads). Если две величины, avg_physical_reads и max_physical_reads, примерно равны на многократных запусках (запрос регулярный), это означает стабильное обращение к долговременной медленной памяти, являющееся, в свою очередь, проблемой нехватки выделенного для СУБД объёма ОЗУ.

В PostgreSQL также имеется набор видов/таблиц мониторинга. Вы можете найти их в документации в главе «Monitoring Database Activity. The Statistics Collector». Трассировка может осуществляться настройкой журналов уровня сервера, обратите внимание на соответствующую опцию `log_statement` в файле конфигурации `postgresql.conf`.

В Firebird, начиная с версии 2.5, были введены виды и таблицы мониторинга, имеющие префикс «MON$». Кроме того, добавлена утилита командной строки для трассировки запросов. С помощью этих видов и функций вы сможете произвести аналогичный анализ.

Разумеется, некоторая часть запросов может просочиться через статистическое сито, но основную их массу удаётся отловить, и тогда оставшиеся начинают проявлять себя более предсказуемо и стабильно, сводя поиск к трассировке, описанной в первом случае.

Удачной охоты!

Основы нагрузочного тестирования

Если в предыдущей главе рассматривались способы обнаружения узких мест в приложении, связанных с СУБД, то цель нагрузочного тестирования — выявить эти узкие места ещё на стадии системных тестов до передачи продукта заказчику на приёмку или в опытную эксплуатацию.

Инструменты и методы

Инструментов, реализующих нагрузочное тестирование для разных СУБД создано много, в том числе открытых. Поиск в Интернет по ключевым словам «*название_СУБД* load testing» или «stress testing» выдаст достаточно много ссылок. Ниже мы рассмотрим утилиту SQL Load Generator, ориентированную на SQL Server, но при этом максимально простую и бесплатную, с открытым исходным кодом.

Прежде чем приступить к тестированию, следует составить представление о типовых сценариях работы пользователей с системой. Из них будут вытекать, во-первых, типы используемых запросов, во-вторых, оценка числа активных соединений.

Для примера возьмём БД, использованную в тестах постраничной выборки (см. главу «Постраничные выборки»). Нам понадобится генератор псевдослучайных значений в диапазоне (0..1), создадим его как вид.

```
CREATE VIEW dbo.rand2 AS
SELECT rand(abs(convert(int, convert(varbinary, newid()))))
AS rand_value;
```

Пусть в системе будет 10 пользователей, выполняющих запросы типа

```
SELECT *
FROM sales
WHERE id_customer = N AND id_product = M
```

Здесь N и M — произвольным образом выбранные целочисленные идентификаторы, соответственно, клиента и товара. Из условий теста постраничной выборки, при заполнении соответствующих таблиц использовались значения 10000 для количества клиентов и 1000 для количества товаров. С применением генератора псевдослучайных чисел запрос примет следующий вид.

```
SELECT *
FROM sales
WHERE id_customer = (SELECT CAST((10000 * rand_value) AS int)
FROM rand2) AND
      id_product = (SELECT CAST((1000 * rand_value) AS int)
FROM rand2)
```

Одновременно, ещё 10 пользователей выполняют запросы вида

```
UPDATE sales
SET qty = X
WHERE id_customer = N AND id_product = M
```

Здесь X — случайное значение количества товара в диапазоне (0..1000).

```
UPDATE sales
SET qty = (SELECT CAST((1000 * rand_value) AS int) FROM
rand2)
WHERE id_customer = (SELECT CAST((10000 * rand_value) AS int)
FROM rand2) AND
      id_product =(SELECT CAST((1000 * rand_value) AS int)
FROM rand2)
```

Определим в опциях параметры соединения утилиты с СУБД Microsoft SQL Server.

Теперь добавляем новый запрос (кнопка Add Query) и заносим туда первый из подготовленных выше. Для параметра числа одновременных активных соединений определяем значение «10» согласно сценарию. Аналогично поступаем со вторым запросом на обновление.

Осталось запустить оба сценария на выполнение и подождать некоторое время, например, до выполнения первой тысячи-двух запросов на модификацию данных.

После остановки запросов смотрим статистику выполнения по уже известной методике.

```
SELECT TOP 100
  qt.TEXT,
  qs.execution_count,
  qs.total_worker_time / 1000000 AS total_worker_time_sec,
  CASE qs.total_worker_time
    WHEN 0 THEN NULL
    ELSE
     CAST(qs.execution_count AS float) /
```

```
          qs.total_worker_time * 1000000
  END AS queries_per_sec
FROM sys.dm_exec_query_stats qs
CROSS APPLY sys.dm_exec_sql_text(qs.sql_handle) qt
CROSS APPLY sys.dm_exec_query_plan(qs.plan_handle) qp
WHERE qt.text LIKE '%rand_value%'
ORDER BY execution_count DESC
```

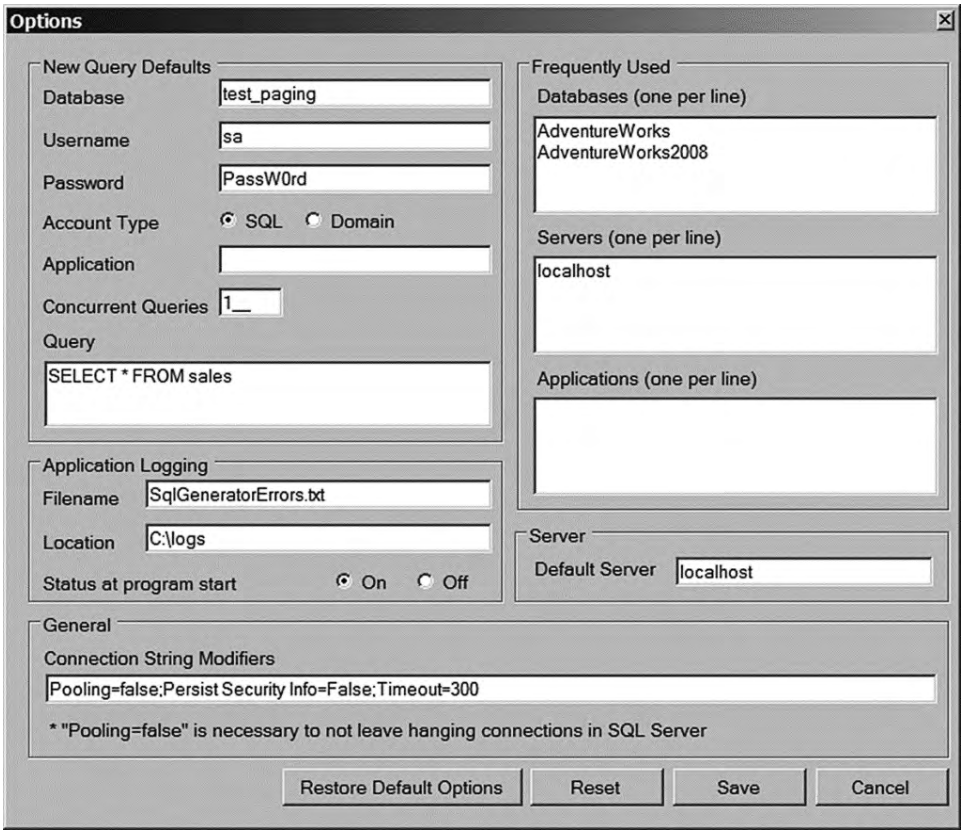

Рис.65. Установка опций соединения

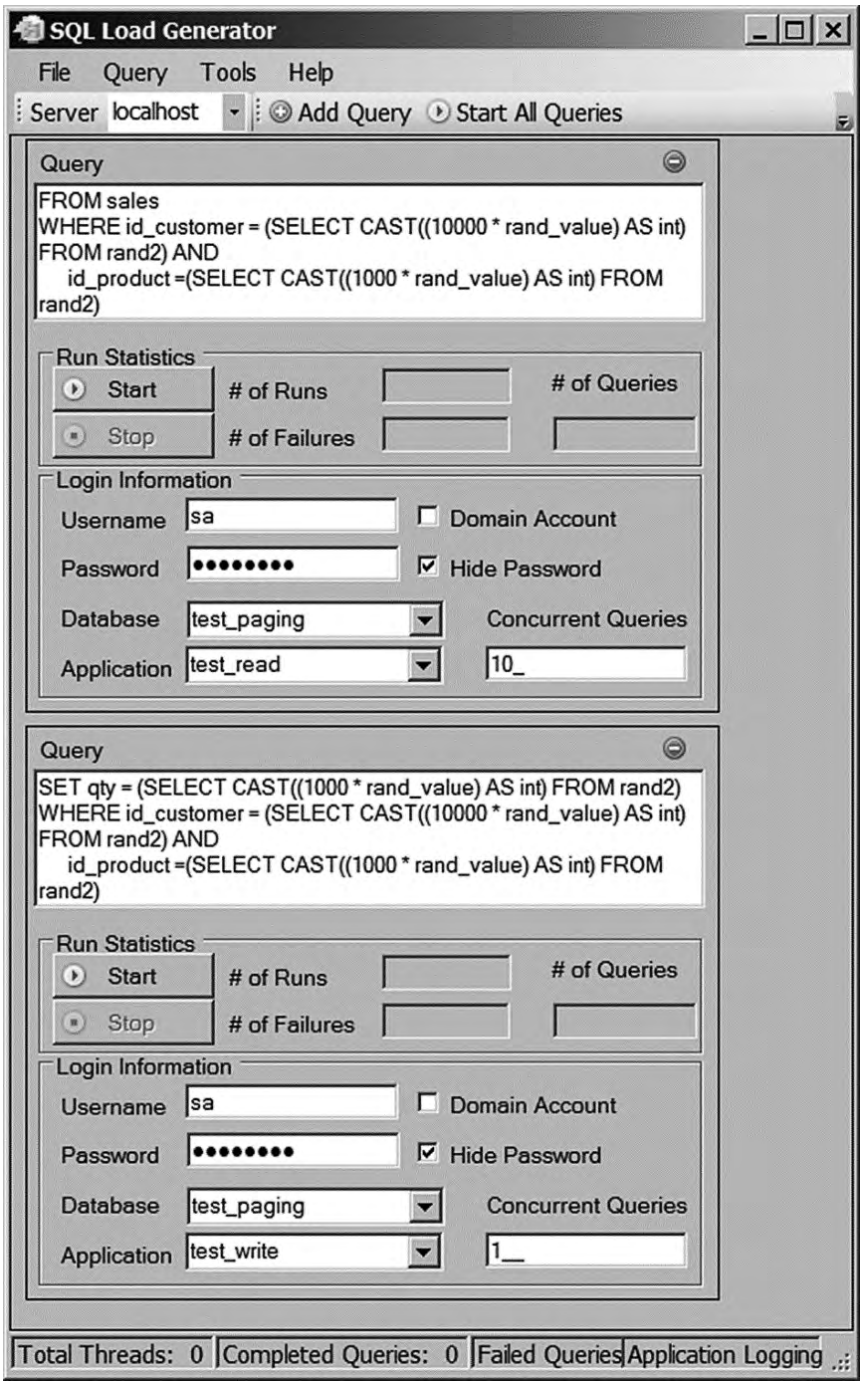

Рис.66. Создание тестовых запросов

Для запросов на чтение и запись получаем следующие показатели.

Запрос	Выполнено, раз	Потрачено времени, сек	Транзакций в секунду
Чтение (запрос №1)	8767	8	1083
Запись (запрос №2)	4286	7	548

Если полученные значения по количеству транзакций в секунду вас устраивают, то данный сценарий использования системы можно считать проверенным. Вас можно поздравить.

А если нет? Тогда вас тоже можно поздравить: узкое место выявлено на раннем этапе разработки, когда стоимость его ликвидации ещё относительно невысока. По крайней мере, заказчик не столкнётся с этой проблемой на собственной шкуре.

Учёт степени параллелизма

В не то чтобы старые и не столь уж однозначно добрые времена один физический процессор на рабочем месте программиста или на сервере имел одно ядро. Поэтому данный вопрос не являлся таким значимым, как сейчас, когда количество ядер на одном «камне» позволяет СУБД распараллеливать выполнение запросов уже на самых первых этапах процесса разработки программ.

Допустим, вы отлаживаете SQL-запрос, получив на рабочем месте или на сервере разработки удовлетворительное время. Но попав на тестовый сервер да ещё и под нагрузку ваш запрос начинает показывать на порядок худшее время. В чем проблема?

При отладке запроса СУБД, как правило, позволяет выводить сопутствующую статистическую информацию и план его выполнения. Одним из ключевых показателей является время, затраченное непосредственно процессором. В SQL Server такая информация выводится в виде «CPU time = 123 ms», показывая, сколько времени запрос выполнялся процессором. В других СУБД также имеются возможности по выводу аналогичной статистики (EXPLAIN в PostgreSQL, SET PLAN в Firebird и т. д.).

Положим, SQL-запрос выводит общее количество заказов, количество товаров в которых превышает среднее количество во всех заказах.

```
/* Очистка кеша процедур и данных */
DBCC FREEPROCCACHE;
DBCC DROPCLEANBUFFERS;
/* Вывод статистики */
SET STATISTICS TIME ON;
SET STATISTICS IO ON;
/* Тестовый запрос */
SELECT COUNT(1)
FROM
  (SELECT id_product, AVG(qty) AS qty
   FROM sales
   GROUP BY id_product
   ) t1
WHERE t1.qty > (
  SELECT AVG(qty)
  FROM (SELECT id_product, AVG(qty) AS qty
        FROM sales
        GROUP BY id_product) t2)
```

При очищенных буферах и включённых опциях SET STATISTICS TIME и SET STATISTICS IO сопровождающие выдачу результата сообщения на MS SQL Server будет выглядеть следующим образом.

```
(1 row(s) affected)
Table 'sales'. Scan count 2, logical reads 106820, physical
reads 12, read-ahead reads 50383, lob logical reads 0, lob
physical reads 0, lob read-ahead reads 0.

 SQL Server Execution Times:
   CPU time = 32627 ms,  elapsed time = 47882 ms.
```

Запустим запрос ещё раз, но без очистки кеша, т. е. в «горячем» режиме.

```
(1 row(s) affected)
Table 'sales'. Scan count 2, logical reads 106820, physical
reads 0, read-ahead reads 0, lob logical reads 0, lob
physical reads 0, lob read-ahead reads 0.

 SQL Server Execution Times:
   CPU time = 30323 ms,  elapsed time = 35714 ms.
```

Статистика хорошо показывает, что запуск в холодном и горячем режимах ожидаемо влияет на общее время выполнения запроса (elapsed time), но в

гораздо меньшей степени — на процессорное время (CPU time). Это объясняется наличием большого числа операций физического и упреждающего чтения, сопровождающегося обменом с долговременной памятью (дисками).

Зная, например, что данный пример был выполнен на виртуальной машине с единственным одноядерным процессором, можно взять полученный результат за основу и предполагать, что с увеличением числа задействованных ядер и процессоров эта составляющая времени выполнения запроса будет пропорционально снижаться.

С той же целью для более точной оценки можно задействовать в запросе специфичные для СУБД опции, указывающие максимальную степень параллелизма, установив это значение в единицу. Например, в SQL Server этому служить опция MAXDOP.

```
/* запрос будет выполняться на одном процессоре/ядре */
SELECT id_product, AVG(qty) AS qty
FROM sales
GROUP BY id_product
OPTION(MAXDOP 1)
```

Верно и обратное. Положим, на вашем однопроцессорном четырёхядерном рабочем месте, где установлена локальная версия СУБД, некоторый SQL-запрос выполняется за 500 миллисекунд, в том числе 300 миллисекунд занимает процессорное время. Значит на одном ядре запрос выполнялся бы примерно 1200 миллисекунд, в четыре раза дольше. Если на тестовом сервере установлены два процессора с четырьмя ядрами, то под имитацией нагрузки 15-20-ти пользователей ваш запрос начнёт выполняться в районе 2,5-3 секунд и выше. Время выросло почти на порядок!

Таким образом, при оценке соответствия производительности запросов требуемому времени отклика в режиме модульного тестирования и профилирования следует принять во внимание особенности конфигурации СУБД-серверов соответствующей среды, прежде всего, количество физических или виртуальных процессоров и их ядер.

SQL Server и MongoDB на простом тесте

Описанное ниже испытание было проведено в рамках одного из предпроектных обследований. Хотя выбор SQL Server был фактически предопределён, целью тестирования было скорее понять, какие преимущества и проблемы могло бы принести использование в системе MongoDB.

Задача, поставленная в условиях теста, не является сильной стороной документ-ориентированных моделей, поскольку атрибутика данных достаточно жёсткая. Тем не менее, испытание показывает основные отличия при работе с СУБД разных моделей.

На сайте MongoDB была загружена последняя стабильная версия 2.0.2 для 64-разрядной Windows. В качестве альтернативы MongoDB выступал MS SQL Server 2008 R2 Developer Edition, тоже 64-разрядный. Если у вас такого нет, подойдёт и бесплатная редакция SQL Server Express 2005 и выше, которую также можно загрузить с сайта Microsoft. Компьютер для обоих тестов использовался относительно слабый, уровня обычной рабочей станции, под управлением ОС Windows 7, двухядерный Intel 2,6 ГГц с одним достаточно «экономичным» диском 5400 оборотов/мин, 300 Гб, и шестью гигабайтами памяти. Поскольку целью является сравнение некоторых характеристики на одной и той же конфигурации, то аппаратная мощность компьютера не имеет значения.

Тест вставки записей

Тест интенсивной вставки данных от множества датчиков ограничен 10 миллионами записей. Реальные объёмы на порядки выше, но и такое количество более чем достаточно для выявления потенциальных проблем.

Данные вставляются в коллекцию MongoDB и таблицу SQL Server, соответственно, имеющие одинаковую структуру.

К сожалению, MongoDB не поддерживает транзакционность на уровне группы операторов. Но в данном случае это должно дать выигрыш в производительности.

```
function pad(n) { return n < 10 ? '0' + n : n };
function ISODateString(d) {
```

```
  return d.getUTCFullYear() + '-'
      + pad(d.getUTCMonth() + 1) +'-'
      + pad(d.getUTCDate()) + 'T'
      + pad(d.getUTCHours()) + ':'
      + pad(d.getUTCMinutes()) + ':'
      + pad(d.getUTCSeconds());
};

print("Starting fill database: " + Date());
maxLines = 1000;
progressStep = maxLines / 100;
devicesCount = maxLines / 5;
for (var i = 1; i <= maxLines; i++) {
  db.measuresData.insert(
  {
    "measureId"  : i,
    "deviceId"   : Math.floor(Math.random() * devicesCount +
1),
    "stateId"    : Math.floor(Math.random() * 2 + 1),
    "groupId"    : Math.floor(Math.random() * 100 + 1),
    "measureDate": new Date(),
    "intVal"     : Math.floor(Math.random() * 2000000 -
1000000),
    "floatVal"   : Math.random() * 2000000 - 1000000
  });
  if (i % progressStep == 0) {
    print(i + ";" + ISODateString(new Date()));
  }
};
print("Finished at: " + Date());
```

Для SQL-скрипта из главы «Загрузка данных» вы уже знаете, что приложение должно сначала накопить массив строк, а потом вставить их в таблицу в контексте одной транзакции. В тесте выбран размер пакета из 10 записей. Не забудьте изменить пути физического размещения файлов в соответствии с вашей конфигурацией.

```
SET NOCOUNT ON;

CREATE DATABASE testvol
ON PRIMARY (
  NAME = N'testvol',
  FILENAME = N'D:\MSSQL\DATA\testvol.mdf',
  SIZE = 200MB,
  MAXSIZE = UNLIMITED,
```

```
  FILEGROWTH = 200MB
)
LOG ON (
  NAME = N'testvol_log',
  FILENAME = N'D:\MSSQL\DATA\testvol_log.ldf',
  SIZE = 100MB,
  MAXSIZE = UNLIMITED,
  FILEGROWTH = 100MB
)
GO

ALTER DATABASE [testvol] SET RECOVERY SIMPLE
GO

USE testvol
GO
CREATE TABLE dbo.measuresData (
  measureId   int NOT NULL PRIMARY KEY CLUSTERED,
  deviceId    int NOT NULL,
  stateId     int NOT NULL,
  groupId     int NOT NULL,
  measureDate datetime NOT NULL,
  intVal      int NOT NULL,
  floatVal    float NOT NULL
)
GO

PRINT 'Starting fill database: ' + convert(nvarchar(32),
getdate(), 120);
DECLARE @maxLines int = 10000000;
DECLARE @progressStep int = @maxLines / 100;
DECLARE @devicesCount int = @maxLines / 5;
DECLARE @batchSize int = 10;
DECLARE @i int = 1;
BEGIN TRANSACTION;
WHILE @i <= @maxLines BEGIN
  INSERT INTO dbo.measuresData (
    measureId,
    deviceId,
    stateId,
    groupId,
    measureDate,
    intVal,
    floatVal
  )
```

```
  VALUES (
    @i,
    floor(rand() * @devicesCount + 1),
    floor(rand() * 2 + 1),
    floor(rand() * 100 + 1),
    getdate(),
    floor(rand() * 2000000 - 1000000),
    rand() * 2000000 - 1000000
  );

  if @i % @progressStep = 0 BEGIN
    PRINT convert(nvarchar(16), @i) + ';' +
          convert(nvarchar(32), getdate(), 120);
  END
  if @i % @batchSize = 0 BEGIN
    COMMIT TRANSACTION;
    BEGIN TRANSACTION;
  END
  SET @i = @i + 1;
END
COMMIT TRANSACTION;
PRINT 'Finished at: ' + convert(nvarchar(32), getdate(),
120);
GO
```

По результатам теста получился следующий график, показывающий, что несмотря на отсутствие транзакций, вставка пачек по 10 записей в таблицу SQL Server производительнее вставки соответствующих документов в коллекцию MongoDB.

Оценка размера базы данных также говорит в пользу SQL Server. Коллекция из 10 миллионов документов заняла 3,95 гигабайт, тогда как база данных SQL Server всего 0,5 гигабайт (без использования компрессии), то есть почти в 8 раз меньше. Избыточность требуемого размера хранения объясняется внутренним форматом JSON, используемом в MongoDB.

В отношении потребляемой оперативной памяти, MongoDB «отъела» практически всю свободную, заняв 4 гигабайта против всего 1 гигабайта у SQL Server при выставленном ему лимите в 3 гигабайта.

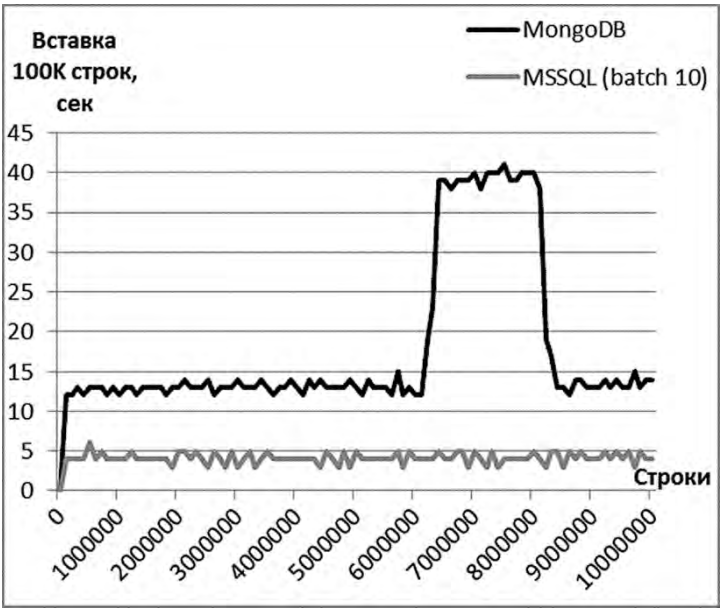

Рис.67. Время вставки 100 тысяч записей (пачками по 10) и документов

К сожалению, ограничить объем используемой оперативной памяти у MongoDB на данном этапе её развития было нельзя. Запрос на изменения висел больше года на сайте, не встретив у разработчиков понимания. В общем случае решение состоит в создании виртуального сервера для СУБД, которой отдаётся вся имеющаяся память.

Запросы и хронометраж

Запросы в CRUD-логике тестировать не имело большого смысла, потому что сразу после вставки таблица должна была служить для отчётности и оперативной аналитики. То есть, запросы предполагаются ещё не настоящие тяжёлые аналитические, но они могут вызывать сканирование всей таблицы.

Для тестов приходится делать перезапуск процесса `mongod` для очистки памяти. В SQL Server для аналогичного эффекта просто очищаем буфера и кэш соответствующими командами DBCC.

Первым впечатлением от языковых средств MongoDB было ощущение возврата куда-то в начало 1990-х годов: навигационный подход в стиле программ на Clipper (Harbour), с его бесконечными обходами таблиц в

циклах, явным выбором текущего индекса для поиска, наложениями фильтров, ручным суммированием и прочими давно забытыми в SQL-среде особенностями.

В MongoDb отсутствуют встроенные функции агрегации. Для нахождения сумм или средних величин нужно фактически написать свой аналог стандартных в SQL функций SUM() или AVG(), только на языке JavaScript, или использовать более общий метод MapReduce.

Справедливости ради, надо сказать, что начиная с версии SQL Server 2005, программисты могут писать свои нестандартные функции агрегации на C#. Но я посоветовал бы вначале хорошо подумать о необходимости этого шага.

В случае, если запрос с агрегацией возвращает более 10 тысяч документов, использование MapReduce становится обязательным, иначе система выдаёт ошибку. Таким образом, из встроенных функций остаётся только подсчёт числа элементов, в том числе уникальных. Но использовать подсчёт можно только в применении к целым коллекциям, то есть вне контекста запросов с группировками. Поэтому в общем случай программист должен писать рутинный типовой код на JavaScript и для функции агрегации COUNT().

Оказалось невозможным и отсортировать результаты запроса с группировкой. Разработчики на форуме поддержки предложили выбрать результат в приложение и сделайте сортировку в нём.

> *Q: How can I sort on the resultset of the group by operation?*
>
> *A: Group currently just returns an object, so you should be able to sort easily client side*

В результате код запросов с кратким и ясным синтаксисом декларативного языка SQL превращается в длинную «простыню» из императивных инструкций и параметров-деклараций. Разницу можно проследить на последующих примерах.

Запрос №1. Поиск минимального и максимального значения дат в таблице/коллекции.

SQL Server

```
SELECT MIN(measureDate), MAX(measureDate)
FROM dbo.measuresData
```

MongoDB

```
var minDate = new Date(1900,1,1,0,0,0,0);
var maxDate = new Date(2100,1,1,0,0,0,0);
db.measuresData.group(
{
  key: { },
  reduce: function(obj, prev)
  {
    if (obj.measureDate.getTime() < prev.minMsec)
      prev.minMsec = obj.measureDate.getTime();
    else if (obj.measureDate.getTime() > prev.maxMsec)
      prev.maxMsec = obj.measureDate.getTime();
  },
  initial: { minMsec: maxDate.getTime(), maxMsec:
minDate.getTime() },
  finalize: function(out)
  {
    out.minMeasureDate = new Date(out.minMsec);
    out.maxMeasureDate = new Date(out.maxMsec);
  }
})
.forEach(printjson);
```

Вариант MongoDB использует представление дат в виде числа миллисекунд, отсчитываемых от 01/01/1970, поэтому прямое их сравнение в функции вызывало ошибку `TypeError: this["get" + UTC + "FullYear"] is not a function nofile_b:2`.

Опыт работы с навигационными СУБД пригодился и в MongoDB. Если построить индекс по элементу `measureDate`, то можно найти крайние значения относительно простым способом: отсортировать по индексу в порядке возрастания и взять первый элемент, повторив процедуру в порядке убывания значений.

```
// построение индекса
db.measuresData.ensureIndex({measureDate: 1});
// первый элемент
db.measuresData.find({}, {measureDate: 1})
  .sort({measureDate: 1}).limit(1);
```

```
// последний элемент
db.measuresData.find({}, {measureDate: 1})
  .sort({measureDate: -1}).limit(1);
```

Запрос №2. Подсчёт сумм целых и вещественных значений по состоянию и группе устройств. Запрос для MongoDB не выполняет сортировку.

SQL Server

```
SELECT SUM(intVal), SUM(floatVal), stateId, groupId
FROM dbo.measuresData
GROUP BY stateId, groupId
ORDER BY stateId, groupId
```

MongoDB

```
db.measuresData.group(
{
  key: { stateId: true, groupId: true },
  reduce: function(obj, prev)
  {
    prev.sumIntVal += obj.intVal;
    prev.sumFloatVal += obj.floatVal;
  },
  initial: { sumIntVal: 0, sumFloatVal: 0.0 }
})
.forEach(printjson);
```

Запрос №3. Подсчёт общего числа устройств и количества уникальных устройств по состоянию и их группе.

SQL Server

```
SELECT COUNT(deviceId), COUNT(DISTINCT deviceId), stateId,
groupId
FROM dbo.measuresData
GROUP BY stateId, groupId
ORDER BY stateId, groupId
```

MongoDB (не выполняет сортировку)

```
db.measuresData.group(
{
  key: { stateId: true, groupId: true },
  reduce: function(obj, prev)
  {
    prev.count++;
  },
  initial: { count: 0 },
```

```
  finalize: function(out)
  {
    out.distCount = db.measuresData
      .distinct("deviceId", {stateId: out.stateId, groupId:
out.groupId})
      .length;
  }
})
.forEach(printjson);
```

«Бортовой» журнал СУБД показывает примерно 35 секунд для функции `finalize` на каждый цикл вычисления `distinct` из примерно 200 строк. Такая производительность совершенно неприемлема, запрос будет крутиться в течение почти 2 часов. Прерываем выполнение и приступаем к оптимизации. Необходимо построить индекс по элементам `stateId` и `groupId`.

```
db.measuresData.ensureIndex({stateId: 1, groupId: 1})
```

Индекс строился 141 секунду, тогда как аналогичный в SQL Server — за 20 секунд. Размер БД возрастает до 4,2 гигабайта.

Перезапускаем запрос. Цикл вычисления `distinct` снижается до 300 миллисекунд, в 100 раз! В третьем перезапуске время цикла выросло до 1,2 секунды, потом начало снижаться до 300 миллисекунд. Общее время — более 15 минут! Возникают подозрения, что СУБД практически не использует кэш данных.

Запрос №4. Подсчёт сумм целых и вещественных значений по состоянию и группе устройств для заданного диапазона дат. В качестве диапазона выбрана одна минута примерно в середине интервала между минимальным и максимальным значениями дат. Так как распределение тестовых данных близко к равномерному, каждый диапазон включает в себя около 500 тысяч строк.

SQL Server

```
SELECT SUM(intVal), SUM(floatVal), stateId, groupId
FROM dbo.measuresData
WHERE measureDate BETWEEN '20120208 22:54' AND '20120208
22:55'
GROUP BY stateId, groupId
```

```
ORDER BY stateId, groupId
```

MongoDB (не выполняет сортировку)

```
var date1 = new Date(ISODate("2012-02-08T15:20:00.000Z"));
var date2 = new Date(ISODate("2012-02-08T15:21:00.000Z"));

db.measuresData.group(
{
  key: { stateId: true, groupId: true },
  cond: {measureDate: { $gte: date1, $lt: date2 } },
  reduce: function(obj, prev)
  {
    prev.sumIntVal += obj.intVal;
    prev.sumFloatVal += obj.floatVal;
  },
  initial: { sumIntVal: 0, sumFloatVal: 0.0 }
})
.forEach(printjson)
```

Запрос №5. Подсчёт общего числа устройств и количества уникальных устройств по состоянию и их группе для того же заданного диапазона дат.

SQL Server

```
SELECT COUNT(deviceId), COUNT(DISTINCT deviceId), stateId,
groupId
FROM dbo.measuresData
WHERE measureDate BETWEEN '20120208 22:54' AND '20120208
22:55'
GROUP BY stateId, groupId
ORDER BY stateId, groupId
```

MongoDB (не выполняет сортировку)

```
var date1 = new Date(ISODate("2012-02-08T15:20:00.000Z"));
var date2 = new Date(ISODate("2012-02-08T15:21:00.000Z"));

db.measuresData.group(
{
  key: { stateId: true, groupId: true },
  cond: {measureDate: { $gte: date1, $lt: date2 } },
  reduce: function(obj, prev)
  {
    prev.count++;
  },
  initial: { count: 0 },
  finalize: function(out)
```

```
  {
    out.distCount = db.measuresData
      .distinct("deviceId", {stateId: out.stateId, groupId:
out.groupId})
      .length;
  }
})
.forEach(printjson);
```

Проведённый хронометраж осуществлялся по каждому запросу в три последовательных запуска, первый из них — в «холодном» режиме.

Для MongoDB результаты без оптимизации выглядят неприемлемо долгими. После оптимизации, где она возможна, результаты можно оценить, как неудовлетворительные.

Со стороны SQL Server оптимизация сводится к построению кластерного индекса по колонке `measureDate`. Соответственно, имеющийся первичный ключу по `measureId` объявляется некластерным. В запросах №2 и №3 необходимо полное сканирование данных, поэтому оптимизация индексацией для SQL Server не производилась.

Первый запуск — так называемый «холодный», вторые и третьи запуски — «горячие», они хорошо иллюстрируют работу кэша СУБД.

Табл. 23. Хронометраж запросов

	SQL Server				**MongoDB**		
Запуски	1	2	3		1	2	3
Без оптимизации, время, сек							
Запрос №1	7	1	1		190	185	189
Запрос №2	8	3	3		>7200		
Запрос №3	26	20	20		345	341	349
Запрос №4	8	1	1		22	22	22
Запрос №5	15	10	12		141	141	141
После оптимизации, время, сек							
Запрос №1	0	0	0		0	0	0
Запрос №2	8	3	3		322	800	619
Запрос №3	не производилась						
Запрос №4	1	1	1		12	12	13
Запрос №5	7	7	7		85	84	85

Выводы

Несмотря на не вполне подходящие условия задачи для использования NoSQL-СУБД, можно сделать некоторые выводы в отношении использования. С другой стороны, делать скидку на «не вполне подходящие условия», значит прямо признать, что СУБД является специализированной и имеет узкий сегмент применения.

Из необходимости построения индекса в случае запроса №2 следует, что без оптимизации, требующего вмешательства администратора БД, время отклика простого adhoc-запроса становится неприемлемым. С точки зрения пользователя это означает, что «запрос не работает».

Недостатки:

- сложность программирования запросов, являющихся обычными в языке SQL;
- неудовлетворительное время отклика при выполнении запросов;
- избыточность физического хранения;
- проблемы явного ограничения использования оперативной памяти;
- неэффективное использование кэша данных и запросов.

Преимущества:

- гибкая структура данных, не требующая предварительного определения схемы.

Перечисленное даёт возможность утверждать, что выбор подобного решения должен быть обоснован и подтверждён сравнительными тестами с универсальными РСУБД, в том числе с использованием их встроенной поддержки неполно структурированных данных.

Тестовые и демонстрационные базы данных

Многие поставщики СУБД собственными силами и при помощи сообщества пользователей предоставляют для испытаний и ознакомления так называемые тестовые и демонстрационные базы данных.

В некоторых случаях, например, Firebird, небольшая демонстрационная база входит в дистрибутив установки.

Для Microsoft SQL Server в соответствующем разделе сайта сообщества разработчиков (http://msftdbprodsamples.codeplex.com) можно найти реляционные демобазы для транзакционных и аналитических приложений, а также многомерные базы для Analysis Services.

На странице документации сообщества пользователей PostgreSQL (http://wiki.postgresql.org/wiki/Sample_Databases) приводится множество ссылок на базы данных, которые можно загрузить и установить для проведения тестов.

Разработчики Oracle также предоставляют всем интересующимся примеры не только собственно баз данных, но и программную реализацию некоторых техник работы с продуктом, например, использования API для добычи данных (Data Mining). Подробнее см. http://docs.oracle.com/cd/E11882_01/install.112/e24501/toc.htm.

Заключение

Теперь, отложив книгу в сторону, можно немного поразмыслить. Насколько целостной была подаваемая информация? Удалось ли узнать что-то новое? Остались ли за рамками издания вопросы, которые хотелось бы рассмотреть? Станет ли чтение толстых монографий из списка литературы более осмысленным? Ваши отзывы могут быть учтены в новом дополненном переиздании.

Одной из целей книги было уменьшение числа ситуаций, когда программистам требуется помощь специалистов по СУБД. Потому что работа эксперта должна, в идеале, быть направлена на совместный синтез проектных решений, а не на ликвидацию последствий реализации непродуманных подходов, когда, зачастую, наилучшим выходом является уже полная переделка частей системы.

Я не опасаюсь, что, приумножив собственные знания в области СУБД, вы станете обходиться без помощи соответствующих специалистов, но искренне надеюсь, что будете вовлекать их в проект на более ранних этапах, предотвращая грядущие проблемы.

Ipsa scientia potestas est. Knowledge itself is power. Francis Bacon.

Знание — сила. Фрэнсис Бэкон.

Литература

1. Дейт К. Дж. Введение в системы баз данных, 8-е издание.: Пер. с англ. — М.: Издательский дом "Вильямс", 2005. — 1328 с.: ил.
2. Карпова Т. С. Базы данных: модели, разработка, реализация. — СПб.: Питер,. 2002. —304 с., ил.
3. Грабер М. Введение в SQL. — М.: Лори, 1996.
4. Ульман Дж. Основы систем баз данных. — М.: Финансы и статистика, 1983. — 334 с, ил.
5. Мартин Дж. Организация баз данных в вычислительных системах. — М.: Мир, 1978.— 616 с.
6. Пржиялковский В. В. Абстракции в проектировании баз данных — Журнал «СУБД» №1-2/1998
7. Когаловский М. Р. Абстракции и модели в системах баз данных. — Журнал «СУБД» №4-5/1998
8. Чен, Петер Пин-Шен. Модель «сущность-связь» - шаг к единому представлению о данных. — Журнал «СУБД» №3/1995.
9. Хендерсон К. Профессиональное руководство по SQL Server: хранимые процедуры; пер. с англ. — СПб.: Питер, 2005.
10. Хендерсон К. Профессиональное руководство по SQL Server: структура и реализация; пер. с англ. — М. :Издательский дом «Вильямс», 2006.
11. Мирошниченко Г. А. Реляционные базы данных: практические приёмы оптимальных решений. — СПб.: БХВ-Петербург, 2005. — 400 с.: ил.
12. Чёрч А. Введение в математическую логику.: Пер. с англ. — Т. 1. — М., 1960.
13. Тарасов С. Дефрагментация мозга. Софтостроение изнутри.— СПб.: Питер, 2013.
14. Усов А. Ключ или отмычка?—2001 г., статья на веб-сайте: http://alexus.ru/russian/articles/dbms/keys/index.htm
15. Тенцер А. Естественные ключи против искусственных ключей. — 1999 г., статья на веб-сайте: http://ibase.ru/devinfo/NaturalKeysVersusAtrificialKeysByTentser.html

16. Емельянов Н. Е. Введение в СУБД ИНЕС. — М.:Наука, 1988, 256 с. (Библиотечка программиста).
17. Кодд Э. Ф. Реляционная модель данных для больших совместно используемых банков данных. Перевод Communications of the ACM, Volume 13, Number 6, June, 1970.— Журнал «СУБД», №1/1995.
18. Stonebraker, Michael. Readings in Database Systems (2nd edition). San Mateo, Calif.: Morgan Kaufmann, 1994. Introduction to Chapter 1.
19. Codd, E.F. Is Your DBMS Really Relational?; Computerworld, October 14th, 1985.
20. Codd, E.F. Does Your DBMS Run By The Rules?; Computerworld, October 21st, 1985.
21. Codd, E.F. The Relational Model For Database Management Version 2. Reading, Mass.: Addison-Wesley, 1990.
22. ИНТУИТ. Модели и смыслы данных в Cache и Oracle. Лекция на веб-сайте дистанционного обучения (http://www.intuit.ru).
23. Raima Database Manager 11.0. SQL Language Guide. Официальное руководство (англ.яз) на веб-сайте производителя СУБД (http://www.raima.com).
24. Д. Кнут. Искусство программирования (в 3 томах). Издание 2. Пер. с англ. М.: Вильямс, 2001.
25. Н. Вирт. Алгоритмы + Структуры данных = Программы. М.: Мир, 1985.
26. Joe Celko, Trees and Hierarchies. Morgan-Kaufmann, 2004.
27. Гамма Э. и др. Приёмы объектно-ориентированного проектирования. Паттерны проектирования. Пер. с англ. СПб.: Питер, 2006.
28. Фаулер М. и др. Архитектура корпоративных программных приложений. Пер с англ. М.: Вильямс, 2006.
29. Тарасов С. Разработка ядра информационной системы. Часть 1. Журнал «Мир ПК» №7 2007 г.
30. Тарасов С. В., Бураков В. В. Способы реляционного моделирования иерархических структур данных. Журнал «Информационно-управляющие системы» №6 2013 г.

Тарасов Сергей Витальевич

СУБД для программиста

Базы данных изнутри

Ответственный за выпуск: В. Митин
Верстка: СОЛОН-Пресс
Обложка: СОЛОН-Пресс

ООО «СОЛОН-Пресс»
123001, г. Москва, а/я 82
Телефоны:
(499) 254-44-10, (499) 795-73-26
E-mail: avtor@solon-press.ru
www.solon-press.ru

По вопросам приобретения обращаться:
ООО «ПЛАНЕТА АЛЬЯНС»
Тел: (499) 782-38-89, **www.alians-kniga.ru**

По вопросам подписки на журнал «Ремонт & Сервис» обращаться:
ООО «СОЛОН-Пресс»
тел.: (499) 795-73-26, **e-mail: rem_serv@solon-press.ru**
www.remserv.ru

ООО «СОЛОН-Пресс»
115142, г. Москва, Кавказский бульвар, д. 50
Формат 70×100/16. Объем 20 п. л. Тираж 200 экз.

Заказ №