The Pragmatic Programmers

Антипаттерны SQL

Как избежать ловушек при работе с базами данных

Билл Карвин

под редакцией Жаклин Картер

SQL Antipatterns, Volume 1

Avoiding the Pitfalls of Database Programming

Bill Karwin

The Pragmatic Bookshelf

Raleigh, North Carolina

Антипаттерны SQL

Как избежать ловушек при работе с базами данных

Билл Карвин

Выпущено
при поддержке
КРОК

ПИТЕР®

Санкт-Петербург • Москва • Минск
2024

Билл Карвин

Антипаттерны SQL. Как избежать ловушек при работе с базами данных

Серия «Библиотека программиста»

Перевел с английского Е. Матвеев

ББК 32.973.233.02
УДК 004.65

Карвин Билл

К21 Антипаттерны SQL. Как избежать ловушек при работе с базами данных. — СПб.: Питер, 2024. — 368 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2178-6

Язык SQL необходим для работы со структурированными данными. Программисты, прекрасно знающие свой любимый язык (Java, Python или Go), не могут разбираться во всем, и часто не являются экспертами в SQL. Это приводит к появлению антипаттернов — решений, которые на первый взгляд кажутся правильными, но со временем создают все больше проблем.

Научитесь выявлять и обходить многие из этих распространенных ловушек! Проведите рефакторинг унаследованного кошмара и превратите его в жизнеспособную модель данных!

Примеры SQL-кода основаны на версии MySQL 8.0, но в тексте также упоминаются другие популярные РСУБД. В примерах кода используется Python 3.9+ или Ruby 2.7+.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ISBN 978-1680508987 англ.
ISBN 978-5-4461-2178-6

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 03.2024. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 12.02.24. Формат 70×100/16. Бумага офсетная. Усл. п. л. 29,670. Тираж 1000. Заказ 0000.

Краткое содержание

ЧАСТЬ III
АНТИПАТТЕРНЫ ЗАПРОСОВ

ЧАСТЬ IV
АНТИПАТТЕРНЫ РАЗРАБОТКИ ПРИЛОЖЕНИЙ

ЧАСТЬ V
ДОПОЛНЕНИЕ: МИНИ-АНТИПАТТЕРНЫ ВНЕШНИХ КЛЮЧЕЙ

Оглавление

ЧАСТЬ II
АНТИПАТТЕРНЫ ФИЗИЧЕСКОГО ПРОЕКТИРОВАНИЯ БАЗ ДАННЫХ

ЧАСТЬ III
АНТИПАТТЕРНЫ ЗАПРОСОВ

ЧАСТЬ V
ДОПОЛНЕНИЕ: МИНИ-АНТИПАТТЕРНЫ ВНЕШНИХ КЛЮЧЕЙ

Отзывы о книге «Антипаттерны SQL»

Неповторимая, превосходная книга, посвященная тем вопросам программирования баз данных, которые часто упускают из виду. На понятных примерах и проверенных рекомендациях читатель познакомится с миром антипаттернов, часть из которых способна серьезно влиять на производительность базы данных и даже на процессы разработки. Отличный источник информации для разработчиков всех уровней квалификации.

- *Шломи Ноах* (*Shlomi Noach*)
  Инженер баз данных, PlanetScale

Обязательно прочитайте эту книгу, если вы пишете на SQL. Доступно и в то же время подробно. Билл дает много полезной информации и описывает решения, которые помогут вам значительно улучшить навыки работы с SQL.

- *Дэниел Нихтер* (*Daniel Nichter*)
  Администратор баз данных, автор книги *Efficient MySQL Performance*, Block Inc.

Жаль, что когда я был новичком, у меня не было этой книги. В ней описано столько антипаттернов, что теперь, вспоминая, сколько я в свое время наделал ошибок, я могу только разводить руками. Если бы я знал это раньше!

- *Сэмюэл Маллен* (*Samuel Mullen*)
  Старший менеджер отдела разработки, ActiveProspect

Книга под завязку набита полезными сведениями. Как новички, так и люди с опытом наверняка узнают из нее что-нибудь новое. Я, например, узнал немало.

- *Стивен Грим* (*Steven Grimm*)
  Техлид, Terraformation

Отличная книга для начинающих разработчиков SQL, полезный справочник по встречающимся антипаттернам. Мне особенно нравятся главы «Не вижу зла» и «Дипломатический иммунитет». Они идеально подходят для тех, кто пытается улучшить работу своей команды инженеров.

- *Макс Тилфорд* (*Max Tilford*)
  Сеньор-разработчик, Firstup

«Антипаттерны SQL» Билла Карвина — не только одна из лучших книг о базах данных, которую я прочитал. Это еще и определенно один из лучших образцов технической литературы, которые мне попадались. Я трижды прочел ее от корки до корки, и то, что я из нее узнал, помогает мне каждый день.

➤ *Пим Бруверс* (*Pim Brouwers*)
Старший архитектор программного обеспечения,
Ассоциация игроков Национальной хоккейной лиги

Билл Карвин проделал отличную работу по выявлению распространенных антипаттернов в области баз данных. Он дает полезные советы о том, как обнаружить эти паттерны и как от них избавиться. Эту книгу стоит прочитать каждому, кто занимается проектированием баз данных или запросов, — она поможет применять в работе лучшие практики.

➤ *Алекс Острем* (*Alex Ostrem*)
Инженер по надежности баз данных и разработке
программного обеспечения, Etsy

Даже для читателя, который не может похвастаться большим опытом в программировании, книга «Антипаттерны SQL» станет отличным началом знакомства с темой. Она объясняет, почему и отчего некоторые программные решения не работают как задумано, и содержит понятный разбор и конкретные примеры более эффективных решений.

➤ *Дженнифер Песек* (*Jennifer Pesek*)
Библиограф-консультант

Посвящаю своей жене Джен,
моей самой надежной поддержке и опоре

Благодарности

В том, что я узнал все, что было необходимо знать для написания этой книги, заслуга многих людей.

Мои родители распознали мою страсть к compute science, когда я был еще подростком, и серьезно потратились, чтобы купить мне персональный компьютер, хотя эти штуки еще были редкостью. Позже благодаря моим родителям и дедушке я получил образование в Калифорнийском университете.

У меня было много преподавателей, которые знакомили меня с теорией и практикой computer science, но я хочу особо поблагодарить двоих: доктора Кевина Керпласа (Dr. Kevin Karplus) и доктора Дэна Скрипчера (Dr. Dan Scripture). Они разработали учебный курс для технических писателей; этот курс показал мне всю важность практики составления документации в области, требующей внимания для изложения сложных идей.

Меня вдохновляли многие руководители и коллеги. Кейт Рейнольдс (Keith Reynolds) поручил мне первые проекты на языке C, которые научили меня, что программирование — это по большей части работа с чужим кодом. Дэвид Бреденберг (David Bredenberg) на собственном примере показал, что не стоит жалеть времени на то, чтобы помогать другим разработчикам. А благодаря коучингу Реа Бэррона (Rhea Barron) я повысил свои навыки написания технических текстов до нового уровня.

Я благодарен научным редакторам, которые любезно пожертвовали свое время на работу над вторым изданием книги: Рональду Брэдфорду (Ronald Bradford), Жану-Франсуа Ганю (Jean-François Gagné), Стивену Гримму (Steven Grimm), Сэмюэлу Маллену, Алексу Остраму, Дженнифер Песек, Максу Тилфорду и Пиму ван дер Валу. Их отзывы сделали эту книгу лучше. Спасибо вам!

Я также благодарю тех многих людей, кто оставил положительные рецензии и поделился своим мнением после выхода первого издания книги.

Я бесконечно благодарен своей жене Джен Дуайер (Jan Dwyer) — состоявшемуся писателю и разработчику баз данных. Она была первой, кто оценивал черновики обоих изданий этой книги. Ее вклад и мнение были бесценны, и без ее постоянной помощи, моральной поддержки и вдохновения я бы не справился.

Введение

Эта книга посвящена SQL — популярному языку программирования для работы с данными. А точнее, она посвящена ошибкам при использовании SQL.

Ошибаются все, но эксперты стараются учиться на ошибках, превращая их в возможности для совершенствования своих навыков. Если вы изучите самые распространенные ошибки, сделанные другими разработчиками, и узнаете, как их исправить, вы сможете стать лучше как профессионал.

Моя первое знакомство с SQL обернулось отказом от работы. Я только что окончил колледж, и ко мне обратился менеджер, который работал в университете и знал меня по летней практике. Он планировал запустить собственный стартап, который занимался бы разработкой ПО, и ему требовалась система управления базами данных, работающая на разных платформах UNIX на основе shell-скриптов. Ему нужен был программист вроде меня, который написал бы код для распознавания и выполнения ограниченного подмножества языка SQL.

«Нам не нужна полная поддержка языка — это слишком сложно. Только одной инструкции SQL — `SELECT`».

В колледже SQL не преподавали. Но я занимался разработкой полноценных приложений shell-скриптов и немного знал о парсерах и языках предметных областей. И я склонялся к тому, чтобы принять предложение. Что сложного в парсинге одной инструкции такого специализированного языка, как SQL?

Но начав читать о SQL, я сразу заметил, что этот язык отличается от всех других, на которых я прежде работал. Назвать `SELECT` «всего лишь одной инструкцией» этого языка — все равно что назвать двигатель «всего лишь одной деталью» автомобиля. Оба утверждения справедливы, но они совершенно не учитывают сложность и глубину предмета. Я понял, что для поддержки выполнения одной инструкции SQL мне пришлось бы разработать полнофункциональную систему управления реляционными базами данных и механизм запросов. Было очевидно, что на это даже у опытного разработчика ушло бы несколько лет, а я еще был слишком зелен, чтобы браться за такой проект самостоятельно.

Я отказался от предложения запрограммировать парсер SQL и ядро РСУБД в shell-скриптах. Руководитель недооценил масштаб проекта — возможно, он просто не понимал, что делает РСУБД.

Мой первый опыт использования SQL не отличался от опыта большинства. Многие разработчики изучают SQL самостоятельно; часто из-за того, что начинают работать над проектом, для которого он необходим. Обычно разработчики — как простые энтузиасты, так и профессионалы или признанные теоретики с докторской степенью — начинают использовать SQL без специальной подготовки. Это приводит к тому, что многие типичные ошибки совершаются снова и снова.

О втором издании

С момента выхода первого издания книги я успел поработать консультантом по SQL, преподавателем, разработчиком и администратором базы данных. Я побывал в десятках компаний в самых разных областях бизнеса, и все они использовали SQL. Я общался с другими опытными разработчиками и администраторами баз данных на конференциях и встречах, слушал рассказы об их успехах и неудачах.

Все программисты работают с данными независимо от того, какой язык или систему они используют, а SQL остается самым часто используемым языком для работы с данными. Так как отрасль разработки в целом продолжает расширяться, количество специалистов, применяющих SQL, постоянно растет, хотя альтернативные технологии баз данных также набирают популярность.

Во второе издание книги добавлены последние наблюдения, касающиеся распространенных ошибок SQL и разработки приложений на основе данных. Я учел всю обратную связь, полученную в отношении первого издания. Также были обновлены ссылки на актуальные сайты и новейшую информацию в Сети.

К существующим главам был добавлен целый ряд новых «мини-антипаттернов». В этих разделах кратко рассматриваются новые типы ошибок, а также описываются простые и быстрые решения, позволяющие их избежать.

Были также обновлены примеры кода, чтобы соответствовать последним версиям MySQL и Python — самым популярным современным языкам базы данных с открытым исходным кодом и динамического программирования.

Для кого эта книга

Книга «Антипаттерны SQL» подойдет каждому, кто работает с SQL, то есть практически всем, от новичков до матерых профессионалов. Темы, рассматриваемые в книге, будут полезны разработчикам любого уровня.

Возможно, вы уже изучали синтаксис SQL. Вы знаете все секции инструкции `SELECT` и можете приступать к работе. Постепенно вы будете повышать свои

навыки в SQL, читая код, книги и блоги. Но освоите ли вы тем самым лучшие практики или только загоните себя в угол?

В книге вам могут встретиться уже знакомые темы. Даже если решения вам известны, вы взглянете на них под другим углом. Лучшие практики полезно подкреплять, изучая ошибки других разработчиков и причины, по которым этих ошибок желательно избегать.

Отношения между разработчиками и администраторами баз данных нередко складываются напряженно. Если вы администратор базы данных, эта книга поможет вам объяснить своим коллегам-разработчикам принципы лучших практик и то, чем чреват отказ от них.

О книге

Основам языка SQL посвящено множество книг и интернет-ресурсов, поэтому предполагается, что читатель уже достаточно знаком с синтаксисом SQL, чтобы использовать язык и решать практические задачи.

Производительность, масштабируемость и оптимизация — важные характеристики приложений баз данных, особенно в веб-среде, но это не главные темы моей книги. Вот некоторые издания, которые можно порекомендовать по темам производительности и масштабируемости: «SQL Performance Tuning» [GP03], «High Performance MySQL, 4th Edition» [BT21], «Efficient MySQL Performance» [Nic21] и «Effective MySQL Optimizing SQL Statements» [Bra11].

Каждая РСУБД на основе SQL использует собственные инструменты и команды, но эта книга не справочник по командам и не сборник рецептов.

Фреймворки доступа к данным и библиотеки объектно-реляционного отображения — довольно полезные инструменты, но и они не занимают центральное место в книге.

Задачи администрирования и эксплуатации баз данных (такие, как настройка размера сервера, планирование мощности, установка и конфигурация, мониторинг, резервное копирование, анализ журналов и безопасность) очень важны и заслуживают отдельной книги.

Билл Карвин
Октябрь 2022 г.

Условные обозначения

В этом разделе описаны некоторые условные обозначения, используемые в книге.

Шрифты

Ключевые слова SQL записываются в верхнем регистре моноширинным шрифтом, чтобы они выделялись в тексте, например `SELECT`.

В именах таблиц SQL, также оформленных моноширинным шрифтом, каждое слово записывается с прописной буквы, например `Accounts` или `BugsProducts`. Имена столбцов SQL, также оформленные моноширинным шрифтом, записываются в нижнем регистре, а слова разделяются символами подчеркивания, например `account_name`.

Строковые литералы записываются курсивом, например *bill@example.com*.

Терминология

Название языка SQL произносится «эс-ку-эль», а не «сиквэл». Впрочем, оба варианта встречаются достаточно часто, так что окружающие в любом случае поймут, что вы имеете в виду.

В SQL термины «*запрос*» (query) и «*инструкция*» (statement) считаются отчасти взаимозаменяемыми; под ними понимается любая завершенная команда SQL, которую можно выполнить. В книге для большей ясности термином «*запрос*» обозначаются инструкции `SELECT`, в остальных случаях используется термин «*инструкция*».

Онлайн-ресурсы

Примеры и исходный код, приведенные в книге, находятся по ссылке на вебсайте Pragmatic Bookshelf[1]. На этом же сайте можно сообщить о найденных ошибках в англоязычном издании и отправить свои предложения.

Если вам понравилась эта книга и она пригодилась вам в работе, надеюсь, вам захочется рассказать о ней другим — ваши рецензии очень ценны. Твиты и посты в соцсетях — отличный способ поделиться информацией. Со мной можно связаться в Твиттере на канале *@billkarwin* или написать напрямую *@pragprog*.

[1] https://pragprog.com/book/bksap1

О научном редакторе русского издания

Александр Петраки — старший инженер-разработчик компании КРОК. Занимается проектированием архитектуры высоконагруженных приложений и выполняет реализацию back-end части на Java и Spring с применением СУБД MySQL, PostgreSQL, Oracle, JanusGraph.

От издательства

Мы выражаем огромную благодарность компании КРОК за помощь в работе над русскоязычным изданием книги и их вклад в повышение качества переводной литературы.

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Эксперт — это человек, который совершил все возможные ошибки в очень узкой специальности.

➤ *Нильс Бор*

ГЛАВА 1

Что такое антипаттерн?

Антипаттерн — прием, предназначенный для решения одной проблемы, но при этом часто создающий другие. Существует множество способов применения антипаттернов, но у них у всех есть общее. Разработчик может прийти к антипаттерну самостоятельно, либо воспользовавшись советом коллеги, либо прочитав о нем в книге или в статье. Многие антипаттерны из области объектно-ориентированного проектирования и управления проектами представлены в сборнике паттернов Portland Pattern Repository[1], а также в книге 1998 года AntiPatterns [BMMM98].

В этой книге описаны самые частые промахи, которые по наивности допускают разработчики при использовании SQL. Я общался с такими разработчиками на семинарах, посвященных вопросам технической поддержки и обучения, работал вместе с ними над проектами и отвечал на их вопросы на интернет-форумах. Многие из этих ошибок я совершал сам; ничто не учит так хорошо, как исправление собственных ляпов долгими вечерами.

Типы антипаттернов

Книга включает в себя четыре части в соответствии с категориями антипаттернов.

- *Антипаттерны логического проектирования баз данных*. Прежде чем браться за написание кода, следует решить, какая информация должна храниться в базе данных, и выбрать лучший способ упорядочить и связать эти данные. В частности, на этом этапе проводится планирование таблиц баз данных, столбцов и отношений между ними.

[1] https://wiki.c2.com/?AntiPattern

- *Антипаттерны физического проектирования баз данных*. Определив, какие данные необходимо сохранить, вы реализуете управление данными настолько эффективно, насколько позволяет технология используемой РСУБД. На этой стадии определяются таблицы и индексы, а также выбираются типы данных. При этом используется *язык определения данных* SQL с такими инструкциями, как `CREATE TABLE`.
- *Антипаттерны запросов*. Данные необходимо добавить в БД, а потом прочитать их. Запросы SQL записываются на *языке манипулирования данными* с такими инструкциями, как `SELECT`, `UPDATE` и `DELETE`.
- *Антипаттерны разработки приложений*. Язык SQL предназначен для использования в контексте приложений, написанных на другом языке, таком как C++, Java, PHP, Python или Ruby. Существуют правильные и неправильные способы применения SQL в приложении, и в этой части книги описываются некоторые распространенные ошибки.

В области разработки или эксплуатации ПО встречается множество других антипаттернов, но в этой книге мы сосредоточимся на тех, которые имеют отношение к языку SQL.

Кроме того, в книге разбираются *мини-антипаттерны*. К этой категории относятся другие ошибки, часто совершаемые разработчиками при работе с SQL. Мини-антипаттерны рассматриваются не так подробно, как основные антипаттерны.

Многие антипаттерны имеют забавные или образные названия. Такие имена часто присваивают паттернам проектирования (как «правильным», так и «неправильным», то есть антипаттернам), чтобы их было проще запоминать.

В приложении содержатся практические описания некоторых теоретических концепций реляционных баз данных. Многие антипаттерны, приведенные в книге, возникают из-за неверного понимания теории баз данных.

Анатомия антипаттерна

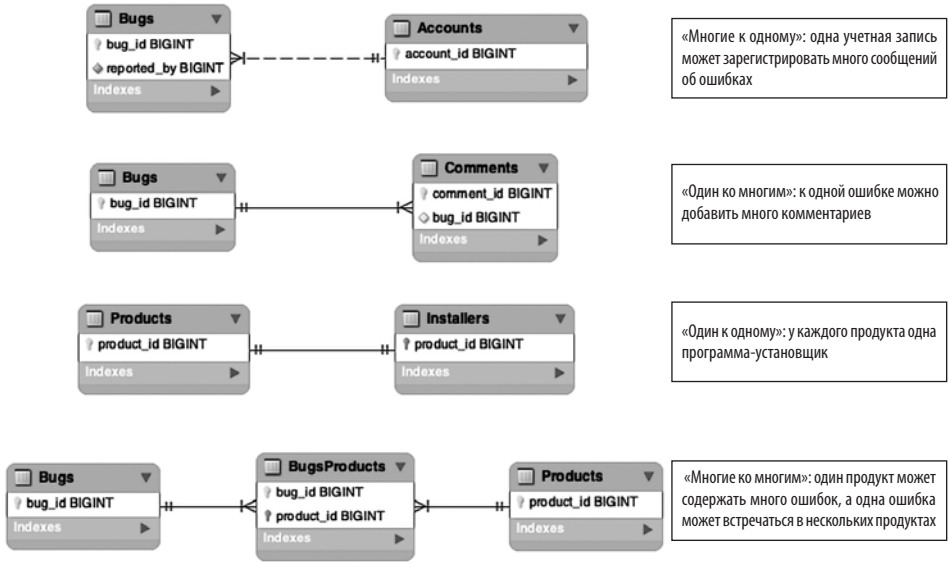
Глава, посвященная каждому антипаттерну, состоит из следующих подразделов:

- *Цель*. Это проблема, которую необходимо решить. Антипаттерны должны стать ее решением, но в конечном итоге порождают больше проблем, чем решают.
- *Антипаттерн*. В этом разделе описывается природа типичного решения, а также непредвиденные последствия, из-за которых оно становится антипаттерном.

- *Как распознать антипаттерн*. Существуют признаки, которые помогают выявить антипаттерны в проекте. На присутствие антипаттерна могут указывать сложности, с которыми вы сталкиваетесь, или фразы, которые произносите вы или ваши коллеги.
- *Допустимые применения антипаттерна*. Иногда то, что обычно считается антипаттерном, может оказаться вполне приемлемым решением (или по крайней мере меньшим из возможных зол).
- *Решение*. В этом разделе описываются предпочтительные способы решения исходной проблемы, позволяющие избежать нежелательных последствий, вызываемых антипаттерном.

Диаграммы «объект — отношение»

Для наглядного представления реляционных баз данных чаще всего используются *диаграммы «объект — отношение»* (entity-relationship diagram). На таких диаграммах таблицы изображаются в виде блоков, а отношения — в виде линий, соединяющих блоки; при этом условные знаки на концах линии описывают кратность отношения. Вот несколько примеров диаграмм «объект — отношение»:



Пример базы данных

Многие темы в книге поясняются на примере базы данных вымышленного приложения для отслеживания ошибок.

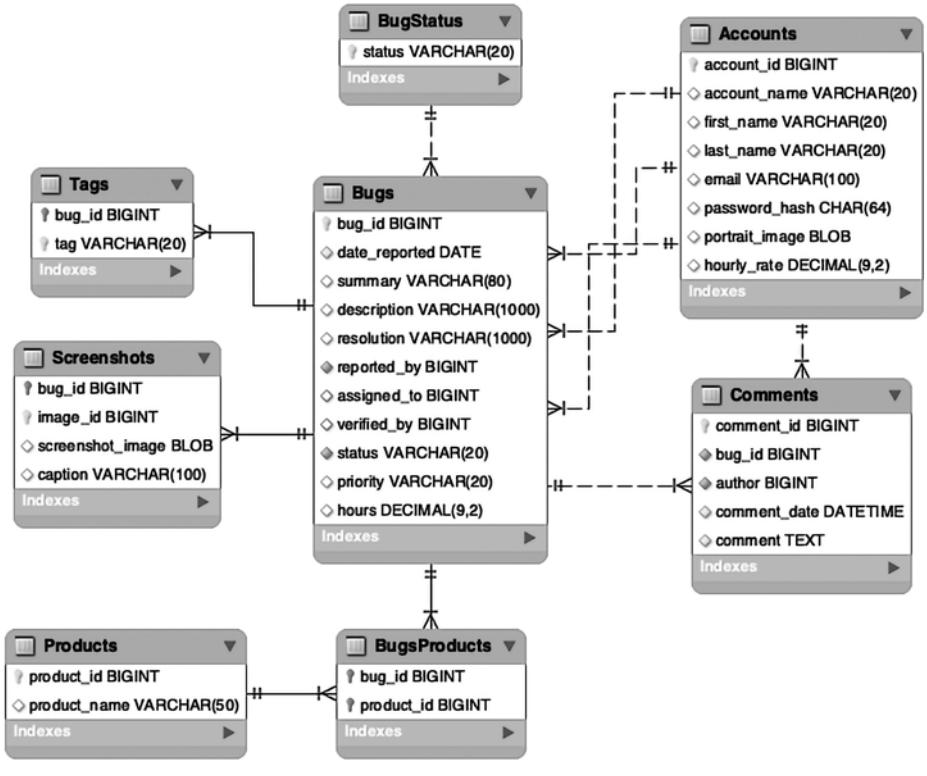
Следующий фрагмент языка определения данных содержит определения таблиц в SQL. В некоторых случаях выбор решения обусловлен примерами, которые приводятся далее в книге, так что он не всегда совпадает с выбором, который был бы сделан в реальных условиях. В примерах используется только стандартный вариант SQL, чтобы они работали в любой конкретной базе данных, но в них могут встречаться и специфические типы данных MySQL, такие как `SERIAL` и `BIGINT`.

Introduction/setup.sql

```
CREATE TABLE Accounts (
  account_id SERIAL PRIMARY KEY,
  account_name VARCHAR(20),
  first_name VARCHAR(20),
  last_name VARCHAR(20),
  email VARCHAR(100),
  password_hash CHAR(64),
  portrait_image BLOB,
  hourly_rate NUMERIC(9,2)
);

CREATE TABLE BugStatus (
  status VARCHAR(20) PRIMARY KEY
);

CREATE TABLE Bugs (
  bug_id SERIAL PRIMARY KEY,
  date_reported DATE NOT NULL DEFAULT (CURDATE()),
  summary VARCHAR(80),
  description VARCHAR(1000),
  resolution VARCHAR(1000),
  reported_by BIGINT UNSIGNED NOT NULL,
  assigned_to BIGINT UNSIGNED,
  verified_by BIGINT UNSIGNED,
  status VARCHAR(20) NOT NULL DEFAULT 'NEW',
  priority VARCHAR(20),
  hours NUMERIC(9,2),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (verified_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (status) REFERENCES BugStatus(status)
);
```

```
CREATE TABLE Comments (
  comment_id SERIAL PRIMARY KEY,
  bug_id BIGINT UNSIGNED NOT NULL,
  author BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  comment TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);

CREATE TABLE Screenshots (
  bug_id            BIGINT UNSIGNED NOT NULL,
  image_id          BIGINT UNSIGNED NOT NULL,
  screenshot_image  BLOB,
  caption           VARCHAR(100),
  PRIMARY KEY (bug_id, image_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);

CREATE TABLE Tags (
  bug_id BIGINT UNSIGNED NOT NULL,
  tag VARCHAR(20) NOT NULL,
  PRIMARY KEY (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);

CREATE TABLE Products (
  product_id SERIAL PRIMARY KEY,
  product_name VARCHAR(50)
);

CREATE TABLE BugsProducts(
  bug_id BIGINT UNSIGNED NOT NULL,
  product_id BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (bug_id, product_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

Диаграмма «объект — отношение» для базы данных ошибок:



В некоторых главах, особенно из части I «Антипаттерны логического проектирования баз данных», приводятся другие определения базы данных — либо в целях демонстрации антипаттерна, либо чтобы представить альтернативное решение, позволяющее избежать антипаттерна.

Эта глава кратко познакомила вас с антипаттернами, и теперь вы знаете их основные признаки. Перейдем к изучению отдельных антипаттернов и проблем, которые они создают.

ЧАСТЬ I

Антипаттерны логического проектирования баз данных

Прежде чем браться за написание кода, следует решить, какая информация должна храниться в базе данных, и выбрать лучший способ упорядочить и связать эти данные. В частности, на этом этапе проводится планирование таблиц баз данных, столбцов и отношений между ними.

Один инженер из Netscape, имя которого мы называть не будем, однажды передал указатель в JavaScript, сохранил его в строке, а потом передал обратно в C. Было много жертв.

➤ *Блейк Росс (Blake Ross)*

ГЛАВА 2

Кривая дорожка

Вы работаете над приложением для отслеживания ошибок и разрабатываете функциональность, которая позволяет назначить пользователя основным контактным лицом для продукта. Исходная архитектура позволяла выбрать только одного пользователя в качестве контакта. Впрочем, как это часто бывает, потом от вас потребовали, чтобы контактными лицами можно было назначить сразу нескольких пользователей.

В тот момент решение казалось простым: изменить базу данных, чтобы вместо одного идентификатора, как в предыдущей версии, в ней хранился список идентификаторов учетных записей пользователей, разделенных запятыми.

Но вскоре вас вызывает начальник и сообщает о возникшей проблеме. «Техотдел добавляет в проекты ассистентов. Они говорят, что могут добавить только пять человек. Когда пытаются добавить больше, происходит ошибка. В чем дело?»

«Да, в проект можно добавить ограниченное число контактов» — киваете вы, словно это совершенно обычное дело.

Похоже, шефу требуется более подробное объяснение. «Ну, от пяти до десяти, может, чуть больше. Зависит от того, когда были созданы их учетные записи». Шеф удивленно поднимает брови. Вы продолжаете: «Я храню идентификаторы учетных записей для проекта в списке, разделенном запятыми. Список идентификаторов должен помещаться в строке максимальной длины. Если идентификаторы учетных записей короткие, в списке их поместится больше. Учетным записям, созданным первыми, назначаются идентификаторы 99 и менее, и они занимают меньше места».

Шеф хмурится. Похоже, вам придется поработать по вечерам.

Разработчики часто используют списки, разделенные запятыми, чтобы не создавать таблицы пересечений для отношений «многие ко многим». Такой антипаттерн называется «Кривая дорожка» (jaywalking)[1, 2].

Цель: хранение многозначных атрибутов

Если столбец таблицы может содержать только одно значение, ее структура получается простой: достаточно выбрать тип данных SQL для представления одного экземпляра этого значения (например, целого числа, даты или строки). Не совсем понятно, как сохранить в столбце коллекцию связанных значений.

В примере с БД для отслеживания ошибок продукт связывается с контактом через целочисленный столбец таблицы `Products`. С каждой учетной записью может быть связано несколько продуктов, и каждый продукт хранит ссылку на один контакт, так что продукты связаны с учетными записями отношением «многие ко многим».

Jaywalking/obj/create.sql

```
CREATE TABLE Products (
  product_id SERIAL PRIMARY KEY,
  product_name VARCHAR(1000),
  account_id BIGINT UNSIGNED,
  -- . . .
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', 12);
```

В процессе развития продукта вы понимаете, что контактов для него может быть несколько. Кроме отношения «многие к одному», необходимо поддерживать отношение «один ко многим» от продуктов к учетным записям. Одна строка данных таблицы `Products` должна быть способна хранить сразу несколько контактов.

Антипаттерн: форматирование списка, разделенного запятыми

Чтобы свести к минимуму изменения в структуре базы данных, вы решаете переопределить столбец `account_id` с типом `VARCHAR`, чтобы в нем можно

---

[1] Jaywalking — в частности, это переход улицы в неположенном месте. Дальше у автора игра слов: jaywalking — это избегание перекрестка (intersection), поэтому так назвали паттерн, родившийся из-за избегания пересечений (intersections). — *Примеч. пер.*

[2] Этот антипаттерн часто называют просто Multi-Valued Attribute. — *Примеч. науч. ред.*

было хранить несколько идентификаторов учетных записей, разделенных запятыми.

Jaywalking/anti/create.sql

```
  CREATE TABLE Products (
  product_id SERIAL PRIMARY KEY,
  product_name VARCHAR(1000),
  account_id VARCHAR(100), -- список, разделенный запятыми
-- . . .
);

INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', '12,34');
```

Идея кажется удачной, потому что вам не пришлось создавать дополнительные таблицы или столбцы; вы изменили тип данных всего одного столбца. Но давайте присмотримся к проблемам производительности и целостности данных, свойственным такой структуре таблицы.

Запрос продуктов для конкретной учетной записи

Объединение всех внешних ключей в одном поле усложняет запросы. Теперь невозможно использовать равенство; вместо него приходится применять поиск по шаблону. Например, запрос для получения всех продуктов для учетной записи 12 в MySQL будет выглядеть примерно так:

Jaywalking/anti/regexp.sql

```
SELECT * FROM Products WHERE account_id REGEXP '\\b12\\b';
```

Выражения с поиском по шаблону могут возвращать ложные совпадения. Производительность ухудшается, потому что при поиске совпадения невозможно извлечь пользу из индексов. Так как в разных базах данных используются разные варианты синтаксиса поиска по шаблону, код SQL будет привязан к конкретному продукту.

Запрос учетных записей для конкретного продукта

Кроме того, соединение списка, разделенного запятыми, с соответствующими строками таблицы выполняется слишком медленно и неудобно.

Jaywalking/anti/regexp.sql

```
SELECT * FROM Products AS p JOIN Accounts AS a
    ON p.account_id REGEXP '\\b' || a.account_id || '\\b'
WHERE p.product_id = 123;
```

Соединение двух таблиц в таких выражениях делает использование индексов полностью невозможным, так что производительность снова страдает. Запрос должен просканировать обе таблицы, сгенерировать перекрестное произведение и применить регулярное выражение к каждой комбинации строк данных.

Создание агрегатных запросов

Агрегатные запросы используют такие функции, как `COUNT()`, `SUM()` и `AVG()`. Однако эти функции проектировались для использования с группами строк, а не со списками, разделенными запятыми. Приходится идти на разные ухищрения, например, вычислять длину строки со значениями, разделенными запятыми, за вычетом длины строки с удаленными запятыми. Результат может использоваться для подсчета элементов в списке.

Jaywalking/anti/count.sql

```
SELECT product_id,
    LENGTH(account_id) - LENGTH(REPLACE(account_id, ',', '')) + 1
      AS contacts_per_product
FROM Products;
```

Такие трюки впечатляют, но они неочевидны. На разработку таких решений уходит много времени, а отлаживать их трудно. Некоторые агрегатные запросы вообще невозможно реализовать, как ни старайся.

Обновление учетных записей для конкретного продукта

Новый идентификатор можно добавить в конец списка с помощью конкатенации строк, но это может привести к нарушению сортировки списка.

Jaywalking/anti/update.sql

```
UPDATE Products
SET account_id = account_id || ',' || 56
WHERE product_id = 123;
```

Чтобы удалить элемент из списка, необходимо выполнить два запроса SQL: на получение старого списка и на сохранение нового.

Jaywalking/anti/remove.py

```
import mysql.connector

cnx = mysql.connector.connect(user='scott', database='test')
cursor = cnx.cursor()

product_id_to_search = 2
value_to_remove = '34'
```

```
query = "SELECT product_id, account_id FROM Products WHERE product_id = %s"
cursor.execute(query, (product_id_to_search,))
for (row) in cursor:
    (product_id, account_ids) = row
    account_id_list = account_ids.split(",")
    account_id_list.remove(value_to_remove)
    account_ids = ",".join(account_id_list)
    query = "UPDATE Products SET account_id = %s WHERE product_id = %s"
    cursor.execute(query, (account_ids, product_id,))

cnx.commit()
```

Слишком много кода, чтобы просто удалить элемент из списка.

Проверка идентификаторов продуктов

Что мешает пользователю ввести недопустимый идентификатор, например `banana`?

Jaywalking/anti/banana.sql

```
INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', '12,34,banana');
```

Пользователь рано или поздно введет что-нибудь не то, и ваша база данных превратится в тыкву. Возможно, обойдется без ошибок, но полученная информация не будет иметь никакого смысла.

Даже если значения являются целыми числами, нельзя быть уверенными, что эти целые числа присутствуют в таблице `Accounts`. Стандартный способ проверки основан на использовании ограничения внешнего ключа, но внешние ключи могут проверять только целые столбцы, а не отдельные элементы списка.

Выбор символа-разделителя

Если вы храните список строковых значений вместо целых чисел, некоторые элементы списка могут содержать символ-разделитель. Использование запятой в качестве разделителя между элементами может быть нежелательно. Можно выбрать другой символ, но нельзя гарантировать, что он больше нигде не встретится.

Ограничения длины списка

Сколько элементов списка можно сохранить в столбце `VARCHAR(30)`? Зависит от длины каждого из них. Если все элементы имеют длину в два символа, получится сохранить десять элементов (с учетом запятых). Если каждый элемент имеет длину в шесть символов, поместятся только четыре элемента:

Jaywalking/anti/length.sql

```
UPDATE Products SET account_id = '10,14,18,22,26,30,34,38,42,46'
WHERE product_id = 123;

UPDATE Products SET account_id = '101418,222630,343842,467790'
WHERE product_id = 123;
```

Как гарантировать, что `VARCHAR(30)` поддерживает самый длинный список, который вам может понадобиться в будущем? Какой длины будет достаточно? Попробуйте-ка объяснить причины ограничений длины шефу или заказчику.

Как распознать антипаттерн

Если вы слышите из уст коллег подобные фразы, это может указывать на применение антипаттерна «Кривая дорожка»/Multi-valued Attribute:

- «Сколько элементов максимум должно быть в этом списке?»

  Этот вопрос часто возникает при выборе максимальной длины столбца `VARCHAR`.

- «А ты знаешь, как найти границу слова в SQL?»

  Если вы используете регулярные выражения для выделения частей строки, возможно, эти части следует хранить по отдельности.

- «Какого символа нет ни в одном элементе списка?»

  Вы ищете символ-разделитель, который не создаст неоднозначности, но любой символ когда-нибудь может встретиться в элементе списка.

Допустимые применения антипаттерна

Можно повысить производительность некоторых запросов, применяя *денормализацию* к структуре базы данных. Хранение списков в виде строк, разделенных запятыми, является примером денормализации.

Приложению могут требоваться данные, разделенные запятыми, но при этом ему не нужно обращаться к отдельным элементам списка. Аналогичным образом, если приложение получает данные, разделенные запятыми, из другого источника и вы просто хотите сохранить весь список в базе данных и позднее прочитать его точно в таком же виде, разделять значения не обязательно.

Если вы решите применить денормализацию, действуйте консервативно. Начните с использования нормализованной структуры базы данных, потому что с ней код приложения может быть более гибким и она помогает сохранить целостность данных в базе.

Некоторые продукты баз данных SQL расширяют типы данных SQL разновидностями типа массива. Поддержка массивов предусмотрена в PostgreSQL, Oracle,

IBM DB2 и Informix. В зависимости от реализации они помогают избежать некоторых проблем, описанных ранее в этой главе. Например, можно задать скалярный тип данных для элементов массива. Тем не менее они не решат всех проблем. Их сложно использовать, и вам придется учиться работать с каждым диалектом SQL по отдельности.

Решение: создание таблицы пересечений

Вместо того чтобы хранить `account_id` в таблице `Products`, храните значение в отдельной таблице, чтобы каждое значение атрибута занимало отдельную строку. Новая таблица `Contacts` реализует отношение «многие ко многим» между `Products` и `Accounts`:

Jaywalking/soln/create.sql

```
CREATE TABLE Contacts (
  product_id BIGINT UNSIGNED NOT NULL,
  account_id BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (product_id, account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id),
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

INSERT INTO Contacts (product_id, account_id)
VALUES (123, 12), (123, 34), (345, 23), (567, 12), (567, 34);
```

Таблица с внешними ключами, относящимися к двум таблицам, называется *таблицей пересечений* (intersection table). Иногда эта таблица называется «таблицей соединения», «таблицей “многие к многим”», «таблицей отображения» или как-то еще. Название роли не играет; принцип остается тем же. Эта таблица реализует отношение «многие ко многим» между двумя связываемыми таблицами. Это означает, что каждый продукт может быть связан через таблицу пересечений с несколькими учетными записями, а каждая учетная запись может быть связана с несколькими продуктами. Диаграмма «объект — отношение» выглядит так:

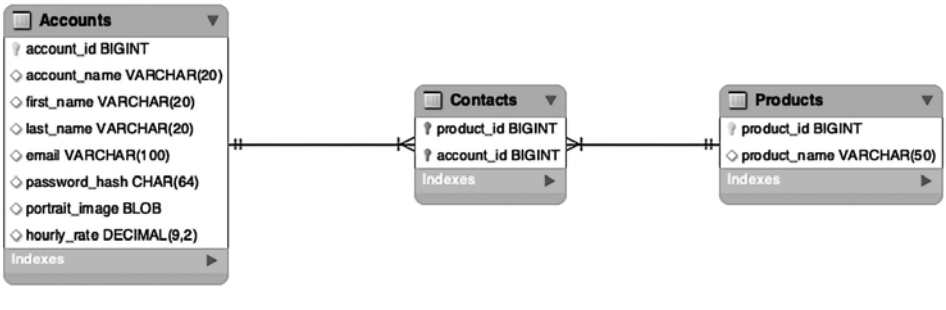


Таблица пересечений решает все проблемы, описанные в разделе «Антипаттерн: форматирование списка, разделенного запятыми».

Запрос продукта по учетной записи и наоборот

Чтобы получить атрибуты всех продуктов для заданной учетной записи, проще выполнить соединение таблицы `Products` с таблицей `Contacts`:

Jaywalking/soln/join.sql

```
SELECT p.*
FROM Products AS p JOIN Contacts AS c ON (p.product_id = c.product_id)
WHERE c.account_id = 34;
```

Некоторые разработчики не любят запросы, содержащие соединения; они считают их слишком медленными. Однако этот запрос использует индексы намного эффективнее, чем решение, приведенное выше в разделе «Антипаттерн: форматирование списка, разделенного запятыми».

Кроме того, запросы информации учетной записи легко читать и оптимизировать. Вместо непонятных регулярных выражений в них используются индексы для эффективного соединения:

Jaywalking/soln/join.sql

```
SELECT a.*
FROM Accounts AS a JOIN Contacts AS c ON (a.account_id = c.account_id)
WHERE c.product_id = 123;
```

Создание агрегатных запросов

Следующий пример возвращает количество учетных записей, связанных с каждым продуктом:

Jaywalking/soln/group.sql

```
SELECT product_id, COUNT(*) AS accounts_per_product
FROM Contacts
GROUP BY product_id;
```

Получить количество продуктов на учетную запись также просто:

Jaywalking/soln/group.sql

```
SELECT account_id, COUNT(*) AS products_per_account
FROM Contacts
GROUP BY account_id;
```

Можно запросить и более сложную сводную информацию, например продукт с наибольшим количеством учетных записей:

Jaywalking/soln/group.sql

```
SELECT product_id, COUNT(*) AS accounts_per_product
FROM Contacts
GROUP BY product_id
ORDER BY accounts_per_product DESC
LIMIT 1;
```

Обновление контактов для конкретного продукта

Добавление или удаление элементов из списка реализуется вставкой или удалением строк данных в таблице пересечений. Каждая ссылка на продукт хранится в отдельной строке таблицы `Contacts`, что позволяет добавлять или удалять их по одной.

Jaywalking/soln/remove.sql

```
INSERT INTO Contacts (product_id, account_id) VALUES (456, 34);

DELETE FROM Contacts WHERE product_id = 456 AND account_id = 34;
```

Проверка идентификаторов продуктов

Внешний ключ также может использоваться для проверки элементов на принадлежность множеству допустимых значений в другой таблице. Вы объявляете, что `Contacts.account_id` ссылается на `Accounts.account_id`, и доверяете базе данных обеспечивать ссылочную целостность. Теперь можно гарантировать, что таблица пересечений содержит только существующие идентификаторы учетных записей.

Также для ограничения значений элементов можно использовать типы данных SQL. Например, если элементы списка должны быть значениями `INTEGER` или `DATE` и вы объявляете столбец с этими типами данных, вы можете быть уверены, что все элементы содержат значения, допустимые для этого типа (а не бессмыслицу вроде *banana*).

Выбор символа-разделителя

Символ-разделитель не используется, так как каждый элемент хранится в отдельной строке. Присутствие в элементах запятых или других символов, которые могут использоваться в качестве разделителей, не создает никакой неоднозначности.

Ограничения на длину списка

Так как каждый элемент находится в отдельной строке таблицы пересечений, список ограничивается только количеством строк, физически существующих в одной таблице. Если задача требует ограничения количества элементов, это

можно обеспечить с помощью подсчета числа элементов (а не совокупной длины списка).

Другие преимущества таблицы пересечений

Индекс по `Contacts.account_id` повышает производительность и работает более эффективно, чем поиск подстроки в списке, разделенном запятыми. Объявление внешнего ключа для столбца неявно строит индекс по этому столбцу во многих базах данных (но подробности ищите в документации к своей БД).

Кроме того, можно создать дополнительные атрибуты для каждого элемента, для чего в таблицу пересечений добавляются столбцы. Например, можно сохранить дату добавления контакта для заданного продукта или атрибут, отличающий первоочередный контакт от второстепенных. Со списком, разделенным запятыми, это сделать невозможно.

Сохраняйте каждое значение в отдельном столбце и строке.

Мини-антипаттерн: разбиение CSV-данных на строки

Если вы испытываете сложности с использованием данных в строках, разделенных запятыми, как в антипаттерне «Кривая дорожка», это может говорить о такой проблеме, как расширение строки значений, разделенных запятыми, в набор строк, как если бы они изначально хранились по одному значению на строку. Например, вы не сможете изменить способ хранения данных, потому что большой объем существующего кода зависит от текущего формата, но вам все равно потребуется изменить выводимые данные, по крайней мере для результата конкретного запроса.

Например, вам может понадобиться запрос, который возвращает список учетных записей по одной на строку, чтобы результат запроса можно было передать в другую программу (например, электронную таблицу). По какой-то причине вам необходимо разбить строку, но вы не можете рассчитывать на то, что в SQL найдется подходящая функция для решения этой задачи. В некоторых разновидностях SQL такая функция есть. В PostgreSQL имеется нестандартная функция `string_to_array()`, которая преобразует строку, разделенную запятыми, к типу массива. Далее массив можно передать функции `unnest()`, которая расширяет массив в набор строк данных:

Jaywalking/mini/unnest.sql

```
SELECT a FROM Products
CROSS JOIN unnest(string_to_array(account_id, ',')) AS a;
```

В Microsoft SQL Server 2016 также существуют свои нестандартные операции:

Jaywalking/mini/cross-apply.sql

```
SELECT product_id, product_name, value
FROM Products CROSS APPLY STRING_SPLIT(account_id, ',');
```

Другое решение основано на соединении списка, разделенного запятыми, с предварительно определенным множеством целых чисел, по одному целому числу на строку данных. Для каждой из полученных соединенных записей используется выражение подстроки для извлечения N-го элемента из списка, разделенного запятыми, как показано в следующем примере.

Jaywalking/mini/int-table.sql

```
SELECT p.product_id, p.product_name,
  SUBSTRING_INDEX(SUBSTRING_INDEX(p.account_id, ',', n.n), ',', -1)
  AS account_id
FROM Products AS p
JOIN Numbers AS n
  ON n.n <= LENGTH(p.account_id) - LENGTH(REPLACE(p.account_id, ',', ''));
```

Этот запрос будет труднее понять. Вам понадобится таблица `Numbers`, которая содержит столбец `n`, заполненный целыми числами от единицы. Затем таблица чисел соединяется с `Products`, чтобы количество строк данных было равно количеству элементов, разделенных запятыми. Количество запятых вычисляется как разность между длиной всей строки и длиной строки без запятых. Наконец, в списке `SELECT` функция MySQL `SUBSTRING_INDEX()` используется для извлечения подстрок до *n*-го элемента, после чего функция используется снова с аргументом `-1`, чтобы оставить только последний элемент списка.

Таблица с последовательностью целых чисел не всегда оказывается под рукой. Иногда приходится генерировать последовательность динамически. Для этого можно воспользоваться операцией `UNION`:

Jaywalking/mini/int-union.sql

```
SELECT p.product_id, p.product_name,
  SUBSTRING_INDEX(SUBSTRING_INDEX(p.account_id, ',', n.n), ',', -1)
  AS account_id
FROM Products AS p
JOIN (
  SELECT 1 AS n UNION SELECT 2 UNION SELECT 3 UNION SELECT 4 -- and so on
) AS n
  ON n.n <= LENGTH(p.account_id) - LENGTH(REPLACE(p.account_id, ',', ''));
```

Наконец, можно воспользоваться рекурсивным решением, которое последовательно возвращает каждый элемент списка:

Jaywalking/mini/recursive.sql

```
WITH RECURSIVE cte AS (
  SELECT product_id, product_name,
    SUBSTRING_INDEX(account_id, ',', 1) AS account_id,
    SUBSTRING(account_id, LENGTH(SUBSTRING_INDEX(account_id, ',', 1))+2)
    AS remainder
  FROM Products
  UNION ALL
  SELECT product_id, product_name, SUBSTRING_INDEX(remainder, ',', 1),
    SUBSTRING(remainder, LENGTH(SUBSTRING_INDEX(remainder, ',', 1))+2)
  FROM cte
  WHERE LENGTH(remainder) > 0
)
SELECT product_id, product_name, account_id FROM cte;
```

Похоже, задача получается более сложной, чем должна быть. Некоторые из этих решений работают только с отдельными разновидностями или версиями баз данных SQL. Лучшее решение, которое работает в любой базе данных, — хранить данные в том виде, в каком они должны использоваться (и не совершать ошибку антипаттерна «Кривая дорожка»).

Если бы у меня было пять минут, чтобы срубить дерево,
я бы потратил первые три на то, чтобы наточить топор.

➤ *Аноним*

ГЛАВА 3

Наивное применение деревьев

Предположим, вы разработчик популярного сайта с новостями науки и техники.

Это современный сайт, так что читатели могут оставлять комментарии и даже отвечать друг другу в ветках обсуждений. Такие ветки могут расходиться и уходить на достаточную глубину. Вы выбираете простое решение для отслеживания этих цепочек ответов: каждый последующий комментарий ссылается на тот комментарий, на который он отвечает.

Trees/intro/parent.sql

```
CREATE TABLE Comments (
  comment_id SERIAL PRIMARY KEY,
  parent_id BIGINT UNSIGNED,
  comment TEXT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Comments(comment_id)
);
```

Однако вскоре становится ясно, что загружать длинную цепочку ответов в одном запросе SQL трудно. Вы можете получить непосредственных потомков («детей») и, возможно, соединиться с потомками следующего уровня («внуками») до фиксированной глубины. Если ветви могут иметь *неограниченную* длину, вам придется выполнить много запросов SQL, чтобы получить все комментарии для заданной ветви.

Тогда у вас появляется другая идея: загрузить *все* комментарии и собрать из них древовидные структуры в памяти приложения, используя традиционные алгоритмы деревьев, которые вы изучали в школе. К сожалению, редакторы сайта говорят, что ежедневно на сайте публикуются десятки статей и к каждой статье могут быть сотни комментариев. Сортировать миллионы комментариев каждый раз, когда кто-то просматривает сайт, нецелесообразно.

Должен быть более эффективный способ хранения веток комментариев, который бы позволял легко и эффективно загрузить всю цепочку обсуждения.

Цель: хранение и загрузка иерархий

Между данными могут существовать рекурсивные связи. Данные могут быть организованы в древовидные или иерархические структуры. В древовидной структуре каждый элемент называется *узлом*. Узел может иметь много детей и одного родителя. Верхний узел, не имеющий родителя, называется *корнем*. Узлы нижнего уровня, не имеющие потомков, называются листовыми узлами (или просто *листьями*). Узлы, находящиеся между ними, называются *нелистовыми*.

Может возникнуть необходимость в запросе отдельных элементов, связанных подмножеств или всей коллекции описанных выше иерархических данных.

Примеры древовидных структур данных:

Организационная схема: отношения между работниками и руководителями — классический пример древовидных структур. Организационные схемы рассматриваются в бесчисленных книгах и статьях по SQL. В организационной схеме у каждого работника есть руководитель, который представляет *родителя* работника в древовидной структуре. Руководитель также является работником.

Структурированное обсуждение: как мы уже увидели в этой главе, древовидная структура может использоваться для представления цепочки комментариев, отвечающих на другие комментарии. В дереве непосредственными потомками узла комментария являются его ответы.

В этой главе мы рассмотрим антипаттерн и возможные решения на примере структурированного обсуждения.

Антипаттерн: постоянная зависимость от родителя

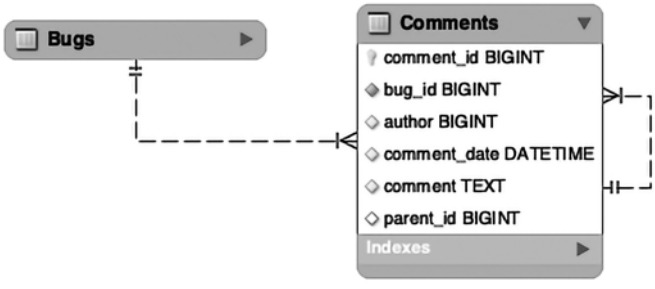
Наивным решением, часто встречающимся в книгах и статьях, является добавление столбца `parent_id`. Столбец ссылается на другой комментарий в той же таблице, и чтобы обеспечить соблюдение этих отношений, можно создать ограничение внешнего ключа.

Trees/anti/adjacency-list.sql

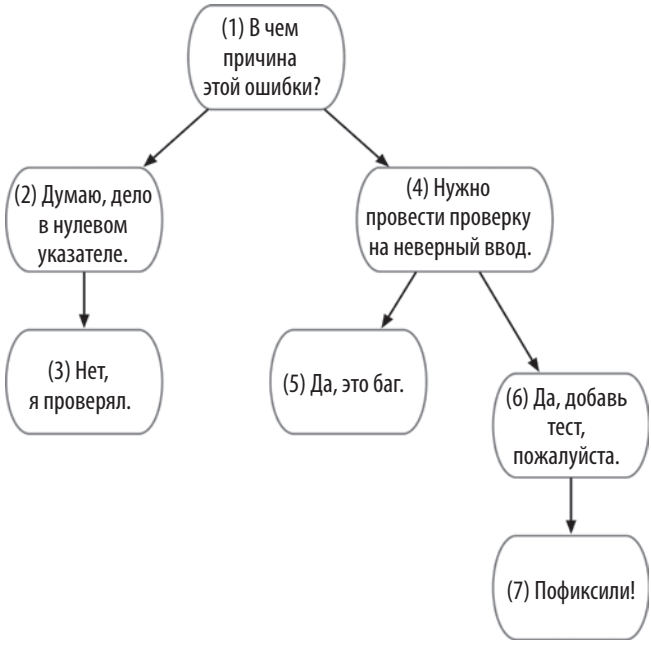
```
CREATE TABLE Comments (
  comment_id SERIAL PRIMARY KEY,
  bug_id BIGINT UNSIGNED NOT NULL,
  author BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  comment TEXT NOT NULL,
```

```
  parent_id BIGINT UNSIGNED,
  FOREIGN KEY (parent_id) REFERENCES Comments(comment_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

Такая структура называется *списком смежности* (adjacency list). Диаграмма «объект — отношение» для таких таблиц изображена ниже.



Это самая распространенная структура, используемая разработчиками для хранения иерархических данных. Пример такой древовидной структуры представлен на следующей диаграмме.



А вот как выглядят те же данные в табличном формате.

comment_id	parent_id	author	comment
1	NULL	Fran	В чем причина этой ошибки?
2	1	Ollie	Думаю, дело в нулевом указателе.
3	2	Fran	Нет, я проверял.
4	1	Kukla	Нужно провести проверку на неверный ввод.
5	4	Ollie	Да, это баг.
6	4	Fran	Да, добавь тест, пожалуйста.
7	6	Kukla	Пофиксили!

Запросы к дереву с использованием списка смежности

Список смежности — верный способ хранения ссылок из дочерних узлов на родительский узел, однако разработчики часто выбирают неверное решение для самой частой задачи: получить всех потомков.

Для получения комментария и его непосредственных потомков достаточно довольно простого запроса:

Trees/anti/parent.sql

```
SELECT c1.*, c2.*
FROM Comments c1 LEFT OUTER JOIN Comments c2
  ON c2.parent_id = c1.comment_id;
```

Запрос возвращает только два уровня дерева. Одно из свойств дерева заключается в том, что оно может иметь произвольную глубину, так что необходимо предусмотреть возможность запросить всех потомков без ограничений по количеству уровней. Например, может потребоваться вычислить `COUNT()` для всех комментариев в ветке или `SUM()` для стоимости всех деталей механического узла.

Использовать такие запросы со списками смежности неудобно, потому что каждый уровень дерева соответствует очередному соединению, а количество соединений в запросе SQL должно быть фиксированным. Следующий запрос извлекает дерево с глубиной до 4, но не может загрузить более глубокое дерево:

Trees/anti/ancestors.sql

```
SELECT c1.*, c2.*, c3.*, c4.*
FROM Comments c1                      -- 1-й уровень
  LEFT OUTER JOIN Comments c2
    ON c2.parent_id = c1.comment_id  -- 2-й уровень
```

```
  LEFT OUTER JOIN Comments c3
    ON c3.parent_id = c2.comment_id  -- 3-й уровень
  LEFT OUTER JOIN Comments c4
    ON c4.parent_id = c3.comment_id; -- 4-й уровень
```

Запрос также получается громоздким, потому что он включает потомков уровней нарастающей глубины путем добавления новых столбцов. Это усложняет вычисление таких агрегатных функций, как `COUNT()`.

Другой способ запроса древовидной структуры по списку смежности заключается в выборке всех строк из коллекции без применения SQL для выборки подмножества строк (например, предков или потомков).

Trees/anti/all-comments.sql

```
SELECT * FROM Comments WHERE bug_id = 1234;
```

Ваша задача — написать код клиентского приложения для выборки этих строк данных и поэтапного построения древовидной структуры. Это означает, что вы загрузите больше строк, чем требуется для нужного вам поддерева. Ваш код должен отбросить любые строки данных, не входящие в это поддерево.

Копирование большого итогового множества из базы данных в приложение до его анализа влечет лишнюю трату сетевых и вычислительных ресурсов. Например, приложению может требоваться только поддерево, так что запрос с большой вероятностью вернет больше строк, чем нужно. Возможно, вам даже потребуется только сводная информация о данных, например количество `COUNT()` комментариев.

Поддержание дерева со списком смежности

На самом деле некоторые операции, например добавление новых листовых узлов, легко реализовать на базе списков смежности:

Trees/anti/insert.sql

```
INSERT INTO Comments (bug_id, parent_id, author, comment)
  VALUES (1234, 7, 12 /* Kukla */, 'Thanks!');
```

Перемещение отдельного узла или поддерева столь же просто:

Trees/anti/update.sql

```
UPDATE Comments SET parent_id = 3 WHERE comment_id = 6;
```

С удалением узла из дерева будет больше сложностей. Если требуется удалить все поддерево, придется выдать несколько запросов для нахождения всех потомков, а затем удалить потомков, начиная с нижнего уровня до соблюдения целостности внешнего ключа.

Trees/anti/delete-subtree.sql

```
SELECT comment_id FROM Comments WHERE parent_id = 4; -- возвращает 5 и 6
SELECT comment_id FROM Comments WHERE parent_id = 5; -- возвращает none
SELECT comment_id FROM Comments WHERE parent_id = 6; -- возвращает 7
SELECT comment_id FROM Comments WHERE parent_id = 7; -- возвращает none

DELETE FROM Comments WHERE comment_id IN ( 7 );
DELETE FROM Comments WHERE comment_id IN ( 5, 6 );
DELETE FROM Comments WHERE comment_id = 4;
```

Для автоматизации можно использовать внешний ключ с модификатором `ON DELETE CASCADE`, если вы уверены в том, что потомков всегда нужно удалять (а не повышать их уровень или перемещать).

Если же нужно удалить нелистовой узел и повысить уровень его непосредственных потомков или переместить их в другую точку дерева, сначала придется изменить значение `parent_id` его потомков, а затем удалить нужный узел.

Trees/anti/delete-non-leaf.sql

```
SELECT parent_id FROM Comments WHERE comment_id = 6; -- возвращает 4
UPDATE Comments SET parent_id = 4 WHERE parent_id = 6;
DELETE FROM Comments WHERE comment_id = 6;
```

Такие операции требуют выполнения нескольких шагов при использовании структуры со списком смежности. Таким образом, вам придется написать довольно много кода для задач, которые должны выполняться просто и с малыми усилиями.

Как распознать антипаттерн

Если вы слышите из уст коллег подобные фразы, это может указывать на применение антипаттерна «Наивные деревья»:

- «Сколько уровней поддерживать в деревьях?»

  Вам не удается получить всех потомков или всех предков узла без применения рекурсивных запросов. Можно пойти на компромисс и поддерживать только деревья ограниченной глубины, но тогда возникает следующий естественный вопрос: какую глубину считать достаточной?

- «Я боюсь трогать код, управляющий древовидными структурами данных».

  Вы выбрали одно из сложных решений по управлению иерархиями — и ошиблись. Любой метод упрощает одни задачи и в то же время усложняет другие. Возможно, выбранное решение оказалось неподходящим для работы с иерархиями в приложении.

- «Периодически я скриптом удаляю осиротевшие строки в деревьях».

В ходе удаления нелистовых узлов в приложении возникают несвязанные узлы дерева. Сложные структуры данных, хранящиеся в БД, необходимо поддерживать в целостном, корректном состоянии после любых изменений. Чтобы хранимые структуры данных были эластичными, а не хрупкими, можно воспользоваться одним из решений, представленных далее в этой главе, с триггерами и каскадными ограничениями внешних ключей.

Допустимые применения антипаттерна

Даже если используемая версия базы данных SQL не поддерживает рекурсивные запросы, решение со списком смежности может оказаться достаточным. Если вы собираетесь поддерживать в приложении деревья ограниченной глубины и вам не нужно запрашивать всех потомков, список смежности может стать отличным рабочим вариантом.

Избегайте чрезмерного усложнения

Я написал приложение для отслеживания ресурсов центра обработки данных. Часть оборудования была установлена внутри компьютеров; например, кэширующий контроллер дисков был установлен в стойке, а на контроллере дисков были установлены дополнительные модули памяти.

Мне понадобилось SQL-решение для отслеживания использования иерархических коллекций. Также было необходимо отслеживать отчеты о загрузке оборудования, амортизации и окупаемости по каждому отдельному устройству.

Мой шеф указал, что коллекции могут содержать подколлекции, и следовательно, дерево может иметь произвольную глубину. У меня ушло несколько недель на то, чтобы довести до ума код управления деревьями в хранилище базы данных, пользовательский интерфейс, администрирование и отчетность.

Однако на практике приложению никогда не требовалось создавать группировки оборудования с деревом, глубже чем отношение «родитель — ребенок». Если бы заказчик признал, что этого достаточно для моделирования его требований к ресурсам, это сэкономило бы немало времени и сил.

Решение: использование альтернативных моделей деревьев

Для работы с иерархическими данными в SQL можно воспользоваться любым из перечисленных ниже решений. У каждого есть свои достоинства и недостатки, и каждое может лучше или хуже подходить для конкретного приложения.

Рекурсивные запросы

Некоторые РСУБД реализуют синтаксис SQL для поддержки иерархий, хранящихся в формате списка смежности. В стандарте SQL-99 определяется синтаксис рекурсивных запросов с ключевым словом `WITH`, за которым следует рекурсивное *обобщенное табличное выражение* (common table expression).

Trees/soln/cte.sql

```
WITH RECURSIVE CommentTree
    (comment_id, bug_id, parent_id, author, comment_date, comment, depth)
AS (
    SELECT comment_id, bug_id, parent_id, author, comment_date,
      comment, 0 AS depth
    FROM Comments
    WHERE parent_id IS NULL
  UNION ALL
    SELECT c.comment_id, c.bug_id, c.parent_id, c.author, c.comment_date,
      c.comment, ct.depth+1 AS depth
    FROM CommentTree ct
    JOIN Comments c ON (c.parent_id = ct.comment_id)
)
SELECT * FROM CommentTree WHERE bug_id = 1234;
```

Ниже перечислены самые популярные системы управления базами данных SQL с поддержкой рекурсивных запросов, а также версии, в которых появилась эта возможность:

- Oracle 11g
- MySQL 8.0
- Microsoft SQL Server 2005
- PostgresSQL 8.4
- IBM DB2 UDB 8
- SQLite 3.8.3

В Oracle 9i и 10g поддерживается конструкция `WITH`, но не для рекурсивных запросов. Вместо этого в этих версиях поддерживается специализированный синтаксис `START WITH` и `CONNECT BY PRIOR`. Этот синтаксис может использоваться для выполнения рекурсивных запросов:

Trees/legit/connect-by.sql

```
SELECT * FROM Comments
START WITH comment_id = 9876
CONNECT BY PRIOR parent_id = comment_id;
```

Помимо списка смежности, существует ряд альтернативных моделей для хранения иерархических данных, включая *перечисление компонентов пути* (path enumeration), *вложенные множества* (nested sets) и *таблицу замыканий* (closure table).

В следующих трех разделах показаны примеры использования этих моделей для решения проблемы, описанной в разделе «Антипаттерн: постоянная зависимость от родителя», с сохранением и загрузкой древовидной коллекции комментариев.

К этим решениям привыкаешь не сразу. На первый взгляд они кажутся более сложными, чем список смежности, но они упрощают некоторые операции с деревьями, очень сложные или неэффективные при использовании списка смежности. Если ваше приложение должно выполнять эти операции, а использование рекурсивных запросов с обобщенными табличными выражениями кажется слишком сложным, то эти решения могут оказаться более эффективными, чем список смежности.

Перечисление компонентов пути

Один из недостатков списка смежности — высокие затраты на получение предков заданного узла дерева. В модели перечисления компонентов пути эта проблема решается хранением строки предков в атрибуте каждого узла.

Разновидность модели перечисления компонентов пути встречается в иерархиях каталогов. Путь UNIX (такой, как `/usr/local/lib/`) является перечислением компонентов пути файловой системы, где `usr` — родитель `local`, который, в свою очередь, является родителем `lib`.

В таблице `Comments` вместо столбца `parent_id` определяется столбец с именем `path` и длинным типом `VARCHAR`. Строка, хранящаяся в этом столбце, представляет последовательность предков текущей строки данных от начала дерева к концу, по аналогии с путем UNIX. Вы даже можете использовать символ `/` в качестве разделителя.

Trees/soln/path-enum/create-table.sql

```
CREATE TABLE Comments (
  comment_id SERIAL PRIMARY KEY,
  path VARCHAR(1000),
  bug_id BIGINT UNSIGNED NOT NULL,
  author BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  comment TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

comment_id	path	author	comment
1	1/	Fran	В чем причина этой ошибки?
2	1/2/	Ollie	Думаю, дело в нулевом указателе.
3	1/2/3/	Fran	Нет, я проверял.
4	1/4/	Kukla	Нужно провести проверку на неверный ввод.
5	1/4/5	Ollie	Да, это баг.
6	1/4/6	Fran	Да, добавь тест, пожалуйста.
7	1/4/6/7/	Kukla	Пофиксили!

Чтобы запросить предков узла, сравните путь текущей строки с шаблоном, образованным из пути другой строки. Например, поиск предков (ancestors) комментария № 7, путь которого имеет вид `1/4/6/7`, выполняется так:

Trees/soln/path-enum/ancestors.sql

```
SELECT *
FROM Comments AS c
WHERE '1/4/6/7/' LIKE CONCAT(c.path, '%');
```

Совпадение будет найдено для шаблонов, сформированных из путей предков `1/4/6/%`, `1/4/%` и `1/%`.

Чтобы запросить данные потомков (descendants), переставьте аргументы предиката `LIKE` в обратном порядке. Чтобы найти потомков комментария № 4 с путем 1/4/, выполните следующий фрагмент:

Trees/soln/path-enum/descendants.sql

```
SELECT *
FROM Comments AS c
WHERE c.path LIKE '1/4/%';
```

Шаблон `1/4/%` совпадает с путями потомков `1/4/5/`, `1/4/6/` и `1/4/6/7/`.

С появлением возможности быстро выделить подмножество дерева или цепочки предков до вершины дерева вы сможете легко выполнить много других запросов, например просуммировать затраты по узлам поддерева функцией `SUM()` или просто подсчитать количество узлов. Например, чтобы подсчитать комментарии по авторам в поддереве, начинающемся с комментария № 4, выполните следующий фрагмент:

Trees/soln/path-enum/count.sql

```
SELECT c.author, COUNT(*)
FROM Comments AS c
WHERE c.path LIKE '1/4/%'
GROUP BY c.author;
```

Вставка узла выполняется аналогично вставке в модели списка смежности. Вы можете вставить листовой узел без необходимости изменять любую другую запись. Скопируйте путь из родителя нового узла и присоедините к строке пути идентификатор нового узла. Если значение первичного ключа генерируется автоматически при вставке, возможно, вам придется сначала вставить запись, а потом обновить путь, когда вы уже знаете идентификатор новой строки. Например, при использовании MySQL встроенная функция `LAST_INSERT_ID()` возвращает последнее значение идентификатора, сгенерированное для вставленной строки в текущем сеансе. Оставшаяся часть пути берется из родителя нового узла.

Trees/soln/path-enum/insert.sql

```
INSERT INTO Comments (bug_id, author, comment)
VALUES (1234, 56 /* Ollie */, 'Good job!');

UPDATE Comments AS c
CROSS JOIN Comments AS c7 ON c7.comment_id = 7
  SET c.path = CONCAT(c7.path, LAST_INSERT_ID(), '/')
WHERE c.comment_id = LAST_INSERT_ID();
```

У модели перечисления компонентов пути есть свои недостатки, сходные с теми, которые были перечислены в главе 2 «Кривая дорожка». База данных не гарантирует, что путь сформирован корректно или что значения в пути соответствуют существующим узлам. Поддержание строки пути зависит от кода приложения, а ее проверка обходится дорого. С какой бы длиной вы ни объявили столбец `VARCHAR`, его длина все равно будет ограниченна, так что формально модель не поддерживает деревья неограниченной глубины.

Перечисление компонентов пути позволяет легко сортировать наборы строк по их иерархии — при условии, что элементы между разделителями имеют всегда одинаковую длину. С другой стороны, такое решение сильно напоминает антипаттерн «Кривая дорожка», поскольку это строка идентификаторов, разделенных специальным символом.

Вложенные множества

Модель вложенных множеств хранит с каждым узлом информацию, которая относится к множеству потомков, а не к непосредственному родителю узла. Для представления этой информации каждый узел в дереве может кодироваться двумя числами, которые мы назовем `nsleft` и `nsright`.

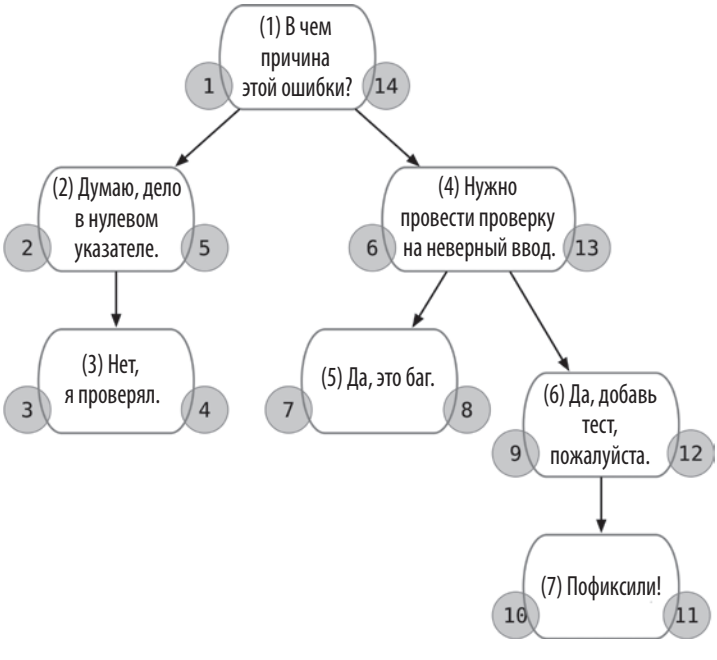
Trees/soln/nested-sets/create-table.sql

```
DROP TABLE IF EXISTS Comments;

CREATE TABLE Comments (
  comment_id SERIAL PRIMARY KEY,
```

```
  nsleft INTEGER NOT NULL,
  nsright INTEGER NOT NULL,
  bug_id BIGINT UNSIGNED NOT NULL,
  author BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  comment TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs (bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

Каждому узлу назначаются числа `nsleft` и `nsright` по следующей схеме: число `nsleft` меньше, чем у всех детей узла, тогда как число `nsright` больше, чем у всех детей узла. Эти числа не связаны со значениями `comment_id`.

Простой способ вычисления этих значений основан на обходе всех узлов дерева в глубину. Когда вы спускаетесь по левой стороне каждой ветви, присваивайте последовательные числовые значения `nsleft`. Когда вы поднимаетесь по правой стороне каждой ветви, присваивайте числа `nsright`. То же самое проделывается для каждой ветви слева направо. Следующая диаграмма поможет наглядно представить эту схему:



В следующей таблице каждый комментарий хранится в строке данных, содержащей столбцы `nsleft` и `nsright`. Значения задавались в порядке обхода в глубину.

comment_id	nsleft	nsright	author	comment
1	1	14	Fran	В чем причина этой ошибки?
2	2	5	Ollie	Думаю, дело в нулевом указателе.
3	3	4	Fran	Нет, я проверял.
4	6	13	Kukla	Нужно провести проверку на неверный ввод.
5	7	8	Ollie	Да, это баг.
6	9	12	Fran	Да, добавь тест, пожалуйста.
7	10	11	Kukla	Пофиксили!

Когда каждому узлу будут присвоены эти числа, их можно будет использовать для нахождения предков и потомков каждого заданного узла. Например, для получения комментария № 4 и его потомков проводится поиск узлов, у которых числа `nsleft` и `nsright` находятся между числами текущего узла.

Trees/soln/nested-sets/descendants.sql

```
SELECT c2.*
FROM Comments AS c1
  JOIN Comments as c2
    ON c2.nsleft BETWEEN c1.nsleft AND c1.nsright
WHERE c1.comment_id = 4;
```

Чтобы найти комментарий № 6 и его предков, проведите поиск узлов, у которых числа `nsleft` и `nsright` включают числа текущего узла. Пример:

Trees/soln/nested-sets/ancestors.sql

```
SELECT c2.*
FROM Comments AS c1
  JOIN Comments AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 6;
```

Главное преимущество модели вложенных множеств заключается в том, что при удалении нелистового узла его потомки автоматически становятся непосредственными потомками родителей удаленного узла. Хотя правое и левое число каждого узла на схеме образуют непрерывную последовательность и всегда отличаются от чисел соседних одноуровневых узлов и родителей на 1, модель не обязана сохранять эту иерархию. Таким образом, когда в результате удаления узла возникают разрывы в последовательности значений, это не нарушает работоспособность структуры.

Например, если подсчитать глубину заданного узла и удалить его родителя, а затем снова вычислить глубину узла, она уменьшится на один уровень.

Trees/soln/nested-sets/depth.sql

```
-- Выводимая глубина = 4
SELECT c1.comment_id, COUNT(c2.comment_id) AS depth
FROM Comments AS c1
  JOIN Comments AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 7
GROUP BY c1.comment_id;

DELETE FROM Comments WHERE comment_id = 6;
-- Выводимая глубина = 3
SELECT c1.comment_id, COUNT(c2.comment_id) AS depth
FROM Comments AS c1
  JOIN Comments AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 7
GROUP BY c1.comment_id;
```

Запросы, которые легко реализовать с использованием списка смежности (например, извлечение непосредственного потомка или непосредственного родителя), оказываются более сложными при реализации с вложенными множествами. Прямой родитель заданного узла `c1` является предком этого узла, но других узлов между ними быть не может. Следовательно, можно использовать дополнительное внешнее соединение для поиска узла, который одновременно является предком `c1` и потомком родителя. Только если такой узел не найден (то есть результат внешнего соединения равен `null`), предок действительно является прямым родителем `c1`.

Например, чтобы найти непосредственного родителя комментария № 6, выполните следующий фрагмент кода:

Trees/soln/nested-sets/parent.sql

```
SELECT c2.*
FROM Comments AS c1
  JOIN Comments AS c2
    ON c1.nsleft > c2.nsleft AND c1.nsleft < c2.nsright
  LEFT JOIN Comments AS c3
    ON c1.nsleft > c3.nsleft AND c1.nsleft < c3.nsright
    AND c3.nsleft > c2.nsleft AND c3.nsleft < c2.nsright
WHERE c1.comment_id = 6
  AND c3.comment_id IS NULL;
```

Операции с деревом, вставкой и перемещением узлов в решении с вложенными множествами обычно получаются более сложными, чем в других моделях. При

вставке нового узла необходимо заново вычислить все левые и правые значения, превышающие левое значение нового узла.

Это необходимо сделать с правыми одноуровневыми узлами нового узла, его предками и правыми одноуровневыми узлами его предков. Также пересчет необходимо выполнить для потомков, если новый узел вставляется как нелистовой. Если предположить, что новый узел является листовым, все необходимые обновления выполняются следующей командой:

Trees/soln/nested-sets/insert.sql

```
-- Выделить место для значений NS 8 и 9
UPDATE Comments
  SET nsleft = CASE WHEN nsleft >= 8 THEN nsleft+2 ELSE nsleft END,
    nsright = nsright+2
WHERE nsright >= 7;

-- Создать нового потомка комментария № 5, которому присваиваются
-- значения NS 8 и 9
INSERT INTO Comments (nsleft, nsright, bug_id, author, comment)
  VALUES (8, 9, 1234, 34 /* Fran */, 'Me too!');
```

Модель вложенных множеств лучше всего применять в случаях, когда быстрота и легкость выполнения запросов поддеревьев важнее операций с отдельными узлами. Операции вставки и перемещения узлов сложны из-за требования к пересчету левых и правых значений. Если ваша работа с деревом требует частых вставок, вложенные множества — не лучший вариант.

Таблица замыканий

Решения с таблицей замыканий служат простым и элегантным способом хранения иерархических данных. В них сохраняются все пути по дереву, а не только пути с непосредственными отношениями «родитель — ребенок».

Наряду с простой таблицей `Comments` создается таблица `TreePaths` с двумя столбцами, каждый из которых является внешним ключом для таблицы `Comments`.

Trees/soln/closure-table/create-table.sql

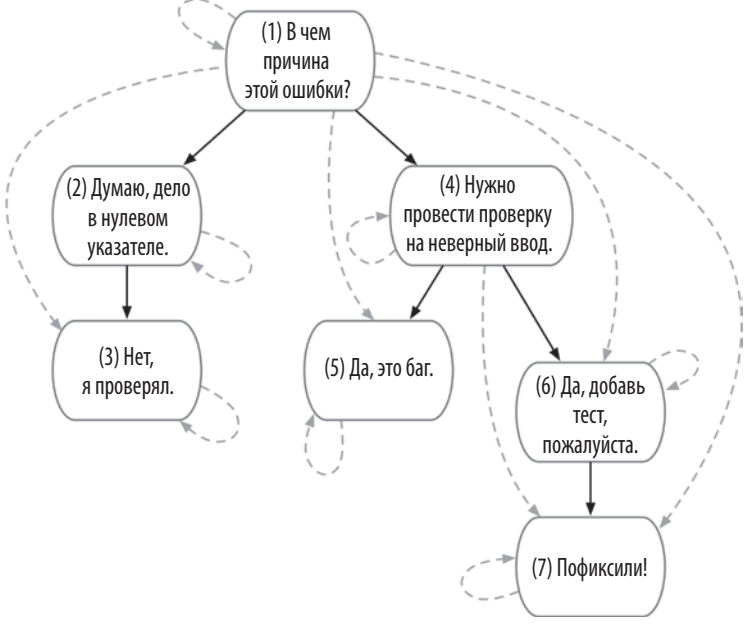
```
CREATE TABLE Comments (
  comment_id SERIAL PRIMARY KEY,
  bug_id BIGINT UNSIGNED NOT NULL,
  author BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,
  comment TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);

CREATE TABLE TreePaths (
```

```
  ancestor BIGINT UNSIGNED NOT NULL,
  descendant BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY(ancestor, descendant),
  FOREIGN KEY (ancestor) REFERENCES Comments(comment_id),
  FOREIGN KEY (descendant) REFERENCES Comments(comment_id)
);
```

Вместо таблицы `Comments` для хранения информации о древовидной структуре используется таблица `TreePaths`:

ancestor	descendant	ancestor	descendant	ancestor	descendant
1	1	1	7	4	6
1	2	2	2	4	7
1	3	2	3	5	5
1	4	3	3	6	6
1	5	4	4	6	7
1	6	4	5	7	7

В таблице сохраняется одна строка данных для каждой пары узлов дерева, объединенных отношениями «предок — потомок», даже если они разделены несколькими уровнями дерева. Также добавляется строка для каждого узла, которая ссылается на себя саму. Ниже приведена схема такого дерева.



Запросы на выборку предков и потомков из такой таблицы еще более просты, чем в решении с вложенными множествами. Чтобы извлечь потомков комментария № 4, найдите в `TreePaths` строки с предком (`ancestor`) 4:

Trees/soln/closure-table/descendants.sql

```
SELECT c.*
FROM Comments AS c
  JOIN TreePaths AS t ON c.comment_id = t.descendant
WHERE t.ancestor = 4;
```

Чтобы извлечь предков комментария № 6, найдите в `TreePaths` строки с потомком (`descendant`) 6:

Trees/soln/closure-table/ancestors.sql

```
SELECT c.*
FROM Comments AS c
  JOIN TreePaths AS t ON c.comment_id = t.ancestor
WHERE t.descendant = 6;
```

Чтобы вставить новый листовой узел (например, нового непосредственного потомка комментария № 5), сначала вставьте строку со ссылкой на саму себя. Затем добавьте в `TreePaths` копию набора строк данных, в которых комментарий № 5 указан как потомок (включая строку, в которой комментарий № 5 ссылается на самого себя), заменив в них потомка номером нового комментария:

Trees/soln/closure-table/insert.sql

```
INSERT INTO TreePaths (ancestor, descendant)
  SELECT t.ancestor, 8
  FROM TreePaths AS t
  WHERE t.descendant = 5
UNION ALL
  SELECT 8, 8;
```

Чтобы удалить листовой узел (например, комментарий № 7), удалите все строки в `TreePaths`, в которых комментарий № 7 указан как потомок:

Trees/soln/closure-table/delete-leaf.sql

```
DELETE FROM TreePaths WHERE descendant = 7;
```

Чтобы удалить полное поддерево (например, комментарий № 4 и его потомков), удалите все записи в `TreePaths`, в которых комментарий № 4 указан как потомок, а также все строки, в которых какой-либо из потомков комментария № 4 указан как потомок:

Trees/soln/closure-table/delete-subtree.sql

```
DELETE t1 FROM TreePaths AS t1
JOIN TreePaths AS t2 ON t1.descendant = t2.descendant
WHERE t2.ancestor = 4;
```

Обратите внимание: при удалении строк в `TreePaths` сами комментарии не удаляются. В примере с `Comments` это может показаться странным, но начинает выглядеть более логичным при работе с другими видами деревьев, например категориями в каталоге товаров или сотрудниками в схеме организации. В таких случаях изменение отношений узла с другими узлами не обязательно должно приводить к его удалению. Благодаря тому что пути хранятся в отдельной таблице, структура хранения данных становится более гибкой.

Чтобы переместить поддерево из одной позиции в другую, сначала отсоедините поддерево от предков, удалив строки, ссылающиеся на предков верхнего узла поддереве, а также потомков этого узла. Например, чтобы переместить комментарий № 6 из его текущей позиции непосредственного потомка комментария № 4 в позицию непосредственного потомка комментария № 3, начните со следующего удаления. Следите за тем, чтобы не удалить автоссылку комментария № 6.

Trees/soln/closure-table/move-subtree.sql

```
DELETE t1 FROM TreePaths AS t1
JOIN TreePaths AS t2 ON t1.descendant = t2.descendant
JOIN TreePaths AS t3 ON t1.ancestor = t3.ancestor
WHERE t2.ancestor = 6 AND t3.descendant = 6
  AND t3.ancestor != t3.descendant;
```

Выбирая предков № 6, но не сам комментарий № 6, и потомков № 6, включая № 6, инструкция удаляет все пути от предков № 6 к № 6 и его потомкам. Другими словами, инструкция удаляет пути (1, 6), (1, 7), (4, 6) и (4, 7). Она не удаляет пути (6, 6) или (6, 7).

Затем «зависшее» поддерево добавляется в иерархию посредством вставки строк, соответствующих предкам новой позиции и потомкам поддерева. Синтаксис `CROSS JOIN` может использоваться для создания декартова произведения, с генерированием строк, необходимых для сопоставления предков новой позиции со всеми перемещаемыми узлами поддерева.

Trees/soln/closure-table/move-subtree.sql

```
INSERT INTO TreePaths (ancestor, descendant)
  SELECT supertree.ancestor, subtree.descendant
  FROM TreePaths AS supertree
    CROSS JOIN TreePaths AS subtree
  WHERE supertree.descendant = 3
    AND subtree.ancestor = 6;
```

Инструкция создает новые пути с использованием предков № 3, включая № 3, и потомков № 6, включая № 6. Таким образом, создаются новые пути (1, 6), (2, 6), (3, 6), (1, 7), (2, 7), (3, 7). В результате поддерево, начинающееся с комментария № 6, перемещается в позицию непосредственного потомка комментария № 3. Перекрестное соединение создает все необходимые пути, даже если поддерево перемещается на более высокий или более низкий уровень дерева.

Решение с таблицей замыканий более прямолинейно, чем решение с вложенными множествами. Оба варианта позволяют быстро и просто запросить предков и потомков, но таблица замыканий упрощает хранение иерархической информации. В обоих случаях запрашивать непосредственных потомков или родителей удобнее, чем в решениях со списками смежности или перечислением компонентов пути.

Таблицу замыканий можно усовершенствовать, чтобы упростить запросы непосредственных родителей или потомков. Добавьте в структуру решения с таблицей замыканий атрибут `TreePaths.path_length`. У автоссылки узла значение `path_length` равно нулю, у его ребенка — 1, у внука — 2 и т. д. Поиск детей комментария № 4 значительно упрощается:

Trees/soln/closure-table/child.sql

```
SELECT *
FROM TreePaths
WHERE ancestor = 4 AND path_length = 1;
```

Какое решение использовать?

У каждого решения свои преимущества и недостатки. Выбирайте его исходя из того, для каких операций требуется наибольшая эффективность.

В следующей таблице указана сложность некоторых операций в соответствующем варианте решения.

Решение	Таблицы	Запрос детей	Дерево запроса	Вставка	Удаление	Ссылочная целостность
Список смежности	1	Просто	Сложно	Просто	Просто	Да
Рекурсивный запрос	1	Просто	Просто	Просто	Просто	Да
Перечисление компонентов пути	1	Просто	Просто	Просто	Просто	Нет
Вложенные множества	1	Сложно	Просто	Сложно	Сложно	Нет
Таблица замыканий	2	Просто	Просто	Просто	Просто	Да

Принимайте во внимание следующие сильные и слабые стороны каждого решения:

- *Список смежности* — традиционное решение, известное многим разработчикам. Его преимуществом перед другими решениями является нормализация. Иначе говоря, у него нет избыточности, и данные в нем не смогут конфликтовать.

  Рекурсивные запросы с `WITH` или `CONNECT BY PRIOR` повышают эффективность решения со списком смежности — при условии, что этот синтаксис поддерживается выбранной версией базы данных SQL.
- *Перечисление компонентов пути* хорошо подходит для «хлебных крошек» в пользовательских интерфейсах, но это решение слишком уязвимо, поскольку оно не обеспечивает ссылочную целостность и хранит избыточную информацию.
- *Вложенные множества* — умное решение... возможно, даже слишком. Кроме того, оно не обеспечивает ссылочную целостность. Лучше всего использовать его в случаях, когда запросы к дереву выполняются чаще, чем модификация дерева.
- *Таблица замыканий* — самое гибкое из альтернативных решений и единственное из описанных в этой главе, в котором узел может принадлежать сразу нескольким деревьям. Оно требует создания дополнительной таблицы для хранения отношений. Кроме того, кодирование глубоких иерархий требует большого количества строк данных, что повышает затраты памяти; это плата за сокращение вычислений. Как и многие денормализованные решения, оно обеспечивает хорошую производительность при выполнении некоторых видов запросов.

О хранении иерархических данных и работе с ними в SQL можно еще много узнать. Есть хорошая книга Джо Селко (Joe Celko), в которой описаны иерархические запросы, — «Trees and Hierarchies in SQL for Smarties» [Cel04]. Другая книга с описанием деревьев и даже графов — «QL Design Patterns»[Tro06] Вадима Тропашко (Vadim Tropashko). Она написана более формальным, академическим языком.

Иерархия состоит из объектов и отношений. Моделируйте их с учетом запросов, которые вам предстоит составлять к иерархии.

Мини-антипаттерн: на моем компьютере все работает

«Что означает эта ошибка? Мой запрос SQL нормально работает на моей машине, но когда я развертываю приложение на продакшен, возникает ошибка!»

```
Error: 1064 (42000): You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the right syntax to use
near 'WITH'
```

(Ошибка в синтаксисе SQL; обратитесь к документации своей версии MySQL Server за описанием синтаксиса, который должен использоваться с 'WITH')

В данном случае ошибка возникает из-за того, что разработчик написал запрос SQL с использованием ключевого слова `WITH` для обобщенного табличного выражения. В MySQL эта синтаксическая возможность была впервые реализована в версии 8.0. Если код попытается запустить этот запрос в более старой версии MySQL Server, он вызовет описанное выше сообщение об ошибке.

В среде разработки обновить версию ПО базы данных намного проще, чем в продакшен-среде. Многие компании не готовы рисковать простоями целых рабочих систем, тогда как обновление на рабочей станции или ноутбуке отдельного разработчика не создаст проблем.

В новых версиях любого программного продукта добавляются возможности, которых не было в более ранних версиях. В базе данных SQL может появиться расширенный синтаксис SQL, новые встроенные функции и даже новые типы данных. Новые средства SQL обычно мощные и привлекательные, но если вы пишете код, который от них зависит, после развертывания приложения на продакшен вас может ждать сюрприз: код разработки и рабочий код работают по-разному.

Такие сюрпризы обходятся дорого, и их трудно отлаживать. Возможно, придется переписывать большой объем кода, для которого предполагалось, что функциональность базы данных будет работать так же, как в копии, использованной при разработке.

Чтобы избежать этой проблемы, в локальной среде разработки необходимо использовать ту же версию ПО базы данных, что и в продакшене. Чем ближе эти две версии, тем лучше, потому что иногда даже некритичное обновление вводит новую функциональность, исправляет ошибки или нарушает обратную совместимость.

У серверов баз данных также имеется дополнительная функциональность, поэтому постарайтесь обеспечить максимальное соответствие сервера разработки серверу эксплуатации. В частности, по следующим критериям:

- параметрам конфигурации, влияющим на работу с данными или поведение запросов;
- параметрам конфигурации, влияющим на глобальные настройки по умолчанию;
- пользователям баз данных, ролям и привилегиям SQL (но лучшие практики безопасности не рекомендуют иметь одинаковые пароли в разных средах).

Стандартная практика — использовать одинаковые версии языка программирования, библиотек, фреймворков и других зависимостей в разработке и на продакшен. Это требование проверяется автоматически при построении и упаковке кода приложения. Тем не менее, несмотря на рост популярности контейнерного развертывания, ПО баз данных обычно не включается в развертываемые пакеты.

Оставшиеся снаружи животные переводили взгляд
со свиней на людей, с людей на свиней, но уже
не могли понять, кто есть кто.

➤ *Джордж Оруэлл, «Скотный двор»*

ГЛАВА 4

Обязательный ID

Недавно я отвечал на популярный вопрос разработчика, которому понадобилось предотвратить дублирование строк данных. Сначала я подумал, что в базе данных отсутствует первичный ключ. Однако проблема была в другом.

В базе данных управления контентом он сохранял статьи, предназначенные для публикации на веб-сайте. Он использовал таблицу пересечений для формирования отношений «многие ко многим» между таблицей статей и таблицей тегов.

ID-Required/intro/articletags.sql

```
CREATE TABLE ArticleTags (
  id SERIAL PRIMARY KEY,
  article_id BIGINT UNSIGNED NOT NULL,
  tag_id BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (article_id) REFERENCES Articles (id),
  FOREIGN KEY (tag_id) REFERENCES Tags (id)
);
```

Он получал некорректные результаты запросов для подсчета количества статей с заданным тегом. Разработчик знал, что с тегом «economy» связано только пять статей, но запрос сообщал ему, что таких статей семь.

ID-Required/intro/articletags.sql

```
SELECT tag_id, COUNT(*) AS articles_per_tag
FROM ArticleTags WHERE tag_id = 327;
```

Когда он запрашивал все строки данных для заданного значения `tag_id`, он видел, что тег был связан с одной конкретной статьей троекратно; для трех строк отображалась одна и та же связь, хотя они имели разные идентификаторы, как показано в таблице ниже.

id	tag_id	article_id
22	327	1234
23	327	1234
24	327	1234

Таблица содержит первичный ключ, но этот первичный ключ не предотвращал появления дубликатов в столбце, который был важен для запроса. Проблему можно было решить созданием ограничения `UNIQUE` по двум другим столбцам, но в этом случае столбец `id` вообще не требовался.

Цель: установление соглашений первичного ключа

Цель заключается в том, чтобы обеспечить наличие первичного ключа у каждой таблицы, однако путаница относительно природы первичного ключа привела к проявлению антипаттерна.

Каждый, кто изучал основы проектирования баз данных, знает, что первичный ключ — важная (и даже обязательная) часть таблицы. И это действительно так; первичные ключи абсолютно необходимы для качественного проектирования баз данных. Первичный ключ гарантированно будет уникален для всех строк таблицы, поэтому это логический механизм для адресации отдельных записей и для предотвращения хранения дубликатов записей. Первичный ключ также используется внешними ключами для создания связей между таблицами.

Достаточно непросто выбрать столбец, который служит первичным ключом. Значение любого атрибута в большинстве таблиц теоретически может принадлежать сразу нескольким строкам. Классические примеры — имя и фамилия человека — очевидным образом могут дублироваться. Даже адрес электронной почты или административные идентификаторы (например, номер социального страхования США или идентификационный номер налогоплательщика) не являются строго уникальными признаками.

В такие таблицы необходимо включить новый столбец для хранения искусственного значения, которое не имеет смысла в предметной области, моделируемой таблицей. Этот столбец используется как первичный ключ, что делает возможной уникальную адресацию строк, тогда как любые другие столбцы атрибутов могут содержать дубликаты при их наличии. Такая разновидность столбца первичного ключа иногда называется *псевдоключом* или *суррогатным ключом*.

Чтобы гарантировать, что строкам будут назначены уникальные значения псевдоключа даже тогда, когда клиенты одновременно добавляют новые

записи, многие базы данных предоставляют механизм для генерирования уникальных последовательных целых значений вне контекста транзакционной изоляции.

Ниже приведен пример стандартного синтаксиса SQL:2003 для псевдоключа (протестировано в PostgreSQL, но MySQL этот синтаксис не поддерживает):

ID-Required/obj/sql:2003.sql

```
CREATE TABLE Bugs (
  bug_id BIGINT GENERATED ALWAYS AS IDENTITY,
  summary VARCHAR(80)
  -- другие столбцы...
);
```

До того как псевдоключи были стандартизированы в SQL:2003, в каждой базе данных приходилось реализовывать для них собственное решение SQL. Однако терминология псевдоключей зависит от конкретного разработчика, как показано в следующей таблице:

Синтаксис	Поддерживается базами данных
`AUTO_INCREMENT`	MySQL
`GENERATOR`	Firebird, InterBase
`IDENTITY`	DB2, Derby, Microsoft SQL Server, Sybase
`ROWID`	SQLite
`SEQUENCE`	DB2, Firebird, Informix, Ingres, Oracle, PostgreSQL
`SERIAL`	MySQL, PostgreSQL
`SQL:2003 standard`	DB2, Derby, Firebird, Ingres, Oracle, PostgreSQL

Псевдоключи — полезная возможность, но это не единственное решение для объявления первичного ключа.

Антипаттерн: на любой случай жизни

Книги, статьи и фреймворки программирования установили соглашение, в соответствии с которым каждая таблица базы данных должна иметь столбец первичного ключа, обладающий следующими параметрами:

- Столбцу первичного ключа присваивается имя `id`.
- Столбец имеет 32-разрядный или 64-разрядный целый тип данных.
- Уникальные значения генерируются автоматически.

Столбцы с именем `id` присутствовали в каждой таблице, и это имя стало синонимом первичного ключа. Программисты, изучающие SQL, начали ошибочно думать, что первичный ключ всегда означает столбец, определенный таким образом.

ID-Required/anti/id-ubiquitous.sql

```
CREATE TABLE Bugs (
  id SERIAL PRIMARY KEY,
  description VARCHAR(1000),
  -- . . .
);
```

Добавление столбца `id` в каждую таблицу влечет за собой ряд последствий, из-за которых его использование начинает казаться не столь привлекательным.

Создание избыточного ключа

Иногда столбец `id` определяется в качестве первичного ключа просто по традиции, хотя в той же таблице имеется другой ключ, который может стать естественным первичным ключом. Другой столбец даже может быть определен с ограничением `UNIQUE`. Например, в таблице `Bugs` приложение может помечать ошибки строкой с мнемоникой проекта, к которому относится ошибка, или другим идентификационным признаком.

ID-Required/anti/id-redundant.sql

```
CREATE TABLE Bugs (
  id SERIAL PRIMARY KEY,
  bug_id VARCHAR(10) UNIQUE,
  description VARCHAR(1000),
  -- . . .
);

INSERT INTO Bugs (bug_id, description, ...)
  VALUES ('VIS-078', 'crashes on save', ...);
```

Столбец `bug_id` в предыдущем примере аналогичен `id` — в том смысле, что он служит уникальным признаком, однозначно идентифицирующим каждую строку.

Разрешение дубликатов

Составной ключ состоит из нескольких столбцов. Одно из типичных применений составного ключа встречается в таблицах пересечений, таких как `BugsProducts`. Первичный ключ должен гарантировать, что заданная комбинация значений `bug_id` и `product_id` встречается в таблице только один раз, несмотря на то что каждое значение может встречаться многократно в разных сочетаниях.

Однако при использовании обязательного столбца `id` в качестве первичного ключа ограничение уже не действует на два столбца, которые должны быть уникальными.

ID-Required/anti/superfluous.sql

```
CREATE TABLE BugsProducts (
  id SERIAL PRIMARY KEY,
  bug_id BIGINT UNSIGNED NOT NULL,
  product_id BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

INSERT INTO BugsProducts (bug_id, product_id)
  VALUES (1234, 1), (1234, 1), (1234, 1); -- дубликаты разрешены
```

Дубликаты в этой таблице пересечений приводят к непредвиденным результатам, когда вы используете таблицу для сопоставления `Bugs` с `Products`. Чтобы предотвратить появление дубликатов, можно объявить ограничение `UNIQUE` по двум столбцам без `id`:

ID-Required/anti/superfluous.sql

```
CREATE TABLE BugsProducts (
  id SERIAL PRIMARY KEY,
  bug_id BIGINT UNSIGNED NOT NULL,
  product_id BIGINT UNSIGNED NOT NULL,
  UNIQUE KEY (bug_id, product_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

Если вам в любом случае необходимо ограничение уникальности по этим двум столбцам, столбец `id` становится лишним.

Сокрытие смысла ключа

У слова «код» есть много определений, и одно из них связано с особой передачей сообщений, главной задачей которой становится компактность или секретность. Разработчик преследует обратную цель — сделать свой код более понятным. Имя `id` настолько общее, что оно теряет всякую содержательность. Это особенно важно, когда вы соединяете две таблицы с одноименными столбцами первичного ключа.

ID-Required/anti/ambiguous.sql

```
SELECT b.id, a.id
FROM Bugs b
JOIN Accounts a ON (b.assigned_to = a.id)
WHERE b.status = 'OPEN';
```

Вы не сможете определить идентификатор ошибки по идентификатору учетной записи в коде своего приложения, если вы ссылаетесь на столбцы по именам, а не по исходной позиции. Это создает проблемы, особенно если результат запроса возвращается в виде ассоциативного массива, документа JSON или динамического объекта, потому что значение столбца замещает другое значение с тем же именем. Чтобы этого не произошло, необходимо указать в запросе синонимы столбцов.

ID-Required/anti/ambiguous.sql

```
SELECT b.id AS bug_id, a.id AS account_id
FROM Bugs b
JOIN Accounts a ON (b.assigned_to = a.id)
WHERE b.status = 'OPEN';
```

Имя столбца `id` не проясняет смысл запроса. Клиенту проще читать результаты запроса со столбцами под именем `bug_id` и `account_id`. Первичный ключ адресует отдельные строки заданной таблицы, так что имя столбца должно давать представление о типе сущности, хранящейся в этой таблице.

Использование USING

Вы, должно быть, знакомы с синтаксисом соединений в SQL: ключевые слова `JOIN` и `ON` предшествуют выражению для нахождения соответствующих записей двух таблиц:

ID-Required/anti/join.sql

```
SELECT * FROM Bugs AS b JOIN BugsProducts AS bp ON (b.bug_id = bp.bug_id);
```

SQL поддерживает и более компактный синтаксис соединения между двумя таблицами. Если столбцы называются одинаково в обеих таблицах, приведенный выше запрос можно переписать следующим образом:

ID-Required/anti/join.sql

```
SELECT * FROM Bugs JOIN BugsProducts USING (bug_id);
```

Тем не менее, если все таблицы обязательно должны определять первичный псевдоключ с именем `id`, то столбец внешнего ключа в зависимой таблице не может использовать такое же имя, как первичный ключ, на который он ссылается. Вместо этого всегда приходится использовать более пространный синтаксис `ON`:

ID-Required/anti/join.sql

```
SELECT * FROM Bugs AS b JOIN BugsProducts AS bp ON (b.id = bp.bug_id);
```

Специальные контексты для последовательностей идентификаторов

Некоторые разработчики генерируют идентификатор для новой строки, взяв наибольшее текущее используемое значение и увеличив его на единицу:

```
SELECT MAX(bug_id) + 1 AS next_bug_id FROM Bugs;
```

Такое решение становится ненадежным, если несколько клиентов одновременно запрашивают следующее значение. Одно значение может использоваться разными клиентами. Возникает так называемая *ситуация гонки*.

Чтобы избежать ситуации гонки, необходимо блокировать одновременных пользователей на время чтения текущего максимума, а затем использовать его значение в новой строке. Для этого надо установить блокировку для всей таблицы — блокировки на уровне строк недостаточно. Блокировка уровня таблицы создает «узкое место», потому что одновременным пользователям приходится в очереди ожидать ее снятия.

Последовательности решают эту проблему выполнением операций вне контекста транзакций. Они никогда не выделяют одно значение разным клиентам и, как следствие, никогда не отменяют выделение значения независимо от того, было ли это значение закреплено в строке данных. Из-за того что последовательности работают подобным образом, несколько клиентов могут одновременно генерировать уникальные значения и быть уверенными, что они не будут одинаковыми.

Многие базы данных поддерживают функцию, которая возвращает последнее сгенерированное значение последовательности. Например, в MySQL эта функция называется `LAST_INSERT_ID()`, Microsoft SQL Server использует имя `SCOPE_IDENTITY()`, а Oracle — `SequenceName.CURRVAL()`.

Эти функции возвращают значение, сгенерированное во время текущего сеанса, даже если другие клиенты одновременно генерировали свои значения. Ситуация гонки при этом исключается.

Составные ключи усложняют код

Некоторые разработчики стараются не использовать составные ключи. Они считают, что они слишком сложны. Любое выражение, которое сравнивает ключ с другим ключом, должно сравнивать все столбцы. Внешний ключ, который ссылается на составной первичный ключ, сам должен быть составным

внешним ключом. При работе с внешними ключами приходится вводить больше символов.

Такие разработчики напоминают математика, который отказывается работать с двумерными или трехмерными координатами и вместо этого выполняет все вычисления так, словно все объекты существуют в одномерном линейном пространстве. Безусловно, такой подход заметно упростит многие геометрические и тригонометрические вычисления, но он не позволяет описать реальные объекты, с которыми вам нужно работать.

Действительно ли первичный ключ необходим?

Некоторые разработчики заявляют, что их таблицам не нужен первичный ключ. Иногда они тем самым желают избежать лишних затрат ресурсов на поддержание уникального индекса, либо в таблицах, с которыми они работают, нет столбцов, которые могут использоваться для этой цели.

Ограничение первичного ключа важно в случаях, когда требуется:

- предотвратить появление строк-дубликатов в таблице;
- ссылаться на отдельные строки данных в запросах;
- поддерживать ссылки внешних ключей.

Есть еще одна неочевидная причина для использования первичного ключа: репликация базы данных. Например, в MySQL при воспроизведении обновлений на базе строк в экземпляре реплицированной базы первичный ключ используется для эффективного применения изменений. Если таблица не имеет первичного ключа, то для каждой обновляемой строки приходится выполнять ее полное сканирование, а это очень плохо влияет на производительность.

Отказываясь от использования ограничений первичного ключа, вы создаете для себя лишнюю нагрузку: проверку дубликатов.

```
SELECT bug_id FROM Bugs GROUP BY bug_id HAVING COUNT(*) > 1;
```

Проверка должна выполняться достаточно часто, в противном случае дубликаты станут проблемой. Если вы обнаружите дубликат `bug_id` в приведенном выше запросе, вам придется проверить обе строки с этим `bug_id` и оставить одну из них. А если запрос находит тысячи дубликатов `bug_id`?

Таблица без первичного ключа — все равно что музыкальная коллекция без названий песен. Можно слушать музыку, но найти нужную песню и удалить дубликат из коллекции не получится.

Как распознать антипаттерн

Симптомы этого антипаттерна легко узнаваемы: таблицы используют слишком общее имя `id` для первичного ключа, при том что для выбора этого имени столбца вместо более содержательного нет практически никаких причин.

Следующие фразы также могут указывать на присутствие антипаттерна:

- «Не думаю, что в этой таблице нужен первичный ключ».

  Разработчик, который так говорит, путает *первичный ключ* с *псевдоключом*. В каждой таблице должно присутствовать *ограничение* первичного ключа для предотвращения дубликатов и идентификации отдельных строк. Возможно, такому разработчику стоит использовать естественный или составной ключ.

- «Откуда тут дубликаты связей “многие ко многим”»?

  Таблица пересечений для отношения «многие ко многим» должна объявить ограничение первичного ключа или, по крайней мере, ограничение *уникальности* ключа по множеству столбцов внешнего ключа.

- «Я читал, что в теории баз данных рекомендуется переместить значения в таблицу сопоставления и обращаться к ним по идентификатору. Но я не хочу так делать, потому что мне придется выполнять соединение каждый раз, когда мне понадобятся актуальные значения».

  Это распространенное неверное представление о концепции теории баз данных, называемой нормализацией, которая в реальности не имеет ничего общего с псевдоключами. Подробнее см. в приложении «Правила нормализации».

Допустимые применения антипаттерна

Хотя нет ничего плохого ни в использовании псевдоключа, ни в присваивании значений через механизм автоматически увеличиваемых целых чисел, не каждой таблице нужен псевдоключ, и никто не заставляет вас присваивать каждому псевдоключу имя `id`.

Некоторые объектно-реляционные фреймворки упрощают разработку, следуя принципу «соглашения важнее конфигурации». Согласно этому принципу, все таблицы должны определять свой первичный ключ одинаково: в виде целочисленного столбца псевдоключа с именем `id`. Если вы используете такой фреймворк, вам стоит соблюдать его соглашения, потому что это откроет вам доступ к другим его полезным функциям.

Псевдоключ может стать хорошим вариантом суррогата для естественного ключа, слишком длинного, чтобы быть практичным. Например, для таблицы,

в которой хранятся атрибуты файла в файловой системе, путь к файлу может стать хорошим естественным ключом, но индексирование по строковому столбцу такой длины будет достаточно затратным.

Аналогичным образом при создании таблиц для многозначных атрибутов подчиненная таблица, ссылающаяся на длинный составной первичный ключ своего родителя, должна добавить дополнительный столбец для своего первичного ключа. По мере того как ссылки образуют более длинные «цепочки» таблиц, первичный ключ может стать слишком длинным. В таких случаях псевдоключ становится единственным практичным решением.

Решение: ситуационное

Первичный ключ — ограничение, а не тип данных. Можно объявить первичный ключ по любому столбцу или набору столбцов при условии, что типы данных поддерживают индексирование. Столбцы первичного ключа также должны быть объявлены с `NOT NULL`, но многие реализации делают это автоматически. Также необходимо иметь возможность определить столбец как автоматически увеличиваемое целое число, не назначая его первичным ключом таблицы. Две концепции не зависят друг от друга.

Не позволяйте негибким соглашениям препятствовать созданию качественного дизайна.

Выбирайте содержательные имена

Выбирайте для первичного ключа значимое имя. Оно должно передавать тип сущности, идентифицируемой первичным ключом. Например, первичному ключу таблицы `Bugs` должно быть присвоено имя `bug_id`.

По возможности используйте во внешних ключах те же имена столбцов. Часто это означает, что имя первичного ключа в схеме должно быть уникальным; две таблицы не должны использовать одно и то же имя для своего первичного ключа, если только один не является также внешним ключом, ссылающимся на другой. Впрочем, существуют исключения. Иногда бывает уместно присвоить внешнему ключу имя, отличное от имени первичного ключа, на который он ссылается, например, чтобы оно лучше описывало природу связи.

ID-Required/soln/foreignkey-name.sql

```
CREATE TABLE Bugs (
  -- . . .
  reported_by BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
);
```

Существует отраслевой стандарт для описания соглашений об именах метаданных. Стандарт ISO/IEC 11179[1] предоставляет рекомендации для «управления классификационными схемами» в системах информационных технологий. Другими словами, таблицам и столбцам следует присваивать осмысленные имена. Как и большинство стандартов ISO, этот документ практически полностью непонятен, но Джо Селко применяет его к SQL на практике в своей книге SQL Programming Style [Cel05].

Действуйте нешаблонно

Объектно-реляционные фреймворки предполагают, что вы используете псевдоключ с именем `id`, но они также позволяют переопределить соглашение и объявить другое имя. В следующем примере используется Ruby on Rails:

ID-Required/soln/custom-primarykey.rb

```
class Bug < ActiveRecord::Base
  set_primary_key "bug_id"
end
```

Некоторые разработчики считают, что назначать столбец первичного ключа необходимо только при поддержке *унаследованных* баз данных, к которым неприменимы привычные соглашения. Вообще говоря, в новых проектах осмысленные имена столбцов также полезны.

Используйте естественные и составные ключи

Если таблица содержит атрибут, который заведомо уникален, отличен от `NULL` и может использоваться для идентификации строки, не обязательно добавлять псевдоключ исключительно ради традиции.

На практике атрибуты таблицы часто изменяются или неуникальны. Базы данных эволюционируют на протяжении жизненного цикла проекта, и лица, принимающие решения, могут пренебрегать неприкосновенностью естественного ключа. Иногда вдруг выясняется, что столбец, который казался подходящим для естественного ключа, содержит дубликаты. В таких случаях псевдоключ остается единственным выходом.

Используйте составные ключи там, где это уместно. Если столбец лучше всего идентифицируется комбинацией нескольких столбцов атрибутов, как в таблице `BugsProducts`, используйте эти столбцы как составной первичный ключ.

---

[1] https://www.iso.org/standard/74570.html

ID-Required/soln/compound.sql

```
CREATE TABLE BugsProducts (
  bug_id BIGINT UNSIGNED NOT NULL,
  product_id BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (bug_id, product_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

INSERT INTO BugsProducts (bug_id, product_id)
  VALUES (1234, 1), (1234, 2), (1234, 3);

INSERT INTO BugsProducts (bug_id, product_id)
  VALUES (1234, 1); -- ошибка: дублирование элемента
```

Обратите внимание: внешний ключ, который ссылается на составной первичный ключ, также должен быть составным. Хотя дублирование этих столбцов в зависимых таблицах может показаться громоздким, у него есть свои преимущества: оно позволяет упростить запрос, который потребовал бы соединения для получения атрибутов указанной строки.

Соглашения хороши только тогда, когда они полезны.

Мини-антипаттерн: хватит ли BIGINT?

«А если `id` с автоувеличением достигнет максимального значения?»

Если приложение работает под высокой нагрузкой и ежедневно вставляет большой объем данных в таблицу, использующую столбец `id` с автоувеличением в качестве первичного ключа, у разработчика возникает резонный вопрос: что произойдет, если закончатся все целые значения?

Чтобы преодолеть эту неопределенность, стоит заняться математикой. Оцените количество строк, которые приложение вставляет в таблицу за минуту. Допустим, это 10 000 строк в минуту. Разделите максимальное целое значение на оцениваемое количество строк в минуту.

Максимальное значение 32-разрядного целого со знаком равно $2^{31} - 1$, или 2 147 483 647. При скорости 10 000 строк в минуту, если предположить, что `id` увеличивается на 1 для каждой строки, все числа будут исчерпаны за 214 749 минут, или 149 дней 3 часа и 9 минут. Получается меньше полугода, так что обычный столбец `INT` с очевидностью недостаточно велик.

При переходе на `INT` без знака (предполагается, что в столбце не будут использоваться отрицательные значения) используется тот же объем памяти (32 бита), но диапазон значений удваивается. Вместо 149 дней приложение проработает 298 дней — все равно меньше года при оценке темпа роста.

Хватит ли `BIGINT`?

Тип `BIGINT` большой. Очень большой. Максимальное значение 64-разрядного числа со знаком вовсе не вдвое больше 32-разрядного, это *квадрат* 32-разрядного числа. Таким образом, оно в 2^{32} раза больше.

Максимальное значение 64-разрядного целого числа со знаком равно 9 223 372 036 854 775 807. Теперь оценим, сколько времени потребуется на использование всех возможных значений. Разделим максимальное значение на 10 000 значений `id` в минуту. Затем разделим на 60 минут в час, затем на 24 часа в день и на 365 дней в год. Получается 1 754 827 252 года.

Разработчики спрашивали меня, что делать, если все значения `BIGINT` будут исчерпаны. Если предположить, что столбец с автоувеличением начинается с 1 и увеличивается на 1 в каждой следующей строке, я им отвечаю: «Я гарантирую, что пока приложение используется, значения `BIGINT` не закончатся. Если же это все-таки произойдет, я верну деньги за консультацию».

Воины-победители сперва побеждают и только потом вступают в битву; те же, что терпят поражение, сперва вступают в битву и только затем пытаются победить.

➤ *Сунь Цзы*

ГЛАВА 5

Сущность без ключа

«Билл, похоже, два менеджера зарезервировали один сервер в нашей лаборатории на одно и то же время, — как такое могло произойти? — написал мне руководитель лаборатории тестирования. — Разберись, пожалуйста. Они оба кричат, что им нужно оборудование и что я срываю график их проектов».

Несколько лет назад я написал приложение для учета оборудования на MySQL. По умолчанию MySQL использовал механизм хранения данных MyISAM, который не поддерживает ограничения внешнего ключа. База данных содержит много логических отношений, но не гарантирует ссылочной целостности.

Со временем проект развивался, способ обработки данных в приложении изменился, и это повлекло проблемы: когда нарушалась ссылочная целостность, в отчетах возникали расхождения, промежуточные итоги не суммировались, а в расписаниях появилось двойное резервирование.

Менеджер проекта попросил меня написать скрипты контроля качества, которые должны были периодически запускаться для обнаружения расхождений. Эти скрипты анализировали состояние базы данных, находили ошибки (например, осиротевшие строки в дочерних таблицах) и отправляли отчет по электронной почте.

Эти скрипты должны были проверять каждое отношение в таблице. С ростом объема данных и количества таблиц также увеличивалось количество запросов контроля качества, а выполнение скриптов занимало больше времени. Увеличился и размер отчетов, отправляемых по электронной почте.

Конечно, решение со скриптом работало, но по сути, оно было дорогостоящим изобретением велосипеда. Нужен был механизм, инициирующий *ранний сбой* приложения каждый раз, когда пользователь отправлял некорректные данные. Эту задачу решали ограничения внешнего ключа.

Цель: упрощение архитектуры базы данных

В проектировании реляционной базы данных отношения между таблицами почти настолько же важны, как сами таблицы. *Целостность ссылок* — важное условие проектирования баз данных и работы с ними. Когда вы объявляете ограничение внешнего ключа для столбца или набора столбцов, значения в этих столбцах должны существовать в столбце первичного ключа или в столбцах уникального ключа родительской таблицы. Вроде бы все просто.

Однако некоторые разработчики рекомендуют избегать ограничений ссылочной целостности и приводят следующие доводы для игнорирования внешних ключей:

- Обновления данных могут конфликтовать с ограничениями полей.
- Используемая архитектура базы данных настолько гибкая, что не поддерживает ограничения ссылочной целостности (см. главу 7 «Полиморфная связь»).
- Разработчики считают, что индекс, который база данных создает для внешнего ключа, влияет на производительность.
- Используемая база данных не поддерживает внешние ключи.
- Разработчикам приходится искать синтаксис объявления внешних ключей.

Антипаттерн: без ограничений

Хотя на первый взгляд может показаться, что отказ от ограничений внешнего ключа делает архитектуру базы данных более простой, гибкой или быстрой, за это приходится расплачиваться за счет других характеристик. Вы только нагружаете себя лишней работой, потому что код обеспечения ссылочной целостности придется писать вручную.

Предположение о безупречности кода

Многие разработчики для решения проблемы ссылочной целостности пишут код приложения так, чтобы отношения данных всегда соблюдались. Каждый раз при вставке строки данных необходимо проверять, что значения столбцов внешнего ключа ссылаются на существующие значения связанной таблицы. Каждый раз при удалении строки данных необходимо проверять, что все дочерние таблицы тоже обновляются соответствующим образом. То есть соблюдайте простое правило: не допускайте ошибок.

Чтобы избежать ошибок ссылочной целостности при отсутствии ограничений внешнего ключа, необходимо выполнять дополнительные запросы `SELECT` при внесении изменений. Эти запросы помогут убедиться в том, что изменение не

приведет к нарушению ссылок. Например, чтобы вставить новую строку, сначала проверьте, что родительская строка существует:

Keyless-Entry/anti/insert.sql

```
SELECT account_id FROM Accounts WHERE account_id = 1;
```

Затем можно добавить строку ошибки, которая ссылается на нее:

Keyless-Entry/anti/insert.sql

```
INSERT INTO Bugs (reported_by) VALUES (1);
```

Чтобы удалить существующую строку, нужно проверить, что у нее нет строк-потомков:

Keyless-Entry/anti/delete.sql

```
SELECT bug_id FROM Bugs WHERE reported_by = 1;
```

После этого можно удалить учетную запись:

Keyless-Entry/anti/delete.sql

```
DELETE FROM Accounts WHERE account_id = 1;
```

Если пользователь со значением `account_id` `1` вмешается и введет новую ошибку после вашего запроса, но до удаления учетной записи, удаление будет заблокировано. Такая ситуация может показаться маловероятной, но Гордон Летвин (Gordon Letwin), архитектор DOS 4, однажды сказал знаменитую фразу: «Один случай из миллиона — значит, в следующий вторник». При этом в базе данных остается неработающая ссылка — ошибка, о которой сообщает несуществующая учетная запись.

Единственное решение — явное установление блокировки таблицы `Bugs` на время ее проверки и снятие блокировки после завершения удаления учетной записи. Любая архитектура, требующая подобных блокировок, неизбежно будет испытывать проблемы в условиях, когда необходима высокая степень параллелизма и масштабируемости.

Проверка на ошибки

Антирешение, описанное в начале этой главы, использует скрипты, созданные разработчиком и сообщающие о нарушениях целостности данных.

Например, в нашей базе данных ошибок столбец `Bugs.status` ссылается на таблицу поиска `BugStatus`. Чтобы найти ошибки с недопустимым значением статуса, можно воспользоваться запросом следующего вида:

Keyless-Entry/anti/find-orphans.sql

```
SELECT b.bug_id, b.status
FROM Bugs b LEFT OUTER JOIN BugStatus s
  ON (b.status = s.status)
WHERE s.status IS NULL;
```

Нетрудно представить, что придется писать аналогичные запросы для каждой ссылочной связи в базе данных.

Если вы заметили, что вам часто приходится искать неработающие ссылки подобным образом, возможно, стоит решить, с какой частотой должны выполняться такие проверки. Если выполнять их слишком часто, это может повлиять на производительность рядовых запросов к базам данных. Если выполнять их слишком редко, в них появятся аномалии данных и это может повлиять на другую работу, прежде чем вы заметите и исправите их.

Если вы обнаружите неработающую ссылку, вам придется ее исправить. Например, заменить недействительное значение статуса ошибки приемлемым значением по умолчанию.

Keyless-Entry/anti/set-default.sql

```
UPDATE Bugs SET status = DEFAULT WHERE status = 'BANANA';
```

Неизбежно будут возникать и другие ситуации, в которых вы не сможете сгенерировать данные для исправления подобных ошибок. Например, столбец `Bugs.reported_by column` должен ссылаться на учетную запись пользователя, который сообщил об ошибке. В этом случае неясно, учетную запись какого пользователя следует выбрать в качестве замены.

«Я не виноват!»

Едва ли весь код, работающий с базой данных, можно написать идеально. Скорее всего, похожие обновления будут применяться к нескольким функциям приложения. Но если код потребуется изменить, не так просто гарантировать, что совместимые изменения были применены во всех точках приложения.

А может быть, к вашей базе данных обращаются приложения и скрипты, которые были написаны кем-то другим. И вы не можете быть уверены, что их авторы вносят изменения корректно.

Также вносить изменения напрямую в базу данных могут пользователи с помощью средства запросов SQL или приватных сценариев. Такие произвольные команды SQL легко могут привести к появлению неработающих ссылок. Скорее всего, в какой-то момент жизненного цикла приложения это произойдет.

База данных должна быть *целостной*, другими словами, ссылки в базе данных должны оставаться корректными после каждого обновления.

Порочный круг обновлений

Многие разработчики стараются обходиться без ограничений внешнего ключа, потому что ограничения усложняют обновление связанных столбцов в нескольких таблицах. Например, если вам понадобится удалить строку данных, от которой зависят другие строки, вам придется сначала удалить дочерние строки, чтобы избежать нарушения ограничений внешнего ключа:

Keyless-Entry/anti/delete-child.sql

```
DELETE FROM BugStatus WHERE status = 'BOGUS'; -- ОШИБКА!
DELETE FROM Bugs WHERE status = 'BOGUS';
DELETE FROM BugStatus WHERE status = 'BOGUS'; -- повторная попытка успешна
```

Необходимо вручную выполнить несколько команд, по одной для каждой дочерней таблицы. Если в будущем понадобится добавить еще одну дочернюю таблицу в базу данных, то код придется исправлять, чтобы удаление выполнялось и с новой таблицей. Впрочем, у этой проблемы есть решение.

Неразрешимая проблема возникает, когда вы обновляете столбец, от которого зависят дочерние строки. Дочерние строки невозможно обновить до обновления родителя, а родителя невозможно обновить до обновления дочерних значений, которые на него ссылаются. Оба изменения необходимо вносить одновременно, но это невозможно сделать в двух разных обновлениях. Возникает своего рода порочный круг.

Keyless-Entry/anti/update-catch22.sql

```
UPDATE BugStatus SET status = 'INVALID' WHERE status = 'BOGUS'; -- ОШИБКА!

UPDATE Bugs SET status = 'INVALID' WHERE status = 'BOGUS'; -- ОШИБКА!
```

Некоторые разработчики считают, что такими сценариями слишком сложно управлять, и поэтому предпочитают вообще не использовать внешние ключи. Однако позже вы увидите, как легко и эффективно решить проблемы многотабличных обновлений и удалений с помощью внешних ключей.

Как распознать антипаттерн

Следующие фразы от коллег по команде могут указывать на применение антипаттерна «Сущность без ключа»:

- «Как проверить, что значение существует в одной таблице, но не существует в другой?»

  Обычно это делается для поиска дочерних строк, родитель которых был обновлен или удален.

- «Можно ли быстро проверить, что значение существует в одной таблице как часть вставки во вторую таблицу?»

  Так проверяется существование родительской строки. Внешний ключ делает это за вас автоматически, используя индекс родительской таблицы, чтобы проверка была максимально эффективной.
- «Внешние ключи? Мне сказали, что их лучше не использовать, потому что они замедляют работу с базой данных».

  Производительность часто используется как оправдание для поиска обходных путей, но обычно они создают больше проблем, чем решают, в том числе проблемы с производительностью.

Некоторые структуры баз данных несовместимы с использованием внешних ключей. Если у вас не получается использовать традиционные ограничения ссылочной целостности, это может сигнализировать о том, что вы применяете антипаттерн SQL. Подробнее см. в главе 6 «Сущность — атрибут — значение» и в главе 7 «Полиморфная связь».

Допустимые применения антипаттерна

Некоторые реализации SQL требуют дополнительных блокировок родительской строки данных при обновлении ссылающихся на нее дочерних строк. Суть в том, чтобы предотвратить обновление или удаление родительской строки во время обновления зависимой строки. Это может оказаться неожиданным для некоторых разработчиков, и они могут сознательно отказаться от ограничения внешнего ключа, чтобы избежать таких блокировок.

Хотя ограничения внешнего ключа эффективно сокращают аномалии в данных, их можно снять, чтобы упростить чистку данных. Иначе говоря, заведомо осиротевшие строки в таблице могут быть разрешены, потому что руководство все еще разбирается с данными, необходимыми для создания ссылок.

Снятие ограничений внешнего ключа может упрощать и другие операции с базами данных. Например, популярный открытый инструмент автоматизации модификации таблиц MySQL `pt-online-schema-change`[1] переименовывает таблицы, вместо того чтобы изменять их. Если ссылки на старый формат измененной таблицы используются во внешних ключах других таблиц, их необходимо удалить и определить заново после переименования таблицы. Это может потребовать внесения изменений во многие таблицы вместо одной, в которой они были нужны. Чтобы избежать этого, некоторые разработчики решают отказаться от ограничений внешнего ключа.

---

[1] https://www.percona.com/doc/percona-toolkit/LATEST/pt-online-schema-change.html

Иногда приходится использовать базу данных, которая не поддерживает ограничения внешнего ключа (например, механизм хранения MyISAM в MySQL или SQLite до версии 3.6.19). В таком случае, возможно, вам придется искать компромиссные решения, например, скрипты контроля качества, описанные в преамбуле этой главы.

Решение: объявление ограничений

Японское выражение *poka-yoke* («пока-ёкэ») означает «защита от дурака». Его ввел инженер-технолог Сигэо Синго в своем исследовании производственной системы Toyota. Этот термин обозначает производственный процесс, в котором дефекты минимизируются благодаря предотвращению ошибок, их исправлению и привлечению внимания к ним. Такая практика улучшает качество и снижает необходимость в исправлениях, что более чем компенсирует затраты на ее применение.

Принцип poka-yoke можно применить при проектировании баз данных, используя ограничения внешнего ключа для обеспечения ссылочной целостности. Вместо того чтобы искать и исправлять ошибки целостности данных, можно с самого начала предотвратить появление этих ошибок в базе данных.

Keyless-Entry/soln/foreign-keys.sql

```
CREATE TABLE Bugs (
  -- . . .
  reported_by BIGINT UNSIGNED NOT NULL,
  status VARCHAR(20) NOT NULL DEFAULT 'NEW',
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (status) REFERENCES BugStatus(status)
);
```

Существующий код и ситуативные запросы подчиняются одним и тем же ограничениям, что полностью исключает забытый код или «черные ходы» для обхода требований. База данных отклоняет все некорректные изменения независимо от их источника.

Внешние ключи избавляют разработчиков от написания лишнего кода и гарантируют, что весь код будет работать одинаково при изменении базы данных. Это снижает время разработки и экономит много часов отладки и сопровождения. Средняя частота ошибок в отрасли составляет от 15 до 50 на 1000 строк кода. При прочих равных условиях чем меньше объем кода, тем меньше в нем ошибок.

Поддержка многотабличных изменений

Внешние ключи предоставляют возможность, которую не получится реализовать в коде приложения: *каскадные обновления*.

Keyless-Entry/soln/cascade.sql

```
CREATE TABLE Bugs (
  -- . . .
  reported_by       BIGINT UNSIGNED NOT NULL,
  status            VARCHAR(20) NOT NULL DEFAULT 'NEW',
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
    ON UPDATE CASCADE
    ON DELETE RESTRICT,
  FOREIGN KEY (status) REFERENCES BugStatus(status)
    ON UPDATE CASCADE
    ON DELETE SET DEFAULT
);
```

Это решение позволяет обновить и удалить родительскую строку, при этом база данных обработает любые дочерние строки, которые ссылаются на нее. Обновления родительских таблиц `BugStatus` и `Accounts` автоматически распространяются на дочерние строки `Bugs`. Порочный круг разрывается.

Способ объявления условий `ON UPDATE` или `ON DELETE` в ограничении внешнего ключа позволяет управлять результатом каскадной операции. Например, `RESTRICT` для внешнего ключа по `reported_by` означает, что не получится удалить учетную запись, если какие-то строки в `Bugs` ссылаются на нее. Ограничение блокирует удаление и инициирует ошибку. С другой стороны, при удалении значения `status` любые ошибки с этим статусом автоматически сбрасываются на значение статуса по умолчанию.

В любом случае база данных изменяет обе таблицы атомарно. Ссылки внешнего ключа остаются целостными как до, так и после изменений.

При добавлении новой дочерней таблицы в базу данных внешние ключи дочерней таблицы определяют каскадное поведение. Изменять код приложения не нужно. Также не нужно ничего менять в родительской таблице, сколько бы дочерних таблиц на нее ни ссылалось.

Накладные расходы? На самом деле нет

Действительно, ограничения внешнего ключа подразумевают некоторые накладные расходы. Но даже несмотря на это, они намного более эффективны, чем их альтернативы.

- Не нужно выполнять проверочные запросы `SELECT` перед вставкой, обновлением или удалением.
- Не нужно блокировать таблицы для защиты многотабличных изменений.
- Не нужно периодически запускать скрипты контроля качества для исправления неизбежных осиротевших строк.

Внешние ключи просты в использовании, они повышают производительность и помогают обеспечить ссылочную целостность при любых изменениях данных, как простых, так и сложных.

Подробнее о предотвращении типичных ошибок при определении внешних ключей см. в главе 26 «Ошибки внешних ключей в стандартном SQL» и главе 27 «Ошибки внешних ключей в MySQL».

Ограничения помогают защитить базу данных от ошибок.

Если вы попытаетесь разобрать кошку, чтобы посмотреть, как она работает, первое, что у вас будет в руках, — это неработающая кошка.

➤ *Дуглас Адамс*

ГЛАВА 6

Сущность — атрибут — значение

«Как подсчитать количество строк с заданной датой?» — очень простая задача для разработчика баз данных. Решение приводится в любом учебнике SQL вводного уровня. В нем используется базовый синтаксис SQL:

EAV/intro/count.sql

```
SELECT date_reported, COUNT(*)
FROM Bugs
GROUP BY date_reported;
```

Простое решение предполагает два условия:

- Дата ошибки хранится только в столбце с именем `Bugs.date_reported` во всех строках таблицы.
- Значения могут сравниваться друг с другом, чтобы конструкция `GROUP BY` могла точно группировать данные с равными значениями.

Не всегда эти предположения верны. Дата может храниться в столбце `date_reported`, или `report_date column`, или в столбце с любым другим именем. Имя может отличаться в каждой строке, поскольку разработчики могли именовать атрибут по-разному в разные дни. Данные могут храниться во множестве разных форматов, что не позволит их сравнить.

Эти и другие проблемы возникают при применении антипаттерна, известного под названием «Сущность — атрибут — значение».

Цель: поддержка переменных атрибутов

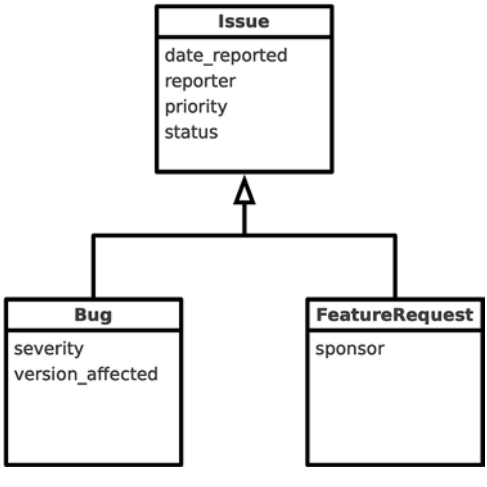
Частая цель программных проектов — расширяемость. Всегда хочется создавать гибкие продукты, адаптируемые к будущему использованию с минимумом дополнительного программирования (или вообще без него).

Эта проблема не нова: похожие аргументы против негибкости метаданных реляционных БД выдвигались почти постоянно с 1970 года, когда реляционная модель была впервые предложена Коддом (E. F. Codd) в статье *A Relational Model of Data for Large Shared Data Banks* [Cod70].

Традиционная таблица состоит из столбцов атрибутов, относящихся к каждой строке таблицы, так как каждая строка представляет экземпляр похожего объекта. Разные наборы атрибутов представляют разные типы объектов и, следовательно, должны принадлежать разным таблицам.

Однако в современных моделях объектно-ориентированного программирования разные типы объектов могут быть связаны: например, они могут расширять один базовый тип. В объектно-ориентированном проектировании такие объекты считаются экземплярами одного базового типа, а также экземплярами соответствующих подтипов. Объекты желательно хранить в строках одной таблицы базы данных, чтобы упростить сравнения и вычисления с несколькими объектами. Также необходимо разрешить объектам каждого подтипа хранить соответствующие столбцы атрибутов, которые могут не относиться к базовому типу или другим подтипам.

Рассмотрим пример из базы данных ошибок. `Bug` и `FeatureRequest` имеют ряд общих атрибутов, которые определены в базовом типе `Issue` на следующей диаграмме классов.

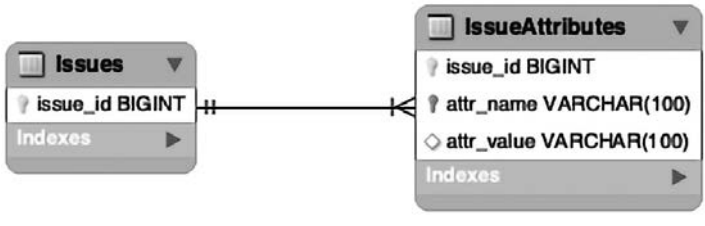


Каждая проблема (`Issue`) связана с человеком, который сообщил о ней. Она также связана с определенным продуктом, и ей назначен приоритет завершения. Объект ошибки (`Bug`) имеет свои атрибуты: версию продукта, в которой произошла ошибка, а также серьезность ошибки. Аналогичным образом `FeatureRequest`

(запрос на добавление функции) имеет собственные атрибуты, например спонсора, бюджет которого обеспечивает разработку этой функции.

Антипаттерн: использование обобщенной таблицы атрибутов

Некоторые разработчики, когда им требуется хранить переменные наборы атрибутов, решают создать вторую таблицу и хранить в ней атрибуты в виде строк данных. Как видно из следующей диаграммы, поначалу такое решение кажется более простым, поскольку позволяет использовать две таблицы вместо трех (или более, если ваша модель данных более вариативна).



Каждая строка таблицы атрибутов состоит из трех столбцов:

- *Сущность*. Обычно это внешний ключ к родительской таблице, которая содержит одну строку для каждой сущности.
- *Атрибут*. В традиционной таблице это просто имя столбца, но в новой архитектуре приходится идентифицировать атрибут для каждой отдельной записи.
- *Значение*. Сущность содержит значение для каждого из ее атрибутов.

Например, ошибка становится сущностью, которая идентифицируется значением первичного ключа `1234`. Она содержит атрибут с именем `status`. Значение этого атрибута для ошибки `1234` равно `NEW`.

Такая архитектура называется архитектурой «сущность — атрибут — значение» (Entity-Attribute-Value), или, сокращенно, EAV. Также иногда встречаются термины *открытая схема*, *бессхемная модель* или *пары «имя — значение»*.

EAV/anti/create-eav-table.sql

```
CREATE TABLE Issues (
  issue_id SERIAL PRIMARY KEY
);

INSERT INTO Issues (issue_id) VALUES (1234);

CREATE TABLE IssueAttributes (
```

```
  issue_id BIGINT UNSIGNED NOT NULL,
  attr_name VARCHAR(100) NOT NULL,
  attr_value VARCHAR(100),
  PRIMARY KEY (issue_id, attr_name),
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id)
);

INSERT INTO IssueAttributes (issue_id, attr_name, attr_value)
  VALUES
    (1234, 'product', '1'),
    (1234, 'date_reported', '2009-06-01'),
    (1234, 'status', 'NEW'),
    (1234, 'description', 'Saving does not work'),
    (1234, 'reported_by', 'Bill'),
    (1234, 'version_affected', '1.0'),
    (1234, 'severity', 'loss of functionality'),
    (1234, 'priority', 'high');
```

Казалось бы, добавление всего одной таблицы дает целый ряд преимуществ:

- Обе таблицы содержат меньше столбцов.
- Для поддержки новых атрибутов не нужно увеличивать количество столбцов.
- Предотвращается появление столбцов, содержащих `NULL` в строках, для которых данный атрибут неприменим.

Вроде архитектура стала лучше. Тем не менее простота структуры базы данных не компенсирует сложностей с ее использованием.

Запрос атрибутов

Вашему руководителю нужен ежедневный отчет с информацией об ошибках, зарегистрированных по дням. В традиционной архитектуре таблица `Issues` содержала бы простой столбец атрибута, например `date_reported`. Чтобы запросить все ошибки с датами их регистрации, можно использовать простой запрос следующего вида:

EAV/anti/query-plain.sql

```
SELECT issue_id, date_reported FROM Issues;
```

Чтобы получить ту же информацию в архитектуре EAV, придется выбрать строки из таблицы `IssueAttributes`, в которой хранится атрибут, имя которого определяется строкой `date_reported`. Запрос становится более длинным и менее понятным.

EAV/anti/query-eav.sql

```
SELECT issue_id, attr_value AS "date_reported"
FROM IssueAttributes
WHERE attr_name = 'date_reported';
```

Поддержка целостности данных

При использовании EAV теряются многие преимущества традиционной архитектуры баз данных.

Невозможно создать обязательные атрибуты

Чтобы помочь руководителю сгенерировать точный отчет, необходимо, чтобы атрибут `date_reported` содержал значение. В традиционной архитектуре баз данных достаточно просто определить столбец с обязательным значением, объявив его с `NOT NULL`.

В архитектуре EAV каждый атрибут соответствует строке в таблице `IssueAttributes`, а не столбцу. Вам понадобится ограничение, которое проверяет, что для каждого значения `issue_id` существует строка данных и эта строка данных содержит строку `date_reported` в столбце `attr_name`.

SQL не поддерживает такие ограничения. Следовательно, нужно написать для него отдельный код приложения. Также придется написать код, проверяющий при каждом чтении сущности, что ошибка содержит дату регистрации, потому что другой клиент мог сохранить ошибку без даты. Если вы найдете ошибку без даты, ее нужно будет исправить (еще больше кода), но вы не можете определить, каким было верное значение. Если вы попробуете угадать или используете значение по умолчанию для отсутствующего атрибута, это повлияет на точность отчета для руководителя.

Невозможно использовать типы данных SQL

Шеф говорит, что у него проблемы с отчетом, потому что разные люди вводят даты в разных форматах, а иногда даже вводят строки, которые вообще не являются датами. В традиционной базе данных это можно предотвратить объявлением столбца с типом данных `DATE`.

EAV/anti/insert-plain.sql

```
INSERT INTO Issues (date_reported) VALUES ('banana'); -- ОШИБКА!
```

В архитектуре EAV столбец `IssueAttributes.attr_value` обычно определяется со строковым типом данных, чтобы вместить все возможные атрибуты одного столбца. Таким образом, он не отклоняет недействительные данные.

EAV/anti/insert-eav.sql

```
INSERT INTO IssueAttributes (issue_id, attr_name, attr_value)
VALUES (1234, 'date_reported', 'banana'); -- Не ошибка!
```

Некоторые разработчики пытаются расширить архитектуру EAV, добавляя отдельный столбец `attr_value` для каждого типа данных SQL и оставляя зна-

чения NULL в неиспользуемых столбцах. Это позволяет использовать типы данных, но запросы становятся еще более сложными:

EAV/anti/data-types.sql

```
EAV/anti/data-types.sql
SELECT issue_id, COALESCE(attr_value_date, attr_value_datetime,
  attr_value_integer, attr_value_numeric, attr_value_float,
  attr_value_string, attr_value_text) AS "date_reported"
FROM IssueAttributes
WHERE attr_name = 'date_reported';
```

Для поддержки предметных областей или типов данных, определяемых пользователем, придется добавить еще больше столбцов.

Невозможно обеспечить ссылочную целостность

В традиционной базе данных можно ограничить диапазон некоторых атрибутов, определяя внешний ключ к таблице сопоставления. Например, атрибут status ошибки или проблемы должен быть одним значением из короткого списка, хранящегося в таблице BugStatus.

EAV/anti/foreign-key-plain.sql

```
CREATE TABLE Issues (
  issue_id SERIAL PRIMARY KEY,
  -- другие столбцы
  status VARCHAR(20) NOT NULL DEFAULT 'NEW',
  FOREIGN KEY (status) REFERENCES BugStatus(status)
);
```

В архитектуре EAV подобные ограничения могут применяться к столбцу attr_value. Ограничение ссылочной целостности применяется к каждой строке таблицы.

EAV/anti/foreign-key-eav.sql

```
CREATE TABLE IssueAttributes (
  issue_id BIGINT UNSIGNED NOT NULL,
  attr_name VARCHAR(100) NOT NULL,
  attr_value VARCHAR(100),
  FOREIGN KEY (attr_value) REFERENCES BugStatus(status)
);
```

Если определить ограничение, *каждый* атрибут должен будет соответствовать значению из BugStatus, а не только атрибуту status.

Невозможно определить произвольные имена атрибутов

Отчеты для руководителя все еще неточные. Вы обнаруживаете, что имена атрибутов непоследовательны. Одна ошибка использует атрибут, имя которого

задается строкой `date_reported`, а в другой ошибке имя атрибута задается строкой `report_date`. Очевидно, обе строки должны содержать одинаковую информацию.

Подсчет ошибок по датам может выглядеть так:

EAV/anti/count.sql

```
SELECT date_reported, COUNT(*) AS bugs_per_date
FROM (SELECT DISTINCT issue_id, attr_value AS date_reported
      FROM IssueAttributes
      WHERE attr_name IN ('date_reported', 'report_date'))
GROUP BY date_reported;
```

Существует риск того, что для заданной ошибки атрибут называется как-нибудь иначе. А может оказаться, что атрибут был сохранен дважды под двумя разными именами. Придется писать новый код для проверки этих случаев.

Как вариант, можно объявить внешний ключ для столбца `attr_name` к таблице сопоставления, которая содержит утвержденные имена атрибутов. С другой стороны, при этом не будет поддерживаться возможность определения атрибутов на ходу для каждой сущности. Это довольно частое применение архитектуры EAV.

Реконструирование записей

При работе с таблицей `Issues` естественно извлекать записи с атрибутами, находящимися в столбцах. Описание проблемы должно содержаться в одной строке, как в традиционной таблице.

issue_id	date_reported	status	priority	description
1234	2009-06-01	NEW	HIGH	Сохранение не работает

Так как каждый атрибут хранится в отдельной записи таблицы `IssueAttributes`, чтобы извлечь их как часть одной записи, требуется соединить все атрибуты. Все атрибуты должны быть известны в момент написания запроса. Следующий запрос реконструирует запись, приведенную выше, с использованием соединений.

EAV/anti/reconstruct.sql

```
SELECT i.issue_id,
  i1.attr_value AS "date_reported",
  i2.attr_value AS "status",
  i3.attr_value AS "priority",
  i4.attr_value AS "description"
FROM Issues AS i
  LEFT OUTER JOIN IssueAttributes AS i1
```

```
    ON i.issue_id = i1.issue_id AND i1.attr_name = 'date_reported'
  LEFT OUTER JOIN IssueAttributes AS i2
    ON i.issue_id = i2.issue_id AND i2.attr_name = 'status'
  LEFT OUTER JOIN IssueAttributes AS i3
    ON i.issue_id = i3.issue_id AND i3.attr_name = 'priority';
  LEFT OUTER JOIN IssueAttributes AS i4
    ON i.issue_id = i4.issue_id AND i4.attr_name = 'description';
WHERE i.issue_id = 1234;
```

С увеличением количества атрибутов возрастает количество соединений, а затраты на такие запросы начинают возрастать по экспоненте.

Альтернативное решение использует агрегирование, группировку записей с одинаковыми `issue_id` и выбор каждого соответствующего атрибута из группы с использованием условного выражения `CASE`.

EAV/anti/reconstruct-groupby.sql

```
SELECT issue_id,
  MAX(CASE attr_name WHEN 'date_reported'
      THEN attr_value END) AS "date_reported",
  MAX(CASE attr_name WHEN 'status'
      THEN attr_value END) AS "status",
  MAX(CASE attr_name WHEN 'priority'
      THEN attr_value END) AS "priority",
  MAX(CASE attr_name WHEN 'description'
      THEN attr_value END) AS "description"
FROM Issues
WHERE issue_id = 1234
GROUP BY issue_id;
```

Эффект внутренней платформы

EAV — классический пример более общего антипаттерна проектирования программного продукта с такими широкими возможностями настройки, что он начинает имитировать платформу, которая использовалась для его создания.

Полное дублирование платформы потребует слишком много усилий. Зрелая и надежная платформа — результат многолетней работы. Естественно, в рамках обычного проекта нет такого ресурса времени.

Из-за этого коду «внутренней платформы» приходится искать легкие пути, реализуя ограниченные версии функциональности платформы. Несмотря на значительные трудозатраты, результат будет не лучше платформы, на которой все начиналось. Наоборот, он будет хуже, сложнее и в нем будет больше ошибок.

Как распознать антипаттерн

Следующие фразы от коллег по команде могут указывать на применение антипаттерна «Сущность — атрибут — значение» (EAV):

- «Эта база данных отлично расширяется без изменения метаданных. Новые атрибуты можно определять прямо во время выполнения».

  Реляционные БД не поддерживают такую гибкость. Когда кто-то заявляет, что он создал произвольно расширяемую базу данных, скорее всего, он использует архитектуру EAV.

- «Сколько всего соединений можно выполнить в запросе?»

  Если вам нужен запрос с таким большим количеством соединений, что вы беспокоитесь о превышении лимитов БД, возможно, проблема кроется в ее архитектуре. Архитектура EAV часто приводит к подобным проблемам.

- «Не понимаю, как составить отчет для нашей платформы электронной коммерции. Нужно нанять консультанта, который сделает это».

  Похоже, многие готовые программные пакеты на основе баз данных и с широкими возможностями настройки используют архитектуру EAV. В результате самые распространенные запросы отчетов становятся очень сложными и даже непрактичными.

Допустимые применения антипаттерна

Трудно оправдать применение антипаттерна EAV в реляционной базе данных. Приходится отказываться от слишком многих преимуществ реляционной парадигмы. Но это не снимает вполне законной потребности в поддержке динамических атрибутов в некоторых приложениях.

В большинстве приложений, требующих бессхемных данных, в действительности они нужны только для нескольких таблиц или даже всего для одной таблицы. Остальные требования к данным соответствуют стандартной архитектуре таблиц. Если вы готовы к лишней работе и риску EAV в своем проекте, возможно, умеренное применение этой архитектуры станет меньшим злом. Опытные консультанты в области баз данных говорят, что системы на базе EAV выходят из-под контроля в течение года.

Если у вас возникает необходимость в нереляционном управлении данными, лучше всего использовать *нереляционную* технологию. Эта книга посвящена SQL, а не альтернативам SQL, так что приведем лишь краткий перечень таких технологий:

- *Berkeley DB*[1] — популярное хранилище пар «ключ — значение», которое легко встраивается в разнообразные приложения.
- *DynamoDB*[2] — бессерверная база данных пар «ключ — значение», предоставляемая в виде облачного сервиса на Amazon.com.
- *Elasticsearch*[3] — распределенное поисковое и аналитическое ядро.
- *Hadoop*[4] и *HBase* составляют СУБД с открытым кодом, вдохновленную алгоритмом MapReduce компании Google для распределения запросов в очень больших полуструктурированных хранилищах данных.
- *MongoDB*[5] — документоориентированная база данных.
- *Redis*[6] — сервер структур данных, по умолчанию хранящий данные в памяти.

Хотя популярность этих и других нереляционных проектов растет, слабые стороны EAV в отношении реляционных баз данных сохраняются и в них. С нежесткими метаданными труднее формулировать простые запросы. Приложениям приходится тратить значительные усилия на определение структуры данных и адаптацию к ней.

Решение: моделирование подтипов

Если архитектура EAV кажется вам подходящим решением, еще раз подумайте, прежде чем реализовывать ее. Скорее всего, после небольшого традиционного анализа вы обнаружите, что проще моделировать данные вашего проекта в структуре традиционных таблиц и с большими гарантиями целостности.

Существует несколько способов хранения таких данных без применения EAV. Многие решения лучше работают при конечном количестве подтипов, когда вы знаете атрибуты каждого подтипа. Выбор зависит от того, как вы собираетесь формулировать запросы, и архитектуру необходимо выбирать для каждого конкретного случая.

Приведенные ниже примеры позаимствованы из книги Мартина Фаулера (Martin Fowler) «Patterns of Enterprise Application Architecture»[7] [Fow03].

---

[1] https://www.oracle.com/database/technologies/related/berkeleydb.html

[2] https://aws.amazon.com/dynamodb/

[3] https://www.elastic.co/elasticsearch/

[4] https://hadoop.apache.org/

[5] https://www.mongodb.org/

[6] https://redis.io/

[7] Фаулер М. Шаблоны корпоративных приложений.

Наследование от общей таблицы

В простейшем решении все взаимосвязанные типы хранятся в одной таблице, с отдельными столбцами для каждого атрибута, существующего в типе. Один атрибут используется для определения подтипа заданной строки. В следующем примере этот атрибут называется `issue_type`. Некоторые атрибуты являются общими для всех подтипов. Многие атрибуты относятся к конкретному подтипу, и эти столбцы должны содержать значение `NULL` для любой строки с объектом, к которому атрибут не относится; столбцы со значениями, отличными от `NULL`, становятся разреженными.

EAV/soln/create-sti-table.sql

```
CREATE TABLE Issues (
  issue_id SERIAL PRIMARY KEY,
  reported_by BIGINT UNSIGNED NOT NULL,
  product_id BIGINT UNSIGNED,
  priority VARCHAR(20),
  version_resolved VARCHAR(20),
  status VARCHAR(20),
  issue_type VARCHAR(10), -- BUG или FEATURE
  severity VARCHAR(20), -- только для ошибок
  version_affected VARCHAR(20), -- только для ошибок
  sponsor VARCHAR(50), -- только для запросов на добавление функций
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

С появлением новых типов объектов база данных должна вмещать атрибуты, их описывающие. При добавлении атрибутов для новых типов объектов придется изменять таблицу и добавлять в нее новые столбцы. При этом можно столкнуться с практическими ограничениями на количество столбцов в таблице.

У наследования от общей таблицы есть и другое ограничение: не существует метаданных, определяющих, к какому подтипу относится тот или иной атрибут. В своем приложении вы можете игнорировать некоторые атрибуты, если знаете, что они не относятся к подтипу объекта заданной строки. При этом необходима осторожность, потому что придется вручную отслеживать, какие атрибуты относятся к каждому подтипу. Оптимально использовать метаданные для определения этой информации в базе данных.

Наследование от общей таблицы лучше всего работает при небольшом количестве подтипов и атрибутов подтипов, а также с использованием паттернов работы с однотабличной базой данных (например, Active Record).

Наследование с таблицами конечных классов

В другом решении для каждого подтипа создается отдельная таблица. Каждая таблица содержит атрибуты, общие для базового типа, а также специфические атрибуты соответствующего подтипа.

EAV/soln/create-concrete-tables.sql

```
CREATE TABLE Bugs (
  issue_id SERIAL PRIMARY KEY,
  reported_by BIGINT UNSIGNED NOT NULL,
  product_id BIGINT UNSIGNED,
  priority VARCHAR(20),
  version_resolved VARCHAR(20),
  status VARCHAR(20),
  severity VARCHAR(20), -- только для ошибок
  version_affected VARCHAR(20), -- только для ошибок
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

CREATE TABLE FeatureRequests (
  issue_id SERIAL PRIMARY KEY,
  reported_by BIGINT UNSIGNED NOT NULL,
  product_id BIGINT UNSIGNED,
  priority VARCHAR(20),
  version_resolved VARCHAR(20),
  status VARCHAR(20),
  sponsor VARCHAR(50), -- только для запросов на добавление функций
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

Преимущество наследования с таблицами конечных классов перед наследованием от общей таблицы заключается в том, что в таблице становится невозможно сохранить строку со значениями атрибутов, не относящихся к подтипу этой строки. При попытке сослаться на столбец атрибута, не существующий в этой таблице, база данных автоматически сообщит об ошибке.

Например, столбец `severity` отсутствует в таблице `FeatureRequests`:

EAV/soln/insert-concrete.sql

```
INSERT INTO FeatureRequests (issue_id, severity) VALUES ( ... ); -- ОШИБКА!
```

У наследования с таблицами конечных классов есть еще одно преимущество: в этом случае не требуется дополнительный атрибут для определения подтипа в каждой строке, в отличие от решения с наследованием от общей таблицы.

Впрочем, у него есть и недостатки. Общие атрибуты трудно отличить от атрибутов подтипов. Кроме того, при добавлении нового атрибута в набор общих атрибутов придется изменять таблицу каждого подтипа.

Из метаданных невозможно понять, что данные, хранящиеся в таблицах подтипов, принадлежат взаимосвязанным объектам. Таким образом, если разработчик, недавно присоединившийся к проекту, просмотрит определения таблиц, он увидит, что некоторые столбцы являются общими для всех таблиц подтипов, но не сможет понять из метаданных, существует ли между таблицами логическая связь или же это сходство — всего лишь совпадение.

Если вам требуется провести поиск по всем объектам независимо от их подтипов, хранение каждого подтипа в отдельной таблице усложнит задачу. Чтобы упростить запрос, задайте представление соединения таблиц с выбором только общих атрибутов.

EAV/soln/view-concrete.sql

```
CREATE VIEW Issues AS
  SELECT b.issue_id, b.reported_by, ... 'bug' AS issue_type
  FROM Bugs AS b
    UNION ALL
  SELECT f.issue_id, f.reported_by, ... 'feature' AS issue_type
  FROM FeatureRequests AS f;
```

Наследование с таблицами конечных классов лучше всего использовать в случаях, когда запрашивать сразу все подтипы требуется редко.

Наследование с таблицами классов

Третье решение моделирует механизм наследования, как будто таблицы являются объектно-ориентированными классами. Создайте одну таблицу для базового типа, содержащую атрибуты, общие для всех подтипов. Затем для каждого подтипа создается другая таблица с первичным ключом, который также служит внешним ключом для базовой таблицы.

EAV/soln/create-class-tables.sql

```
CREATE TABLE Issues (
  issue_id SERIAL PRIMARY KEY,
  reported_by BIGINT UNSIGNED NOT NULL,
  product_id BIGINT UNSIGNED,
  priority VARCHAR(20),
  version_resolved VARCHAR(20),
  status VARCHAR(20),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

CREATE TABLE Bugs (
```

```
  issue_id BIGINT UNSIGNED PRIMARY KEY,
  severity VARCHAR(20),
  version_affected VARCHAR(20),
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id)
);

CREATE TABLE FeatureRequests (
  issue_id BIGINT UNSIGNED PRIMARY KEY,
  sponsor VARCHAR(50),
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id)
);
```

Отношение «один к одному» обеспечивается метаданными, так как внешний ключ зависимой таблицы также является первичным ключом и, следовательно, должен быть уникальным. Такое решение обеспечивает эффективный поиск по всем подтипам при условии, что он касается только атрибутов базового типа. После нахождения элементов, соответствующих критерию поиска, можно получить атрибуты подтипа, формируя запрос к таблицам соответствующих подтипов.

При относительно небольшом количестве подтипов можно написать запрос по всем подтипам сразу; его результатом будет разреженный итоговый набор — такой же, как в случае наследования от общей таблицы. Атрибуты, не применимые к подтипу заданной строки, содержат `NULL`.

EAV/soln/select-class.sql

```
SELECT i.*, b.*, f.*
FROM Issues AS i
  LEFT OUTER JOIN Bugs AS b USING (issue_id)
  LEFT OUTER JOIN FeatureRequests AS f USING (issue_id);
```

Результат также подходит для определения `VIEW`.

Такое решение эффективнее всего, если требуется часто писать запросы по всем подтипам с указанием их общих столбцов.

К недостаткам этого решения (а также архитектуры наследования с таблицами конечных классов) следует отнести невозможность создания ограничений уникальности между таблицами. Также трудно проследить за тем, что каждой сущности назначается только один подтип; она может встречаться сразу в нескольких таблицах подтипов.

Полуструктурированные данные

Если вы работаете с множеством подтипов или вам часто приходится вводить новые атрибуты, можно добавить столбец типа `BLOB` или связанного типа для хранения данных в таком формате, как XML или JSON; в этих форматах коди-

руются как имена атрибутов, так и их значения. Этот паттерн также называется *сериализованным LOB*[1]. Во многих базах данных SQL сейчас для этой цели поддерживается специализированный тип JSON.

EAV/soln/create-blob-tables.sql

```
CREATE TABLE Issues (
  issue_id SERIAL PRIMARY KEY,
  reported_by BIGINT UNSIGNED NOT NULL,
  product_id BIGINT UNSIGNED,
  priority VARCHAR(20),
  version_resolved VARCHAR(20),
  status VARCHAR(20),
  issue_type VARCHAR(10), -- BUG или FEATURE
  attributes JSON NOT NULL, -- все динамические атрибуты строки
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

Преимуществом этого решения является его полная расширяемость, хотя формат более или менее стандартизирован и существуют инструменты для получения элементов внутренних данных. Из-за этого данные и называются полуструктурированными. Новые атрибуты можно сохранить в полуструктурированном столбце в любой момент. В каждой строке теоретически может храниться собственный набор атрибутов, что позволяет создать столько подтипов, сколько в таблице строк.

К недостаткам можно отнести неудобство использования SQL для обращения к конкретным атрибутам такой структуры. Многие реализации SQL содержат встроенные функции для поиска или преобразования XML или JSON, но они не так элегантны или эффективны, как работа с более традиционными типами данных. Выражения SQL и операторы проектировались с расчетом на то, что каждый столбец представляет собой дискретное, скалярное значение данных, а не составной полуструктурированный документ.

Полуструктурированные данные лучше всего использовать, когда конечного множества подтипов недостаточно и необходима абсолютная гибкость определения новых атрибутов в любой момент.

Постобработка

К сожалению, иногда без EAV не обойтись: например, если вы унаследовали проект и не можете изменить его или если ваша компания приобрела стороннюю программную платформу, использующую EAV. В таком случае ознакомьтесь

[1] LOB (от англ. Large OBject) — тип данных, используемый для хранения больших объектов. — *Примеч. ред.*

с возможными проблемами в разделе «Антипаттерн» данной главы, чтобы подготовиться и спланировать работу с этой архитектурой.

Также не пытайтесь писать запросы на извлечение сущности в виде одной записи, как в традиционной таблице. Вместо этого запрашивайте атрибуты, связанные с сущностью, в виде множества строк — так, как они хранятся в базе.

EAV/soln/post-process.sql

```
SELECT issue_id, attr_name, attr_value
FROM IssueAttributes
WHERE issue_id = 1234;
```

Результат запроса может выглядеть так:

issue_id	attr_name	attr_value
1234	date_reported	2009-06-01
1234	description	Сохранение не работает
1234	priority	ВЫСОКИЙ
1234	product	Open RoundFile
1234	reported_by	Билл
1234	severity	потеря функциональности
1234	status	НОВЫЙ

Такой запрос проще написать и проще обработать в базе данных. Запрос возвращает все атрибуты, связанные с ошибкой, даже если их количество неизвестно при написании запроса.

Чтобы использовать результат в этом формате, необходимо написать код приложения, который перебирает строки результирующего набора и задает свойства объекта. Для примера возьмем следующий код Python:

EAV/soln/post-process.py

```
import mysql.connector

cnx = mysql.connector.connect(user='scott', database='test')
cursor = cnx.cursor()

issue_id = 1234
query = """
    SELECT attr_name, attr_value
    FROM IssueAttributes
    WHERE issue_id = %s"""
cursor.execute(query, (issue_id,))
```

```
issue = {}
for (row) in cursor:
    (field, value) = row
    issue[field] = value

cnx.commit()
```

На первый взгляд кажется, что приходится выполнять слишком много лишней работы, но это следствие структур «система внутри системы», таких как EAV. SQL уже предоставляет средства идентификации отдельных атрибутов (помещая их в отдельные столбцы). EAV не позволяет использовать традиционный способ управления метаданными в SQL, поэтому не стоит удивляться, что вам приходится писать больше кода и выполнять работу, которую SQL обычно делает за вас.

Используйте метаданные как метаданные.

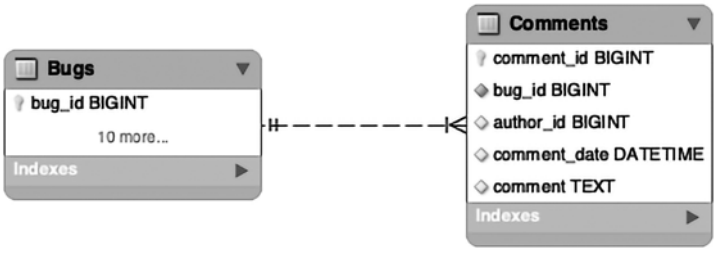
Подъехав к развилке дороги, следуйте по ней.

➤ *Йоги Берра (Yogi Berra)*

ГЛАВА 7

Полиморфная связь

Пользователи должны иметь возможность *оставлять комментарии* к ошибкам. С одной ошибкой может быть связано много комментариев, но любой комментарий должен относиться только к одной ошибке. Таким образом, между `Bugs` и `Comments` существует отношение «один ко многим». Диаграмма «объект — отношение» для подобных простых связей изображена ниже.



Следующий фрагмент SQL показывает, как создается такая таблица:

Polymorphic/intro/comments.sql

```
CREATE TABLE Comments (
  comment_id SERIAL PRIMARY KEY,
  bug_id BIGINT UNSIGNED NOT NULL,
  author_id BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment TEXT NOT NULL,
  FOREIGN KEY (author_id) REFERENCES Accounts(account_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

Однако комментарии могут относиться к двум таблицам. `Bugs` и `FeatureRequests` — схожие сущности, хотя и могут храниться в отдельных таблицах (см. раздел «Наследование с таблицами конечных классов» в главе 6). Комментарии желательно хранить в одной таблице независимо от того, к какому типу проблем они относятся — ошибкам или функциям, но вы не сможете объявить внешний ключ,

ссылающийся на несколько родительских таблиц. Следующее объявление не имеет смысла:

Polymorphic/intro/nonsense.sql

```
  ...
  FOREIGN KEY (issue_id)
      REFERENCES Bugs(issue_id) OR FeatureRequests(issue_id)
);
```

Разработчики также грешат недопустимыми запросами SQL к нескольким таблицам, как в следующем примере:

Polymorphic/intro/nonsense.sql

```
Polymorphic/intro/nonsense.sql
SELECT c.*, i.summary, i.status
FROM Comments AS c
JOIN c.issue_type AS i USING (issue_id);
```

В SQL невозможно выполнять соединения с разными таблицами на уровне записи. Синтаксис SQL требует, чтобы имена всех таблиц были заданы на момент отправки запроса. Таблицы не могут изменяться во время запроса. Что-то тут не сходится.

Цель: ссылки на несколько родительских таблиц

На первый взгляд возможность ссылаться из дочерней таблицы `Comments` на любую из родительских таблиц кажется естественной. Внешний ключ заданной строки данных содержит либо ссылку на `Bugs`, либо ссылку на `FeatureRequests`. Иногда столбец должен содержать информацию о таблице, на которую он ссылается, на уровне отдельных строк данных.

Антипаттерн: использование внешнего ключа двойного назначения

Решение, используемое в подобных случаях, стало таким популярным, что ему было присвоено отдельное имя: *полиморфная связь*. Также иногда используется термин «*беспорядочные связи*», так как это решение предусматривает ссылки на разные таблицы.

Определение полиморфной связи

Чтобы механизм полиморфных связей работал, необходимо добавить дополнительный строковый столбец наряду с внешним ключом для `issue_id`, содержащий имя родительской таблицы, на которую ссылается текущая строка.

В этом примере новый столбец называется `issue_type` и содержит строковое значение, соответствующее именам двух возможных родительских таблиц (либо `Bugs`, либо `FeatureRequests`).

Polymorphic/anti/comments.sql

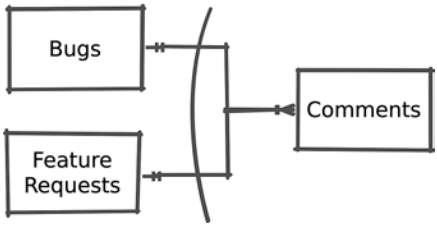
```
CREATE TABLE Comments (
  comment_id SERIAL PRIMARY KEY,
  issue_type VARCHAR(20), -- Bugs или FeatureRequests
  issue_id BIGINT UNSIGNED NOT NULL,
  author BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME,
  comment TEXT,
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

Сразу бросается в глаза отличие: объявление внешнего ключа для `issue_id` отсутствует. Собственно, так как в ограничении внешнего ключа может быть указана только одна таблица, полиморфная связь означает, что для его объявления не может использоваться синтаксис ограничений SQL. В результате пропадает контроль целостности данных, который бы гарантировал, что значение в `Comments.issue_id` совпадает со значением в родительской таблице.

Точно так же никакие метаданные не гарантируют, что строка в `Comments.issue_type` соответствует таблице, существующей в базе данных.

Представление полиморфных связей на диаграмме

Не существует официально принятого способа представления полиморфных отношений на диаграммах «объект — отношение». Разработчику, который хочет использовать эту архитектуру, приходится изобретать собственный стиль. Некоторые диаграммы выглядят так:



На диаграмме показан внешний ключ, который «разветвляется», так что каждая строка таблицы `Comments` связана либо с одной, либо с другой родительской таблицей. Дуга, которая проходит развилку, — условное обозначение, которое иногда используется на таких диаграммах, но так как эти диаграммы официально не утверждены, не стоит ожидать, что это обозначение будет встречаться на них повсеместно.

Запросы с полиморфными связями

Значение `issue_id` в таблице `Comments` может встречаться в столбце первичного ключа обеих родительских таблиц, `Bugs` и `FeatureRequests`. Или значение может встречаться в одной родительской таблице, но отсутствовать в другой. Следовательно, при соединении дочерней таблицы с родительской очень важно правильно использовать `issue_type`. Значение `issue_id` не должно связываться с таблицей `FeatureRequests`, если оно предназначено для связывания с таблицей `Bugs`.

Например, следующая инструкция получает комментарии для заданной ошибки по значению ее первичного ключа `1234`:

Polymorphic/anti/select.sql

```
SELECT *
FROM Bugs AS b JOIN Comments AS c
  ON (b.issue_id = c.issue_id AND c.issue_type = 'Bugs')
WHERE b.issue_id = 1234;
```

Хотя приведенный запрос будет работать, если ошибки хранятся в одной таблице `Bugs`, проблемы возникнут, если таблица `Comments` связана с обеими таблицами, `Bugs` и `FeatureRequests`. В SQL при соединении должны быть указаны все таблицы; невозможно связать `Comments` с двумя разными таблицами и переключаться между ними на уровне строк данных в зависимости от значения столбца `Comments.issue_type`.

Чтобы извлечь ошибку или функциональность для заданного комментария, необходимо выполнить запрос с внешним соединением с обеими родительскими таблицами. Только одна из родительских таблиц будет соответствовать соединению, так как часть условия соединения зависит от значения в столбце `Comment.issue_type`. Использование внешнего соединения означает, что поля из таблицы, для которых нет совпадения, будут содержать `NULL` в итоговом наборе.

Polymorphic/anti/select.sql

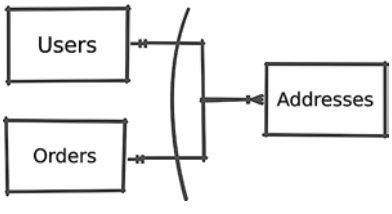
```
SELECT *
FROM Comments AS c
  LEFT OUTER JOIN Bugs AS b
    ON (b.issue_id = c.issue_id AND c.issue_type = 'Bugs')
  LEFT OUTER JOIN FeatureRequests AS f
    ON (f.issue_id = c.issue_id AND c.issue_type = 'FeatureRequests');
```

Результат может выглядеть так:

f.issue_id	b.issue_id	c.comment	c.issue_id	c.issue_type	c.comment_id
6789	Bugs	1234	It crashes!	1234	NULL
9876	Feature...	2345	Great idea!	NULL	2345

Не объектно-ориентированный пример

В примере с `Bugs` и `FeatureRequests` две родительские таблицы должны моделировать взаимосвязанные подтипы. Полиморфные связи также могут использоваться в ситуациях, когда родительские таблицы вообще не связаны друг с другом. На следующей диаграмме представлены таблицы, которые могут использоваться в базе данных интернет-магазина. Две таблицы — `Users` и `Orders` — могут быть связаны с `Addresses`. Как и прежде, формат диаграммы «объект — отношение» неофициальный.



Polymorphic/anti/addresses.sql

```
CREATE TABLE Addresses (
  address_id SERIAL PRIMARY KEY,
  parent VARCHAR(20), -- "Users" or "Orders"
  parent_id BIGINT UNSIGNED NOT NULL,
  address TEXT
);
```

В этом примере таблица `Addresses` содержит полиморфный столбец, в котором указывается `Users` или `Orders` в качестве родительской таблицы для заданного адреса. Учтите, что нужно выбрать либо одну, либо другую таблицу. Заданный адрес не может быть связан и с пользователем, и с заказом, даже если пользователь разместил заказ для отправки покупок на свой адрес.

Кроме того, если у пользователя адрес доставки не совпадает с адресом выставления счета, необходимо отразить эти отличия в таблице `Addresses`; точно так же и остальные родительские таблицы должны помнить об особом случае использования таблицы `Address`. Эти «особые случаи» плодятся как сорняки.

Polymorphic/anti/addresses.sql

```
CREATE TABLE Addresses (
  address_id SERIAL PRIMARY KEY,
  parent VARCHAR(20), -- "Users" или "Orders"
  parent_id BIGINT UNSIGNED NOT NULL,
  users_usage VARCHAR(20), -- "billing" или "shipping"
  orders_usage VARCHAR(20), -- "billing" или "shipping"
  address TEXT
);
```

Как распознать антипаттерн

Если ваши коллеги по команде произносят следующие фразы, это может указывать на применение антипаттерна «Полиморфная связь»:

- «В этой схеме можно связать тег (или другой атрибут) с любым другим ресурсом в базе данных».

  Как и в случае с EAV, следует с осторожностью относиться к любым заявлениям о неограниченной гибкости, потому что, скорее всего, такое решение нарушает какие-нибудь правила.
- «В нашей архитектуре базы данных нельзя объявлять внешние ключи».

  Еще один тревожный признак. Внешние ключи — фундаментальная особенность реляционных баз данных, а у решения, которое неспособно корректно работать со ссылочной целостностью, очень много проблем.
- «Для чего нужен столбец `entity_type`? А, он сообщает, на что указывает тот, другой столбец».

  Любой внешний ключ должен ссылаться на одну таблицу во всех строках данных.

Фреймворк Ruby on Rails поддерживает полиморфные связи, объявляя классы Active Record с атрибутом `:polymorphic`.[1] Например, связывание `Comments` с `Bugs` и `FeatureRequests` может выполняться следующим образом:

Polymorphic/recog/commentable.rb

```
class Comment < ActiveRecord::Base
  belongs_to :commentable, :polymorphic => true
end

class Bug < ActiveRecord::Base
  has_many :comments, :as => :commentable
end

class FeatureRequest < ActiveRecord::Base
  has_many :comments, :as => :commentable
end
```

Фреймворк Hibernate для Java поддерживает отношения наследования между сущностями, используя разнообразные объявления схем[2].

---

[1] https://guides.rubyonrails.org/association_basics.html#polymorphic-associations

[2] https://docs.jboss.org/hibernate/orm/current/userguide/html_single/Hibernate_User_Guide.html#entity-inheritance

Комбинирование данных с метаданными

Возможно, вы заметили некоторое сходство между антипаттерном «Полиморфная связь» и антипаттерном «Сущность — атрибут — значение», описанным в предыдущей главе. В обоих случаях имя объекта метаданных хранится в виде строкового значения. В EAV имя столбца атрибута хранится в виде строки в столбце `attr_name`. В «Полиморфной связи» имена родительских таблиц хранятся в столбце `issue_type`. Иногда такой подход называется *комбинированием данных с метаданными*. Эта же концепция встречается в другой форме в главе 8 «Многостолбцовые атрибуты».

Допустимые применения антипаттерна

Старайтесь избегать антипаттерна «Полиморфная связь» — используйте ограничения (например, внешние ключи) для обеспечения ссылочной целостности. Антипаттерн «Полиморфная связь» слишком сильно зависит от кода приложения (а не от метаданных).

Возможно, вы обнаружите, что этого антипаттерна не избежать при использовании таких объектно-реляционных фреймворков, как Hibernate. Такой фреймворк может снизить риски, создаваемые полиморфной связью, за счет инкапсуляции логики приложения для обеспечения ссылочной целостности.

Выбрав проверенный и надежный фреймворк, можно быть отчасти уверенным в том, что его проектировщики написали код реализации связей без ошибок. Но реализуя полиморфную связь с нуля без помощи фреймворка, по сути, вы заново изобретаете велосипед.

Решение: упрощение отношений

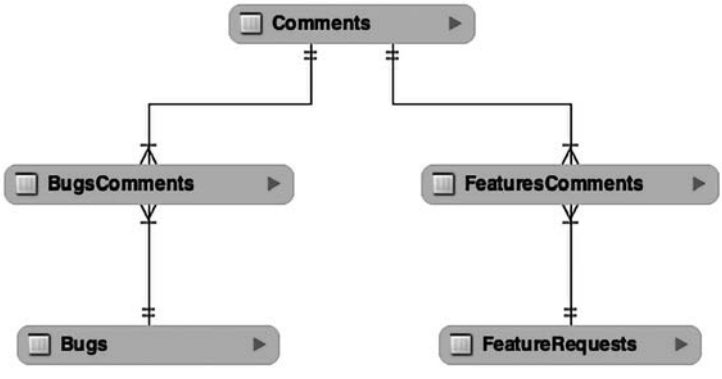
Лучше перепроектируйте базу данных, чтобы избежать недостатков полиморфной связи и при этом поддерживать нужную модель данных. В следующих разделах описывается несколько решений, которые реализуют отношения между данными, но более эффективно используют метаданные для обеспечения целостности.

Обратные ссылки

Одно из решений для этого антипаттерна упрощается, если задуматься над природой проблемы: *полиморфная связь имеет обратное направление*.

Создание таблиц пересечений

Внешний ключ в дочерней таблице `Comments` не может ссылаться на разные родительские таблицы, поэтому вместо этого используются разные внешние ключи, ссылающиеся на таблицу `Comments`. Создайте отдельную таблицу пересечений для каждой родительской таблицы и включите в каждую таблицу пересечений внешний ключ для `Comments`, а также внешний ключ для соответствующей родительской таблицы. Структура изображена на следующей диаграмме.



Polymorphic/soln/reverse-reference.sql

```
CREATE TABLE BugsComments (
  issue_id BIGINT UNSIGNED NOT NULL,
  comment_id BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (issue_id, comment_id),
  FOREIGN KEY (issue_id) REFERENCES Bugs(issue_id),
  FOREIGN KEY (comment_id) REFERENCES Comments(comment_id)
);
CREATE TABLE FeaturesComments (
  issue_id BIGINT UNSIGNED NOT NULL,
  comment_id BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (issue_id, comment_id),
  FOREIGN KEY (issue_id) REFERENCES FeatureRequests(issue_id),
  FOREIGN KEY (comment_id) REFERENCES Comments(comment_id)
);
```

Это решение снимает необходимость в столбце `Comments.issue_type`. Теперь метаданные обеспечивают целостность данных, а приложение перестает зависеть от правильности кода управления связями.

Стоп-сигнал

У этого решения есть потенциальный недостаток: оно разрешает связи, которые вы, возможно, разрешать не захотите. Таблицы пересечений обычно моделируют связи «многие ко многим», поэтому отдельный комментарий может быть

связан с несколькими ошибками или несколькими запросами на добавление функций. Однако вы, скорее всего, хотите, чтобы каждый комментарий относился только к одной ошибке или запросу. Это правило можно реализовать (по крайней мере частично) объявлением ограничения `UNIQUE` для столбца `comment_id` каждой таблицы пересечений.

Polymorphic/soln/reverse-unique.sql

```
CREATE TABLE BugsComments (
  issue_id BIGINT UNSIGNED NOT NULL,
  comment_id BIGINT UNSIGNED NOT NULL,
  UNIQUE KEY (comment_id),
  PRIMARY KEY (issue_id, comment_id),
  FOREIGN KEY (issue_id) REFERENCES Bugs(issue_id),
  FOREIGN KEY (comment_id) REFERENCES Comments(comment_id)
);
```

Тем самым гарантируется, что в таблице пересечений любой комментарий будет упоминаться только один раз, что естественно предотвращает его связывание с несколькими ошибками или запросами функций. Однако метаданные не препятствуют появлению ссылки на комментарий в обеих таблицах пересечений, в результате чего комментарий будет связан как с ошибкой, так и с запросом функции. Скорее всего, это не то, что вам нужно, но избежать этого можно путем написания соответствующего кода приложения.

Поиск в обе стороны

Чтобы получить комментарии для конкретной ошибки или функции, просто используйте таблицу пересечений:

Polymorphic/soln/reverse-join.sql

```
SELECT *
FROM BugsComments AS b
  JOIN Comments AS c USING (comment_id)
WHERE b.issue_id = 1234;
```

Чтобы запросить ошибку или функцию для заданного экземпляра комментария, выполните внешнее соединение с обеими таблицами пересечений. Необходимо указать имена всех возможных родительских таблиц, но это не сложнее запросов, которые вам приходится использовать в антипаттерне «Полиморфная связь». Кроме того, при использовании таблиц пересечений можно рассчитывать на ссылочную целостность, в отличие от ситуаций с полиморфной связью.

Polymorphic/soln/reverse-join.sql

```
SELECT *
FROM Comments AS c
  LEFT OUTER JOIN (
```

```
    BugsComments JOIN Bugs AS b USING (issue_id)
  ) USING (comment_id)
  LEFT OUTER JOIN (
    FeaturesComments JOIN FeatureRequests AS f USING (issue_id)
  ) USING (comment_id)
WHERE c.comment_id = 9876;
```

Слияние

Иногда нужно представить результат запроса к нескольким родительским таблицам так, словно родители хранятся в одной таблице (см. раздел «Наследование от общей таблицы» в главе 6). Это можно сделать одним из двух способов.

Сначала рассмотрим запрос с использованием `UNION`:

Polymorphic/soln/reverse-union.sql

```
SELECT b.issue_id, b.description, b.reporter, b.priority, b.status,
    b.severity, b.version_affected,
    NULL AS sponsor
  FROM Comments AS c
    JOIN (BugsComments JOIN Bugs AS b USING (issue_id))
      USING (comment_id)
  WHERE c.comment_id = 9876;

UNION
  SELECT f.issue_id, f.description, f.reporter, f.priority, f.status,
    NULL AS severity, NULL AS version_affected,
    f.sponsor
  FROM Comments AS c
    JOIN (FeaturesComments JOIN FeatureRequests AS f USING (issue_id))
      USING (comment_id)
  WHERE c.comment_id = 9876;
```

Этот запрос должен гарантированно возвращать одну строку, если в приложении один комментарий связан только с одной родительской таблицей. Так как результаты запросов могут объединяться конструкцией `UNION` только в том случае, если их столбцы совпадают по количеству и типу данных, необходимо добавить `NULL`-заполнители для столбцов, уникальных для каждой родительской таблицы. Столбцы должны перечисляться в одинаковом порядке в обоих запросах, задействованных в `UNION`.

Также рассмотрите следующий запрос, использующий функцию SQL `COALESCE()`. Функция возвращает свой первый аргумент, отличный от `NULL`. Так как в запросе используется внешнее соединение, комментарий, относящийся к функции и не имеющий совпадающих строк в `Bugs`, вернет все поля из `b.*` в виде `NULL`. Точно так же все поля `f.*` будут содержать `NULL`, если комментарий относится к ошибке, а не к функции. Поля, присутствующие только в одной из родительских таблиц, просто перечисляются в запросе; если они не актуальны для соответствующей родительской таблицы, они возвращаются в виде `NULL`.

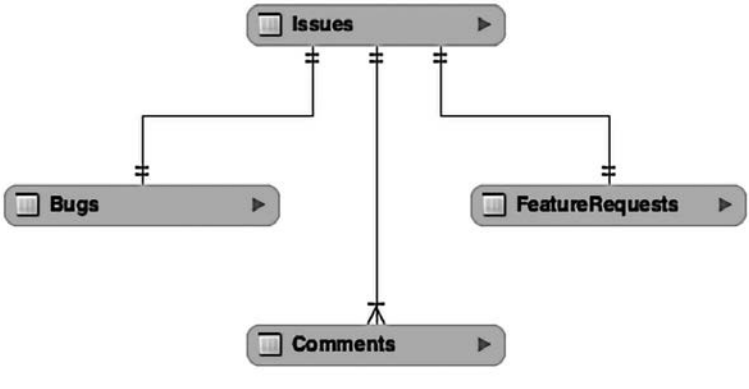
Polymorphic/soln/reverse-coalesce.sql

```
SELECT c.*,
  COALESCE(b.issue_id, f.issue_id ) AS issue_id,
  COALESCE(b.description, f.description) AS description,
  COALESCE(b.reporter, f.reporter ) AS reporter,
  COALESCE(b.priority, f.priority ) AS priority,
  COALESCE(b.status, f.status ) AS status,
  b.severity,
  b.version_affected,
  f.sponsor

FROM Comments AS c
  LEFT OUTER JOIN (BugsComments JOIN Bugs AS b USING (issue_id))
    USING (comment_id)
  LEFT OUTER JOIN (FeaturesComments JOIN FeatureRequests AS f USING (issue_id))
    USING (comment_id)
WHERE c.comment_id = 9876;
```

Оба запроса довольно сложны, поэтому они отлично подходят для выделения в сущность представления базы данных (VIEW), чтобы проще использовать их в создаваемом приложении.

Создание общей супертаблицы

В объектно-ориентированном полиморфизме к двум подтипам можно обращаться похожим образом, потому что они имеют общий супертип. В SQL антипаттерн «Полиморфная связь» не задействует критическую сущность — общий супертип. Проблема решается созданием базовой таблицы, которая расширяется всеми родительскими таблицами (см. раздел «Наследование с таблицами классов» в главе 6). Добавьте в дочернюю таблицу `Comments` внешний ключ, ссылающийся на базовую таблицу. Ниже приведены диаграмма и код возможной реализации.



Polymorphic/soln/super-table.sql

```
CREATE TABLE Issues (
  issue_id SERIAL PRIMARY KEY
);

CREATE TABLE Bugs (
  issue_id BIGINT UNSIGNED PRIMARY KEY,
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id),
  . . .
);

CREATE TABLE FeatureRequests (
  issue_id BIGINT UNSIGNED PRIMARY KEY,
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id),
  . . .
);

CREATE TABLE Comments (
  comment_id SERIAL PRIMARY KEY,
  issue_id BIGINT UNSIGNED NOT NULL,
  author BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME,
  comment TEXT,
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id),
);
```

Обратите внимание: первичные ключи `Bugs` и `FeatureRequests` также являются внешними ключами. Они ссылаются на значение суррогатного ключа, сгенерированное в таблице `Issues`, вместо того чтобы генерировать новое значение для себя.

Для заданного комментария можно получить связанную с ним ошибку или функцию, используя относительно простой запрос. Включать в этот запрос таблицу `Issues` вообще не придется, если только вы не определяете в ней столбцы атрибутов. Кроме того, поскольку значение первичного ключа таблицы `Bugs` и ее родительской таблицы `Issues` одинаковы, можно связать `Bugs` непосредственно с `Comments`. Две таблицы можно соединить даже при отсутствии ограничения внешнего ключа, связывающего их напрямую, при условии, что вы используете столбцы, представляющие сопоставимую информацию в базе данных.

Polymorphic/soln/super-join.sql

```
SELECT *
FROM Comments AS c
  LEFT OUTER JOIN Bugs AS b USING (issue_id)
  LEFT OUTER JOIN FeatureRequests AS f USING (issue_id)
WHERE c.comment_id = 9876;
```

Для заданной ошибки можно так же легко получить связанные с ней комментарии.

Polymorphic/soln/super-join.sql

```
SELECT *
FROM Bugs AS b
  JOIN Comments AS c USING (issue_id)
WHERE b.issue_id = 1234;
```

Суть в том, что при использовании такой родительской таблицы, как `Issues`, можно рассчитывать на ссылочную целостность данных в БД, обеспечиваемую внешними ключами.

В каждом табличном отношении должна быть одна дочерняя и одна родительская таблица.

Возвышенное и смешное часто столь близки, что их трудно разделить.

➤ *Томас Пейн (Thomas Paine)*

ГЛАВА 8

Многостолбцовые атрибуты

Не представляю, сколько раз мне приходилось создавать таблицы для хранения контактной информации. В таких таблицах всегда стандартный набор столбцов: имя человека, обращение, адрес и, возможно, название компании.

С телефонными номерами все сложнее. Люди используют несколько номеров, чаще всего домашний, рабочий, мобильный и факс. В таблице контактных данных эти номера легко сохранить в четырех столбцах.

Однако у пользователей в этой таблице, скорее всего, будут и дополнительные номера: телефон секретаря, второй мобильный телефон, телефон местного офиса и т. д. — или другие нестандартные категории. Можно создать дополнительные столбцы для нетипичных случаев, но такое решение выглядит неуклюже, потому что в формах ввода данных появляются редко используемые поля. Непонятно, сколько столбцов понадобится для хранения всех возможных вариантов.

Цель: хранение многозначных атрибутов

Это та же цель, что и в главе 2 «Кривая дорожка»: кажется, что атрибут принадлежит одной таблице, но у него много значений. Выше мы видели, что объединение множественных значений в строку, разделенную запятыми, усложняет проверку данных, затрудняет чтение или изменение отдельных значений, а также вычисление агрегатных выражений (например, подсчет количества значений).

Разберем этот антипаттерн на новом примере. Вам нужно, чтобы в базе данных ошибок поддерживались теги, по которым можно было бы классифицировать ошибки. Некоторые ошибки можно отнести к подсистеме программного продукта, на которую они влияют (например, *печать, отчеты, электронная почта* и т. д.). Другие можно классифицировать по природе дефекта: например, ошибки, приводящие к аварийному сбою, можно помечать тегом `crash`, ответ о низкой

скорости работы — тегом `performance`, а неудачный выбор цветов в пользовательском интерфейсе — тегом `cosmetic`.

Функциональность пометки ошибок должна поддерживать разные теги, потому что теги не обязательно являются взаимоисключающими. Дефект может влиять на несколько подсистем, например на производительность печати.

Антипаттерн: создание нескольких столбцов

Вам все еще нужно обеспечить хранение нескольких значений в атрибуте, но новое решение должно хранить только одно значение в каждом столбце. На первый взгляд логично создать в таблице несколько столбцов, каждый из которых содержит один тег.

Multi-Column/anti/create-table.sql

```
CREATE TABLE Bugs (
  bug_id SERIAL PRIMARY KEY,
  description VARCHAR(1000),
  tag1 VARCHAR(20),
  tag2 VARCHAR(20),
  tag3 VARCHAR(20)
);
```

При назначении тегов для заданной ошибки значения сохраняются в одном из трех столбцов. Неиспользуемые столбцы содержат `null`.

Multi-Column/anti/update.sql

```
UPDATE Bugs SET tag2 = 'performance' WHERE bug_id = 3456;
```

bug_id	description	tag1	tag2	tag3
1234	Сбой при сохранении	crash	NULL	NULL
3456	Повышение производительности	printing	performance	NULL
5678	Поддержка XML	NULL	NULL	NULL

Задачи, которые легко выполнить с обычными атрибутами, теперь несколько усложнились.

Поиск значений

При поиске ошибок для заданного тега необходимо провести поиск по всем трем столбцам, потому что строка тега может занимать один из этих столбцов. Например, для получения ошибок, относящихся к производительности, используется запрос следующего вида:

Multi-Column/anti/search.sql

```
SELECT * FROM Bugs
WHERE tag1 = 'performance'
  OR tag2 = 'performance'
  OR tag3 = 'performance';
```

Допустим, вам потребовалось найти ошибки, которым назначены оба тега, `performance` и `printing`. Для этого используется запрос следующего вида. Следите за правильным использованием круглых скобок, потому что `OR` имеет более низкий приоритет, чем `AND`.

Multi-Column/anti/search-two-tags.sql

```
SELECT * FROM Bugs
WHERE (tag1 = 'performance' OR tag2 = 'performance' OR tag3 = 'performance')
  AND (tag1 = 'printing' OR tag2 = 'printing' OR tag3 = 'printing');
```

Синтаксис поиска одного значения по нескольким столбцам получается слишком длинным и однообразным. Чтобы сделать его более компактным, можно применить предикат `IN` несколько нетрадиционным образом:

Multi-Column/anti/search-two-tags.sql

```
SELECT * FROM Bugs
WHERE 'performance' IN (tag1, tag2, tag3)
  AND 'printing' IN (tag1, tag2, tag3);
```

Добавление и удаление значений

Операции добавления и удаления значений из набора столбцов тоже создают проблемы. Простое изменение одного из столбцов при помощи `UPDATE` небезопасно, так как мы не знаем точно, какой столбец свободен (и есть ли такой столбец). Возможно, придется извлечь запись в приложение, чтобы узнать.

Multi-Column/anti/add-tag-two-step.sql

```
SELECT * FROM Bugs WHERE bug_id = 3456;
```

Например, в данном случае результат показывает, что `tag2` содержит `null`. После этого можно сформировать инструкцию `UPDATE`:

Multi-Column/anti/add-tag-two-step.sql

```
UPDATE Bugs SET tag2 = 'performance' WHERE bug_id = 3456;
```

Возникает риск того, что в момент запроса к таблице, но до ее обновления другой клиент пройдет через те же этапы чтения записи и ее обновления. В зависимости от того, кто первым применит обновление, у вас или у него может возникнуть конфликт обновлений или изменения будут перезаписаны другой стороной. Чтобы предотвратить подобные двухэтапные запросы, приходится использовать сложные выражения SQL.

Следующая инструкция использует функцию NULLIF(), помечающую каждый столбец как null, если в нем хранится определенное значение. NULLIF() — стандартная функция SQL, которая возвращает null в случае равенства двух аргументов.

Multi-Column/anti/remove-tag.sql

```
UPDATE Bugs
SET tag1 = NULLIF(tag1, 'performance'),
    tag2 = NULLIF(tag2, 'performance'),
    tag3 = NULLIF(tag3, 'performance')
WHERE bug_id = 3456;
```

Следующие инструкции добавляют новый тег performance в первый столбец, который пока что является null. Но если ни один из трех столбцов не равен null, то инструкция не вносит изменения в запись и новое значение тега вообще не сохраняется. Кроме того, писать такую инструкцию довольно сложно. Обратите внимание: строку performance приходится повторять шесть раз.

Multi-Column/anti/add-tag.sql

```
UPDATE Bugs
SET tag1 = CASE
      WHEN 'performance' IN (tag2, tag3) THEN tag1
      ELSE COALESCE(tag1, 'performance') END,
    tag2 = CASE
      WHEN 'performance' IN (tag1, tag3) THEN tag2
      ELSE COALESCE(tag2, 'performance') END,
    tag3 = CASE
      WHEN 'performance' IN (tag1, tag2) THEN tag3
      ELSE COALESCE(tag3, 'performance') END
WHERE bug_id = 3456;
```

Обеспечение уникальности

Скорее всего, одно значение не должно встречаться в нескольких столбцах, но при использовании антипаттерна «Многостолбцовые атрибуты» механизмов защиты от этого не существует. Иначе говоря, трудно избежать такой инструкции:

Multi-Column/anti/insert-duplicate.sql

```
INSERT INTO Bugs (description, tag1, tag2, tag3)
  VALUES ('printing is slow', 'printing', 'performance', 'performance');
```

Обработка расширяющихся наборов значений

Еще один недостаток рассматриваемого решения в том, что трех столбцов может быть недостаточно. Чтобы сохранить структуру с одним значением на столбец, необходимо задать столько столбцов, сколько тегов может быть у ошибки. Как узнать это значение на момент создания таблицы?

Возможная тактика — прикинуть количество столбцов и при необходимости позже добавить новые столбцы. Как правило, базы данных позволяют изменять структуру существующих таблиц, так что в случае чего вы сможете добавить `Bugs.tag4` и другие столбцы.

Multi-Column/anti/alter-table.sql

```
ALTER TABLE Bugs ADD COLUMN tag4 VARCHAR(20);
```

Однако это изменение обходится достаточно дорого по трем причинам:

- Реструктуризация таблиц в базе данных, которая уже содержит данные, может потребовать блокировки всей таблицы, что закроет доступ к ней всем параллельным клиентам.
- В некоторых базах данных подобная реструктуризация таблиц реализуется определением новой таблицы с нужной структурой, копированием данных из старой таблицы и удалением старой таблицы. Если таблица содержит большой объем данных, копирование может занять много времени.
- При добавлении столбца для многостолбцового атрибута вам придется вернуться к каждой инструкции SQL в каждом приложении, которое работает с этой таблицей, и отредактировать ее для поддержки новых столбцов.

Multi-Column/anti/search-four-columns.sql

```
SELECT * FROM Bugs
WHERE tag1 = 'performance'
  OR tag2 = 'performance'
  OR tag3 = 'performance'
  OR tag4 = 'performance'; -- необходимо добавить это новое условие
```

Все это требует скрупулезной работы и отнимает много времени. А если упустить какой-нибудь запрос, работающий со столбцами тегов, в систему могут вкрасться коварные ошибки.

Как распознать антипаттерн

Если в пользовательском интерфейсе или в документации проекта описывается атрибут, которому можно присвоить несколько значений, но их количество имеет фиксированный максимум, это может указывать на применение антипаттерна «Многостолбцовые атрибуты».

Следует признать, что у некоторых атрибутов количество возможных вариантов может ограничиваться намеренно, но чаще таких естественных ограничений не существует. Если какое-то ограничение кажется произвольным или неоправданным, возможно, здесь применен антипаттерн.

Об антипаттерне могут говорить следующие фразы:

- «Какое максимальное количество тегов нужно поддерживать?»

  Для многозначных атрибутов (например, тегов) необходимо заранее определить количество столбцов в таблице.
- «Как провести одновременный поиск по нескольким столбцам в SQL?»

  Если вы ищете заданное значение по нескольким столбцам, возможно, это говорит о том, что разные столбцы в действительности должны храниться в виде одного логического атрибута.

Паттерны среди антипаттернов

У антипаттернов «Кривая дорожка» и «Многостолбцовые атрибуты» есть кое-что общее. Оба являются решениями одной задачи: хранения атрибута, который может иметь несколько значений.

В антипаттерне «Кривая дорожка» в примерах использовались отношения «многие ко многим», тогда как в этой главе представлено простое отношение «один ко многим». Учтите, что оба антипаттерна могут применяться для обоих типов отношений.

Допустимые применения антипаттерна

В некоторых случаях атрибут может иметь фиксированное количество вариантов, и позиция или порядок этих вариантов могут иметь значение. Например, ошибка может быть связана с учетными записями нескольких пользователей, но природа этих связей уникальна. Один пользователь сообщает об ошибке, другой — разработчик, которому поручено ошибку исправить, а третий — инженер по контролю качества, который должен это исправление проверить. И хотя значения столбцов совместимы, из-за разной значимости и особенностей использования они фактически превращаются в логически разные атрибуты.

Вообще говоря, для хранения всех трех атрибутов можно было бы определить в таблице `Bugs` три обычных столбца. Недостатки, описанные в этой главе, не столь важны, потому что, скорее всего, вы будете использовать эти столбцы по отдельности. Возможно, в отдельных случаях вам все равно потребуется выполнить запрос по трем столбцам, например, чтобы вывести список всех, кто участвует в исправлении ошибки. На такую выборочную сложность можно пойти ради упрощения большинства операций.

Возможно и другое структурное решение — создание зависимой таблицы для множественных связей от таблицы `Bugs` к таблице `Accounts` и добавление в новую таблицу столбца для обозначения роли каждой учетной записи относительно данной ошибки. Однако эта структура может создать ряд проблем, описанных в главе 6 «Сущность — атрибут — значение».

Решение: создание зависимой таблицы

Как и в главе 2 «Кривая дорожка», лучшим решением будет создание зависимой таблицы с одним столбцом для многозначного атрибута. Храните множественные значения в нескольких строках (вместо нескольких столбцов). Кроме того, определите внешний ключ в зависимой таблице, чтобы связать значения с родительской строкой данных в таблице `Bugs`.

Multi-Column/soln/create-table.sql

```
CREATE TABLE Tags (
  bug_id BIGINT UNSIGNED NOT NULL
  tag VARCHAR(20),
  PRIMARY KEY (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);

INSERT INTO Tags (bug_id, tag)
  VALUES (1234, 'crash'), (3456, 'printing'), (3456, 'performance');
```

Когда все теги, связанные с ошибкой, находятся в одном столбце, поиск ошибок с заданным тегом становится проще.

Multi-Column/soln/search.sql

```
SELECT * FROM Bugs JOIN Tags USING (bug_id)
WHERE tag = 'performance';
```

Даже более сложные варианты поиска, например ошибка, связанная с двумя конкретными тегами, читаются относительно легко.

Multi-Column/soln/search-two-tags.sql

```
SELECT * FROM Bugs
  JOIN Tags AS t1 USING (bug_id)
  JOIN Tags AS t2 USING (bug_id)
WHERE t1.tag = 'printing' AND t2.tag = 'performance';
```

В этом случае связи добавляются и удаляются проще, чем с антипаттерном «Многостолбцовые атрибуты», — вы просто вставляете или удаляете строку из зависимой таблицы. Нет необходимости проверять несколько столбцов, чтобы понять, где можно добавить значение.

Multi-Column/soln/insert-delete.sql

```
INSERT INTO Tags (bug_id, tag) VALUES (1234, 'save');

DELETE FROM Tags WHERE bug_id = 1234 AND tag = 'crash';
```

Ограничение `PRIMARY KEY` гарантирует отсутствие дубликатов. Любой конкретный тег может быть применен к конкретной ошибке только один раз. При попытке вставить дубликат SQL вернет ошибку повторения ключа.

Вы не ограничены тремя тегами на ошибку, как с таблицей `Bugs` с тремя столбцами `tagN`. Теперь вы можете связать с одной ошибкой столько тегов, сколько потребуется.

Храните значения с одинаковым смыслом в одном столбце (вместо нескольких строк).

Мини-антипаттерн: хранение цен

Обычно структура реляционных баз данных не поощряет хранение избыточных данных. Лучше следовать правилам нормализации (см. приложение «Правила нормализации»), которые рекомендуют хранить столбец только в одной таблице. Некий факт представляется строкой данных и должен храниться в таблице только в одном экземпляре. Если тот же факт существует в нескольких местах базы данных, возникает риск расхождения двух экземпляров, когда становится неясно, какой из них содержит верную информацию. Нет ли у этого правила исключений?

Для примера возьмем таблицу `Orders`, в которой хранится запись клиента, приобретающего товар, в базе данных интернет-магазина. Пример включает столбец `price`, в котором содержится произведение цены товара на количество единиц, приобретенных пользователем.

Multi-Column/mini/orders.sql

```
CREATE TABLE Orders (
  order_id SERIAL PRIMARY KEY
  order_date DATE NOT NULL,
  customer_id INT NOT NULL,
  merchandise_id INT NOT NULL,
  quantity INT NOT NULL
  price NUMERIC(9,2) NOT NULL
);
```

Можно предположить, что цена единицы товара уже хранится в другой таблице `Merchandise`. Не будет ли избыточным хранить цену в таблице `Orders`? Они должны быть одинаковыми.

Допустим, цены могут изменяться. Цена, которую заплатил клиент в какой-то из дней, может отличаться от цены на тот же товар в следующем месяце и даже на следующий день. Кроме того, товар мог находиться на распродаже в день его покупки пользователем, так что цена была ниже. А может быть, клиент получил право на скидку по возрасту, заслугам или принадлежности к определенной категории. Цена в таблице `Orders` может отличаться от текущей цены в таблице `Merchandise` по множеству причин.

На самом деле это не нарушает правила о том, что каждый факт должен сохраняться однократно. Столбец таблицы `Orders` представляет другой факт по сравнению со столбцом таблицы `Merchandise`: в `Orders` хранится цена, которую клиент заплатил в день покупки после применения всех скидок.

В этом примере речь идет о ценах, но тот же принцип будет справедлив и в других сценариях. Состав спортивной команды ежегодно изменяется, и вы хотите знать, кто был в команде в каждом году (или даже в каждой игре), чтобы проанализировать результаты спортсмена. Актеры могут менять имена, поэтому они по-разному указываются в титрах в разных фильмах.

У этих сценариев есть общее: все они представляют собой таблицы пересечений для отношений «многие ко многим» (`Orders` — между клиентами и товарами; `Lineup` — между спортсменами и играми; `Cast` — между актерами и фильмами). Факт, хранящийся в таблице пересечений, относится к связи между двумя (или более) сущностями. Если один из атрибутов в будущем получит другое значение, необходимо будет сохранить значение, которое он имел на то время, оформив в виде атрибута в таблице пересечений.

Ну а можно ли не желать иметь много хорошего?

➤ *Уильям Шекспир, «Как вам это понравится»*

ГЛАВА 9

«Трибблы» метаданных

Моя жена — опытный разработчик Oracle PL/SQL и Java. Она рассказала мне об одном случае, когда использование структуры базы данных, которая должна была упростить работу, привело к тому, что работы стало еще больше.

В таблице `Customers`, с которой работал отдел продаж в ее компании, хранились такие данные, как контактная информация клиентов, тип предприятия и доход, полученный от клиента.

Metadata-Tribbles/intro/create-table.sql

```
CREATE TABLE Customers (
  customer_id NUMBER(9) PRIMARY KEY,
  contact_info VARCHAR(255),
  business_type VARCHAR(20),
  revenue NUMBER(9,2)
);
```

Отделу продаж понадобилось разбить данные по годам, чтобы отслеживать активных клиентов. Было решено добавить серию новых столбцов, при этом имя каждого столбца обозначало год, к которому относились данные:

Metadata-Tribbles/intro/alter-table.sql

```
ALTER TABLE Customers ADD (revenue2002 NUMBER(9,2));
ALTER TABLE Customers ADD (revenue2003 NUMBER(9,2));
ALTER TABLE Customers ADD (revenue2004 NUMBER(9,2));
```

Затем они ввели неполные данные — только для клиентов, которых им было интересно отслеживать. В большинстве строк в столбцах `revenue` остались значения `NULL`. Разработчики начали подумывать о том, чтобы хранить другую информацию в столбцах, которые почти не использовались.

Каждый год им приходилось добавлять еще один столбец. Администратор базы данных отвечал за управление табличными пространствами Oracle. Поэтому каждый год проводились собрания, планировалась миграция данных для из-

менения структуры табличного пространства и добавлялся новый столбец. В итоге все обернулось большими потерями времени и денег.

Цель: обеспечение масштабируемости

С увеличением объема данных производительность любого запроса базы данных снижается. Даже если запрос быстро возвращает результаты с несколькими тысячами записей, в таблице естественным образом накапливаются данные, и в какой-то момент этот же запрос перестанет выполняться за приемлемое время. Разумное использование индексирования помогает, но таблицы все равно разрастаются, и это отражается на скорости запросов к ним.

Чтобы повысить скорость выполнения запросов и эффективно поддерживать постоянно растущие таблицы, необходимо структурировать базу данных.

Антипаттерн: клонирование таблиц или столбцов

В сериале «Звездный путь» (Star Trek) («Star Trek» и связанные с ним знаки являются товарными знаками CBS Studios Inc.) были показаны «трибблы» — маленькие пушистые зверьки, которых держали как домашних питомцев. На первый взгляд трибблы очень симпатичные, но вскоре они начинают бесконтрольно размножаться, и избыток трибблов становится серьезной проблемой. Популяция трибблов заполняет любое отведенное ей место, и никто не желает нести за них ответственность. Со временем капитан Кирк обнаруживает, что его корабль и экипаж не могут нормально работать, и он вынужден приказать в первую очередь очистить корабль от трибблов.

Таблицы и столбцы баз данных порой становятся похожими на трибблов и начинают бесконтрольно плодиться, если в структуре базы данных для каждого последующего значения данных создаются новые таблицы или столбцы.

По собственному опыту мы знаем, что запрос к таблице с несколькими строками при прочих равных условиях выполняется быстрее, чем запрос к таблице с множеством строк. Так возникает распространенное заблуждение, что в таблице нужно хранить поменьше строк независимо от того, что с ней делается. Так мы приходим к двум формам антипаттерна:

- Разбиение одной длинной таблицы на несколько меньших таблиц. Имена таблиц определяются разными значениями данных в одном из атрибутов таблиц.
- Разбиение одного столбца на несколько столбцов, чьи имена определяются значениями другого атрибута.

Однако ничто не дается даром; чтобы добиться желаемого и хранить в каждой таблице небольшое количество строк, придется либо создать таблицы, содер-

жащие слишком много столбцов, либо увеличить количество таблиц. В обоих случаях оказывается, что количество таблиц или столбцов продолжает расти, так как новые значения данных могут потребовать создания новых объектов схемы.

Размножение таблиц

Чтобы разбить данные по разным таблицам, понадобится политика, которая определяет, какие строки данных принадлежат тем или иным таблицам. Например, можно провести разбиение по году в столбце `date_reported`:

Metadata-Tribbles/anti/create-tables.sql

```
CREATE TABLE Bugs_2019 ( . . . );
CREATE TABLE Bugs_2020 ( . . . );
CREATE TABLE Bugs_2021 ( . . . );
```

При вставке строк в базу данных теперь необходимо следить за тем, чтобы использовать правильную таблицу в зависимости от вставляемых значений:

Metadata-Tribbles/anti/insert.sql

```
INSERT INTO Bugs_2021 (..., date_reported, ...)
VALUES (..., '2021-06-01', ...);
```

Перенесемся в 1 января следующего года. Во всех новых баг-репортах возникает ошибка, потому что вы забыли создать таблицу `Bugs_2023`.

Metadata-Tribbles/anti/insert.sql

```
INSERT INTO Bugs_2022 (..., date_reported, ...)
VALUES (..., '2022-02-20', ...);
```

Это означает, что новое значение *данных* может требовать нового объекта *метаданных*. Такая связь нетипична для отношений между данными и метаданными в SQL.

Управление целостностью данных

Предположим, что ваш руководитель пытается подсчитать ошибки, о которых сообщали в этом году, но суммы не сходятся. Выясняется, что часть ошибок 2022 года случайно попала в таблицу `Bugs_2021`. Следующий запрос всегда должен возвращать пустой результат; если возвращается что-то другое, это говорит о наличии проблемы:

Metadata-Tribbles/anti/data-integrity.sql

```
SELECT * FROM Bugs_2021
WHERE date_reported NOT BETWEEN '2021-01-01' AND '2021-12-31';
```

Невозможно автоматически ограничить данные относительно имени таблицы, но можно объявить ограничение CHECK в каждой из таблиц:

Metadata-Tribbles/anti/check-constraint.sql

```
CREATE TABLE Bugs_2021 (
  -- другие столбцы
  date_reported DATE CHECK (EXTRACT(YEAR FROM date_reported) = 2021)
);

CREATE TABLE Bugs_2022 (
  -- другие столбцы
  date_reported DATE CHECK (EXTRACT(YEAR FROM date_reported) = 2022)
);
```

Не забудьте скорректировать значение в ограничении CHECK при создании Bugs_2023. Если вы допустите ошибку, это может привести к созданию таблицы, отклоняющей строки данных, которые она должна принимать.

Синхронизация данных

Аналитик службы поддержки просит изменить дату в сообщении об ошибке. В базе данных она хранится с датой 2022-01-03, но сообщивший о ней клиент в действительности отправил ее неделей раньше: 2021-12-27.

Вы обновляете дату простой инструкцией UPDATE:

Metadata-Tribbles/anti/anomaly.sql

```
UPDATE Bugs_2012
SET date_reported = '2021-12-27'
WHERE bug_id = 1234;
```

К сожалению, это исправление делает строку данных недействительной для таблицы Bugs_2022. Вам придется удалять строку из одной таблицы и вставлять ее в другую таблицу в тех редких случаях, когда простая инструкция UPDATE приводит к этой аномалии.

Metadata-Tribbles/anti/synchronize.sql

```
INSERT INTO Bugs_2021 (bug_id, date_reported, ...)
  SELECT bug_id, date_reported, ...
  FROM Bugs_2022
  WHERE bug_id = 1234;

DELETE FROM Bugs_2022 WHERE bug_id = 1234;
```

Обеспечение уникальности

Следите, чтобы значения первичного ключа были уникальными по всем таблицам, полученным в результате разделения. Если потребуется переместить

строку из одной таблицы в другую, необходимо обеспечить, чтобы значение первичного ключа не конфликтовало с другой строкой.

Если в базе данных поддерживаются объекты последовательностей, можно использовать одну последовательность для генерирования значений для всех полученных таблиц. Для баз данных, поддерживающих только уникальность идентификаторов на уровне таблиц, придется использовать более неудобное решение. Необходимо создать новую таблицу просто для генерирования значений первичного ключа:

Metadata-Tribbles/anti/id-generator.sql

```
CREATE TABLE BugsIdGenerator (bug_id SERIAL PRIMARY KEY);

INSERT INTO BugsIdGenerator (bug_id) VALUES (DEFAULT);
ROLLBACK;

INSERT INTO Bugs_2022 (bug_id, . . .)
  VALUES (LAST_INSERT_ID(), . . .);
```

Запросы к разным таблицам

Рано или поздно руководителю понадобится запрос, ссылающийся на несколько таблиц. Например, он может запросить количество всех открытых ошибок независимо от того, в каком году они были зарегистрированы. Полный набор ошибок можно сформировать, используя `UNION` со всеми таблицами, полученными в результате разделения, и делая запрос как к производной таблице:

Metadata-Tribbles/anti/union.sql

```
SELECT b.status, COUNT(*) AS count_per_status FROM (
  SELECT * FROM Bugs_2020
    UNION
  SELECT * FROM Bugs_2021
    UNION
  SELECT * FROM Bugs_2022 ) AS b
GROUP BY b.status;
```

Со временем и по мере создания новых таблиц, таких как на `Bugs_2023`, вам придется следить, чтобы код приложения корректно работал со вновь создаваемыми таблицами.

Синхронизация метаданных

Шеф требует добавить столбец, хранящий информацию о том, сколько рабочего времени потребовалось для исправления каждой ошибки.

Metadata-Tribbles/anti/alter-table.sql

```
ALTER TABLE Bugs_2021 ADD COLUMN hours NUMERIC(9,2);
```

Если таблица была разбита, новый столбец относится только к одной изменяемой таблице. Во всех остальных таблицах его нет.

Если использовать запрос `UNION` по полученным в результате разделения таблицам, как в случае выше, возникнет новая проблема: объединение таблиц при помощи `UNION` возможно в том случае, если они содержат одинаковые столбцы. Если столбцы различаются, нужно указывать только общие столбцы всех таблиц без использования универсального символа *.

Управление ссылочной целостностью

Если зависимая таблица (например, `Comments`) ссылается на `Bugs`, зависимая таблица не может объявить внешний ключ. Во внешнем ключе должна быть объявлена одна таблица, но в данном случае родительская таблица разбивается на множество таблиц.

Metadata-Tribbles/anti/foreign-key.sql

```
CREATE TABLE Comments (
  comment_id        SERIAL PRIMARY KEY,
  bug_id            BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs_????(bug_id)
);
```

В разбитой таблице также могут возникнуть проблемы из-за того, что она является зависимой, а не родительской. Например, `Bugs.reported_by` ссылается на таблицу `Accounts`. Если необходимо получить все ошибки, о которых сообщил определенный человек независимо от года, используйте запрос следующего вида:

Metadata-Tribbles/anti/join-union.sql

```
SELECT * FROM Accounts a
JOIN (
    SELECT * FROM Bugs_2019
    UNION ALL
    SELECT * FROM Bugs_2020
    UNION ALL
    SELECT * FROM Bugs_2021
  ) t ON (a.account_id = t.reported_by)
```

Идентификация столбцов-«трибблов» метаданных

Столбцы тоже могут стать «трибблами» метаданных. Некоторые столбцы по своей природе обречены на размножение, о котором говорилось в начале главы.

Другой пример, который может встречаться в базе данных ошибок, таблица для хранения сводных данных метрик проекта, отдельные столбцы которой содержат

подытоги. Например, в следующей таблице вам со временем неизбежно понадобится добавить столбец `bugs_fixed_2022`:

Metadata-Tribbles/anti/multi-column.sql

```
CREATE TABLE ProjectHistory (
  bugs_fixed_2019 INT,
  bugs_fixed_2020 INT,
  bugs_fixed_2021 INT
);
```

Как распознать антипаттерн

Следующие фразы могут указывать на то, что в базе данных разрастается антипаттерн «Трибблы метаданных»:

- «Тогда нужно будет создать таблицу (или столбец) для каждого...»

  Если вы описываете базу данных и используете выражение «для каждого» в этом контексте, это означает, что таблицы разбиваются по отдельным значениям одного из столбцов.

Объединение метаданных с данными

Обратите внимание: присоединяя год к базовому имени таблицы, мы комбинируем значение данных с идентификатором метаданных.

Этот пример противоположен объединению данных с метаданными, которое встречалось в описаниях антипаттернов «Сущность — атрибут — значение» и «Полиморфная связь». В тех случаях мы сохраняли идентификаторы метаданных (имя столбца и имя таблицы) в виде строковых данных.

В антипаттернах «Многостолбцовые атрибуты» и «Трибблы метаданных» значение данных превращается в имя столбца или имя таблицы. Любые из этих антипаттернов создадут больше проблем, чем решат.

- «Сколько таблиц (или столбцов) максимум поддерживает база данных?»

  Многие базы данных способны работать с большим количеством таблиц и столбцов, чем вам когда-либо понадобится (если создавать архитектуру грамотно). Если вы опасаетесь превысить максимум, скорее всего, базу данных стоит перепроектировать.
- «Мы поняли, почему приложение не смогло добавить новые данные сегодня утром: мы забыли создать таблицу для нового года».

Это типичное следствие антипаттерна «Трибблы метаданных». Если для новых данных приходится создавать новые объекты баз данных, эти объекты необходимо определять заранее; в противном случае возникает опасность непредвиденных сбоев.

- «Как выполнить запрос для поиска по нескольким таблицам сразу? Во всех этих таблицах одинаковые столбцы».

  Если вам приходится проводить поиск по многим таблицам с идентичной структурой, их стоит сохранить в одной таблице и добавить один дополнительный столбец атрибута для различения этих строк.

- «Как передать параметр для имени таблицы? Мне нужно создать запрос по имени таблицы, к которому динамически присоединяется префикс года».

  Если бы эти данные находились в одной таблице, вам не пришлось бы этого делать.

Допустимые применения антипаттерна

Ручное разбиение таблиц может пригодиться для *архивации* — удаления исторических данных из повседневных операций. Обычно, когда данные перестают быть актуальными, необходимость в запросах к ним значительно сокращается. Если вам достаточно только текущих данных, целесообразно скопировать старые данные в другое место и удалить их из активных таблиц. После архивации данные продолжают храниться в совместимой структуре таблиц, чтобы их можно было анализировать время от времени, но запросы только к текущим данным выполняются быстрее.

Шардинг баз данных в WordPress.com

На конференции MySQL Conference & Expo я обедал с Барри Абрахамсоном (Barry Abrahamson), архитектором баз данных WordPress.com — популярного сервиса хостинга программных средств для ведения блогов.

Барри сказал, что в самом начале работы данные всех клиентов хранились в одной базе данных. В конце концов, контент одной блогерской площадки был не таким уж большим. Казалось вполне логичным, что будет удобнее иметь одну базу данных.

На начальной стадии развития сайта такое решение неплохо работало, но вскоре масштаб операций значительно увеличился. В настоящее время сервис обслуживает 7 миллионов блогов на 300 серверах баз данных. На каждом сервере размещаются данные подмножества клиентов.

При добавлении нового сервера Барри было бы намного труднее выделить из единой базы данных информацию, относящуюся к блогу отдельного клиента. Разбиение данных с созданием отдельной базы данных для каждого клиента значительно упростило перемещение любого отдельного блога с одного сервера на другой. Со временем клиенты

появляются и исчезают, блоги одних клиентов получают высокую нагрузку, а другие уходят в небытие. Работа по перераспределению нагрузки на множестве серверов стала еще более важной.

Создать резервную копию и восстановить отдельную базу данных умеренного размера проще, чем одну базу данных с терабайтами данных. Представьте, что клиент обращается в службу поддержки и говорит, что в его данных полный бардак из-за неверного ввода. Как Барри восстановить данные одного клиента, если все клиенты совместно используют одну монолитную резервную копию базы данных?

Хотя может показаться, что с точки зрения моделирования данных правильно хранить их все в одной БД, грамотное разбиение базы данных упрощает ее администрирование после того, как размер базы данных превысит определенный порог.

Решение: партицирование и нормализация

Помимо ручного разбиения таблиц, существуют и более эффективные способы повысить производительность, если таблица стала слишком большой. Среди них можно выделить горизонтальное партицирование, вертикальное партицирование и использование зависимых таблиц.

Горизонтальное партицирование

Чтобы получить только преимущества разбиения большой таблицы, без недостатков, можно воспользоваться функциональностью, которая называется *горизонтальным партицированием* или *шардингом*. Вы определяете логическую таблицу с некоторым правилом для разделения строк на отдельные секции, а база данных делает все остальное. Физически таблица разбивается, но вы можете выполнять для нее инструкции SQL так, словно она является единым целым.

Горизонтальное партицирование обеспечивает гибкость в определении способов выделения строк каждой отдельной таблицы в отдельное хранилище. Например, поддержка партицирования в MySQL позволяет задать секции в необязательной части инструкции `CREATE TABLE`.

Metadata-Tribbles/soln/horiz-partition.sql

```
CREATE TABLE Bugs (
  bug_id SERIAL PRIMARY KEY,
  -- другие столбцы
  date_reported DATE
) PARTITION BY HASH ( bug_id )
  PARTITIONS 4;
```

В приведенном примере реализуется партицирование, сходное с тем, что мы видели ранее в этой главе; строки разделяются по году в столбце `date_reported`.

Однако преимущества от ручного разбиения таблицы в том, что строки никогда не попадут по ошибке в другую таблицу, даже если значение столбца `date_reported` будет обновлено, а вы можете выполнять запросы к таблице `Bugs` без необходимости ссылаться на отдельные таблицы.

В этом примере для количества отдельных физических таблиц, используемых для хранения строк, установлено фиксированное значение 4. Если у вас строки охватывают более четырех лет, одна из секций будет использоваться для хранения данных за несколько лет. Так будет продолжаться из года в год. Вам не нужно добавлять новые секции, если только объем данных не станет настолько большим, что вы почувствуете необходимость дальнейшего разбиения.

Партицирование не определено в стандарте SQL, так что каждая разновидность базы данных реализует его собственным нестандартным образом. Терминология, синтаксис и конкретные особенности партицирования зависят от конкретного продукта. Тем не менее некоторые формы партицирования поддерживаются всеми популярными базами данных.

Вертикальное партицирование

Если горизонтальное партицирование разбивает таблицу по строкам, то вертикальное разбивает ее по столбцам. Разбиение таблиц по столбцам может иметь свои преимущества, если некоторые столбцы слишком громоздкие или редко используются.

Столбцы `BLOB` и `TEXT` имеют переменный размер и могут быть очень большими. Ради эффективности хранения и загрузки данных многие базы данных автоматически сохраняют столбцы этих типов данных отдельно от других столбцов заданной строки. Если вы выполняете запрос без ссылок на столбцы `BLOB` или `TEXT` таблицы, это повышает эффективность обращения к другим столбцам. Если использовать в запросе шаблон столбца `*`, база данных загрузит все столбцы из таблицы, включая все столбцы `BLOB` или `TEXT`.

Например, в таблице `Products` базы данных ошибок можно сохранить копию установочного файла для соответствующего продукта. Файл обычно представляет собой самораспаковывающийся архив с расширением `.exe` (Windows) или `.dmg` (Mac). Обычно эти файлы очень велики, но в столбце `BLOB` могут храниться двоичные данные колоссального размера.

Вполне логично, что установочный файл должен быть атрибутом таблицы `Products`. Но в большинстве запросов к этой таблице установочный файл не понадобится. Хранение в таблице `Products` такого большого объема данных, которые используются относительно редко, может привести к непредвиденным проблемам с производительностью, если вы привыкли загружать все столбцы при помощи шаблона `*`.

Проблема решается хранением столбца BLOB в другой таблице, отдельно от таблицы Products, но в зависимости от нее. Также назначьте его первичный ключ внешним ключом для таблицы Products; это гарантирует, что для каждой строки продукта существует не более одной связанной строки.

Metadata-Tribbles/soln/vert-partition.sql

```
CREATE TABLE ProductInstallers (
  product_id BIGINT UNSIGNED PRIMARY KEY,
  installer_image BLOB,
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

Мы привели пример крайнего случая, чтобы подчеркнуть суть, но он демонстрирует преимущества хранения некоторых столбцов в отдельных таблицах. Например, в механизме хранения MyISAM базы данных MySQL запрос к таблице работает наиболее эффективно, когда строки имеют фиксированный размер. Тип данных VARCHAR имеет переменную длину, так что наличие в таблице одного столбца с таким типом данных не позволяет таблице воспользоваться этим преимуществом. Если хранить все столбцы переменной длины в отдельной таблице, то это может оптимизировать выполнение запросов к первичной таблице (хотя бы немного).

Metadata-Tribbles/soln/separate-fixed-length.sql

```
CREATE TABLE Bugs (
  bug_id        SERIAL PRIMARY KEY, -- тип данных фиксированной длины
  summary       CHAR(80), -- тип данных фиксированной длины
  date_reported DATE, -- тип данных фиксированной длины
  reported_by   BIGINT UNSIGNED, -- тип данных фиксированной длины
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
);
CREATE TABLE BugDescriptions (
  bug_id      BIGINT UNSIGNED PRIMARY KEY,
  description VARCHAR(1000), -- тип данных переменной длины
  resolution  VARCHAR(1000), -- тип данных переменной длины
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

Исправление столбцов «трибблов метаданных»

По аналогии с решением, приведенным в главе 8 «Многостолбцовые атрибуты», лучшим решением проблемы столбцов «Трибблы метаданных» является создание зависимой таблицы.

Metadata-Tribbles/soln/create-history-table.sql

```
CREATE TABLE  ProjectHistory (
  project_id  BIGINT,
  year        SMALLINT,
```

```
  bugs_fixed  INT,
  PRIMARY KEY (project_id, year),
  FOREIGN KEY (project_id) REFERENCES Projects(project_id)
);
```

Вместо одной строки на проект с несколькими столбцами за каждый год используйте несколько строк с одним столбцом для исправленных ошибок. Так вам не придется добавлять новые столбцы для последующих лет. С течением времени в этой таблице можно хранить любое количество строк для каждого проекта и без необходимости ежегодно модифицировать запросы.

Не позволяйте данным порождать метаданные.

ЧАСТЬ II

Антипаттерны физического проектирования баз данных

Когда вы знаете, какие данные должны храниться в базе, вы можете максимально эффективно управлять данными, используя средства своей РСУБД. В частности, это означает определение таблиц и индексов, а также выбор типов данных. При этом используется язык определения данных в SQL — такие инструкции, как `CREATE TABLE`.

Умножая 0.1 на 10, вы практически никогда
не получите 1.0.

➤ *Брайан Керниган (Brian Kernighan)*

ГЛАВА 10

Ошибки округления

Руководитель требует отчета о стоимости рабочего времени разработчиков, занятых в проекте, на основе общего объема работы по исправлению каждой ошибки. У всех разработчиков из таблицы `Accounts` разная почасовая ставка, поэтому вы определяете количество часов, необходимых для исправления каждой ошибки, в таблице `Bugs` и умножаете его на почасовую ставку `hourly_rate` разработчика, которому поручено выполнение работы.

Rounding-Errors/intro/cost-per-bug.sql

```
SELECT b.bug_id, b.hours * a.hourly_rate AS cost_per_bug
FROM Bugs AS b
  JOIN Accounts AS a ON (b.assigned_to = a.account_id);
```

Чтобы поддерживать этот запрос, необходимо создать новые столбцы в таблицах `Bugs` и `Accounts`. Оба столбца должны поддерживать дробные значения, потому что затраты должны отслеживаться с высокой точностью. Новые столбцы должны определяться с типом `FLOAT`, потому что этот тип данных поддерживает дроби.

Rounding-Errors/intro/float-columns.sql

```
ALTER TABLE Bugs ADD COLUMN hours FLOAT;

ALTER TABLE Accounts ADD COLUMN hourly_rate FLOAT;
```

Вы обновляете столбцы информацией из журналов исправления ошибок и ставок разработчиков, тестируете отчет и считаете работу выполненной.

На следующий день шеф приходит к вам в кабинет с копией отчета. «Не сходится, — цедит он. — Я пересчитал суммы вручную, и оказалось, что ваш отчет неточен — немного, всего на несколько долларов». Вас бросает в пот: кажется, вам придется объяснять, как можно было ошибиться в таких простых вычислениях.

Цель: дроби вместо целых чисел

Целые числа — полезный тип данных, но в нем могут храниться только такие числа, как 1, 327 или –19. Он не позволяет представлять дроби, например 2.5. Если вы работаете с числами с точностью, превышающей точность целого типа, вам понадобится другой тип данных. Например, денежные суммы обычно представляются целыми числами с двумя знаками в дробной части — $19.95 и т. д.

Таким образом, требуется хранить числовые значения, которые не являются целыми, и использовать их в арифметических вычислениях. Еще одна цель, которая обычно явно не формулируется: результаты арифметических вычислений должны быть *верными*.

Антипаттерн: использование типа данных FLOAT

В большинстве языков программирования поддерживается тип данных для представления вещественных чисел; обычно он называется `float` или `double`. В SQL также поддерживается похожий тип данных с тем же именем. Многие разработчики используют тип данных SQL `FLOAT` всегда, когда им требуются дробные числовые данные, потому что они привыкли программировать с типом данных `float`.

Тип данных `FLOAT` в SQL, как и `float` в большинстве языков программирования, кодирует вещественное число в двоичном формате в соответствии со стандартом IEEE 754. Чтобы эффективно использовать числа с плавающей точкой, необходимо понимать некоторые их характеристики.

Округление по необходимости

Многие разработчики не знают об одной особенности формата с плавающей точкой: не все значения, которые можно описать в десятичном виде, могут быть сохранены в двоичном виде. Некоторые числа приходится округлять до очень близкого значения.

Чтобы представить некоторый контекст для этого поведения округления, сравните с рациональными числами (например, 1/3), которые в десятичной системе представляются числом с бесконечным повторением цифр, например 0.333... Истинное значение нельзя представить в десятичном виде, потому что это потребует бесконечного количества цифр. Количество цифр определяет *точность* представления числа, так что повторяющееся десятичное число потребовало бы *бесконечной точности*.

Приходится идти на компромисс и использовать *конечную точность*, то есть выбирать числовое значение, по возможности близкое к исходному, например

0.333. Однако это означает, что представление не совсем точно соответствует представляемому числу.

```
1/3 + 1/3 + 1/3       = 1.000
0.333 + 0.333 + 0.333 = 0.999
```

Даже если увеличить точность, вы все равно не сможете сложить три приближения 1/3, чтобы получить точное значение 1.0. Этот компромисс неизбежен при использовании конечной точности для представления чисел с повторяющимися цифрами.

```
1/3 + 1/3 + 1/3       = 1.000000
0.333333 + 0.333333 + 0.333333 = 0.999999
```

Это означает, что некоторые числа невозможно представить с конечной точностью. Кто-то скажет, что это нормально, потому что все равно нельзя ввести число с бесконечным количеством цифр; естественно предположить, что любое число, которое вводится с клавиатуры, имеет конечную точность и должно храниться точно. К сожалению, это не так.

Стандарт IEEE 754 представляет числа с плавающей точкой по основанию 2. Значения, требующие конечной точности в десятичной записи (например, 59.95), для точного представления в двоичной записи могут потребовать бесконечной точности. Тип данных `FLOAT` на это не способен, поэтому используется ближайшее значение по основанию 2, которое он может сохранить; в десятичной записи оно равно 59.950000762939.

Так уж совпало, что некоторые значения могут быть представлены с конечной точностью в обоих форматах. Теоретически при понимании того, как числа хранятся в формате IEEE 754, вы сможете предсказать, как заданное десятичное значение представляется в двоичном виде. На практике многие люди не проделывают эти вычисления для каждого используемого числа с плавающей точкой. Невозможно гарантировать, что в столбце `FLOAT` в базе данных будут сохраняться только «хорошие» значения, поэтому приложение должно предполагать, что любое значение в этом столбце могло подвергнуться округлению.

Некоторые базы данных поддерживают сходные типы данных `DOUBLE PRECISION` и `REAL`. Точность, обеспечиваемая этими типами данных и `FLOAT`, изменяется в зависимости от реализации базы данных, но все они представляют значения с плавающей точкой с конечной точностью в двоичных цифрах, и поэтому все они ведут себя похожим образом при округлении.

Использование FLOAT в SQL

Некоторые базы данных компенсируют неточность и выводят нужное значение.

Rounding-Errors/anti/select-rate.sql

```
SELECT hourly_rate FROM Accounts WHERE account_id = 123;
```

Возвращаемое значение: `59.95`

Фактическое значение, хранящееся в столбце `FLOAT`, может не совпадать с этим значением с абсолютной точностью. Умножив значение на миллиард, вы заметите расхождения:

Rounding-Errors/anti/magnify-rate.sql

```
SELECT hourly_rate * 1000000000 FROM Accounts WHERE account_id = 123;
```

Возвращаемое значение: `59950000762.939`

Можно было бы ожидать, что увеличенное значение, возвращаемое приведенным запросом, будет равно `59950000000.000`. Это показывает, что значение `59.95` было округлено до значения, которое можно представить с конечной точностью, обеспечиваемой двоичным форматом IEEE 754. В данном случае расхождение не превышает 1/1000000, чего достаточно для большинства вычислений.

Тем не менее для других вычислений даже этого может оказаться недостаточно. Классический пример — использование `FLOAT` при проверке равенства.

Rounding-Errors/anti/inexact.sql

```
SELECT * FROM Accounts WHERE hourly_rate = 59.95;
```

Результат: пустое множество; совпадения отсутствуют.

Выше мы видели, что значение, хранящееся в `hourly_rate`, в действительности чуть больше `59.95`. Таким образом, хотя для `account_id` 123 этому столбцу было присвоено значение `59.95`, сейчас данные этого столбца не совпадают с запросом.

Одно из распространенных обходных решений этой проблемы — считать, что значения с плавающей точкой «фактически равны», если они достаточно близки друг к другу в пределах небольшого порога. Вычтите одно значение из другого и удалите знак из разности при помощи функции модуля `ABS()`. Если результат равен нулю, значит, два значения в точности равны. Если результат достаточно мал, два значения считаются «фактически равными». Следующий запрос успешно находит строку данных:

Rounding-Errors/anti/threshold.sql

```
SELECT * FROM Accounts WHERE ABS(hourly_rate - 59.95) < 0.000001;
```

Однако разность достаточно велика, чтобы сравнение с большей точностью оказалось неудачным:

Rounding-Errors/anti/threshold.sql

```
SELECT * FROM Accounts WHERE ABS(hourly_rate - 59.95) < 0.0000001;
```

Порог изменяется, потому что изменяется модуль разности между значением по основанию 10 и округленным значением по основанию 2.

Другой пример того, как неточная природа `FLOAT` создает проблемы с точностью, — вычисление агрегатных функций для многих значений. Например, если использовать `SUM()` для суммирования значений с плавающей точкой в столбце, в сумме накапливаются погрешности, обусловленные округлением всех значений.

Rounding-Errors/anti/cumulative.sql

```
SELECT SUM( b.hours * a.hourly_rate ) AS project_cost
FROM Bugs AS b
JOIN Accounts AS a ON (b.assigned_to = a.account_id);
```

Формат IEEE 754

Предложения по стандартизации двоичного формата чисел с плавающей точкой начали появляться еще в 1979 году. Формально стандарт был принят в 1985 году, а сейчас он широко применяется в различных программах, большинстве языков программирования и микропроцессорах.

В этом формате для кодирования числа с плавающей точкой используются три поля: *мантисса*, *экспонента* (степень, в которую возводится мантисса) и одноразрядное поле *знака*.

Одно из преимуществ IEEE 754 заключается в том, что с экспонентой можно представлять как очень малые, так и очень большие значения мантиссы. Формат не только поддерживает вещественные числа — диапазон поддерживаемых значений больше целых чисел в формате с фиксированной точкой. Формат с двойной точностью поддерживает еще больший диапазон значений. Таким образом, эти форматы хорошо подходят для научных приложений.

Вероятно, чаще всего дроби применяются при представлении денежных сумм. Использовать IEEE 754 для денежных сумм не нужно, потому что масштабируемый десятичный формат, описанный в этой главе, позволяет работать с денежными величинами так же просто и с большей точностью.

Если вы хотите узнать больше об этом формате, рекомендуем статью о IEEE 754 в Википедии[1] и статью Дэвида Голдберга *What Every Computer Scientist Should Know About Floating-Point Arithmetic [Gol91]*.

[1] https://ru.wikipedia.org/wiki/IEEE_754-2008

Совокупное воздействие неточных чисел с плавающей точкой становится еще более серьезным при вычислении агрегатного произведения множества чисел (вместо суммы). Разности кажутся небольшими, но они накапливаются. Например, если умножить значение `1` на точное значение `1.0`, результат всегда равен `1`. Неважно, сколько раз будет выполняться умножение. Но если множитель в действительности равен `0.999`, результат будет другим. Если умножить значение `1` на `0.999` тысячу раз подряд, результат будет равен приблизительно `0.3677`. Чем больше раз выполняется умножение, тем значительнее расхождение.

Хороший пример многократного умножения — расчет сложного процента в финансовых приложениях. Неточные числа с плавающей точкой создают погрешность, которая кажется крошечной, но растет при возведении в степень. Поэтому в финансовых приложениях особенно важно использовать точные значения.

Как распознать антипаттерн

Практически любое использование типов данных `FLOAT`, `REAL` или `DOUBLE PRECISION` должно насторожить. Во многих приложениях, использующих числа с плавающей точкой, просто не нужны диапазоны значений, поддерживаемые форматами IEEE 754.

Мысль об использовании типа данных `FLOAT` в SQL выглядит логично, так как подобный тип встречается во многих языках программирования. Тем не менее существует и более эффективный вариант.

Допустимые применения антипаттерна

Тип данных `FLOAT` хорошо подходит для числовых значений, выходящих за пределы диапазонов представления типов `INTEGER` или `NUMERIC`. В научных приложениях использование `FLOAT` актуальнее всего. Например, в базе данных температур ежедневные показания термометра обычно представляют собой дробные числа. Хранить температуру с точностью до миллиардной доли градуса вполне нормально, потому что эти данные обычно используются для таких агрегатных вычислений, как `MIN()`, `MAX()` или `AVG()`. Также можно провести поиск строк, в которых температура принадлежит диапазону значений, и запрос, использующий неравенство или `BETWEEN`, будет верно работать с неточными сравнениями. Но вы вряд ли будете искать строки, в которых температура в точности равна какому-то конкретному значению. Oracle использует тип данных `FLOAT` для обозначения точного масштабируемого числового типа, тогда как тип данных `BINARY_FLOAT` представляет неточное числовое значение в кодировке IEEE 754.

Решение: тип данных NUMERIC

Вместо `FLOAT` и его родственников используйте тип данных SQL `NUMERIC` или `DECIMAL` для дробных чисел с фиксированной точностью.

Rounding-Errors/soln/numeric-columns.sql

```
ALTER TABLE Bugs ADD COLUMN hours NUMERIC(9,2);
ALTER TABLE Accounts ADD COLUMN hourly_rate NUMERIC(9,2);
```

Эти типы данных хранят числовые значения с *точностью* (precision), указанной в определении столбца. Точность определяется в аргументе типа данных — по аналогии с синтаксисом, который используется для длины типа данных `VARCHAR`. Точность равна общему количеству цифр, используемых в значении этого столбца. Так, точность 9 означает, что в столбце может храниться значение `123456789`, но не `1234567890`.

Также можно передать во втором аргументе типа данных *масштаб* (scale) — количество цифр в дробной части числа. Эти цифры включены в точность, так что точность 9 с масштабом 2 означает, что можно сохранить значение `1234567.89`, но не `12345678.91` или `123456.789`.

Заданные точность и масштаб применяются к столбцу во всех строках таблицы. Иначе говоря, не получится хранить значения с масштабом 2 в одних строках и с масштабом 4 в других. В SQL тип данных столбца обычно применяется ко всем строкам (точно так же, как если столбец определен с `VARCHAR(20)`, эта длина является допустимой для каждой строки).

Преимущество `NUMERIC` и `DECIMAL` заключается в том, что они позволяют хранить рациональные числа без округления. После присвоения значения `59.95` можно быть уверенным, что сохраняется точное значение. Его проверка на равенство с литеральным значением `59.95` будет успешной.

Rounding-Errors/soln/exact.sql

```
SELECT hourly_rate FROM Accounts WHERE hourly_rate = 59.95;
```

Возвращаемое значение: `59.95`

Если умножить значение на миллиард, получится ожидаемое значение:

Rounding-Errors/soln/magnify-rate-exact.sql

```
SELECT hourly_rate * 1000000000 FROM Accounts WHERE hourly_rate = 59.95;
```

Возвращаемое значение: `59950000000`

Типы данных `NUMERIC` и `DECIMAL` ведут себя одинаково; между ними не должно быть различий. Вместо `DECIMAL` также можно использовать синоним `DEC`.

Сохранять значения, требующие бесконечной точности (например, 1/3), все еще не получится. По крайней мере нам стали более знакомы значения, на которые распространяется это ограничение в дробном формате.

Если вам нужны точные дробные значения, используйте тип данных `NUMERIC`. Тип данных `FLOAT` не способен представлять многие дробные рациональные числа, поэтому с ними приходится работать как с неточными значениями.

Если бы мне давали 10 центов каждый раз, когда я вижу, как кто-нибудь использует FLOAT для хранения денежных сумм, то у меня бы уже накопилось $1000.0000000001588.

Научный подход применим, когда переменные исчислимы, число их невелико, а их комбинации четко выражены и понятны.

➤ *Поль Валери (Paul Valéry)*

ГЛАВА 11

31 ВКУС

Обращение в таблице контактов — хороший пример столбца с ограниченным количеством возможных значений. Поддерживая основные варианты — `Mr.`, `Mrs.`, `Ms.`, `Dr.`, `Rev.` , — вы учитываете практически все возможности. Можно задать этот список в определении столбца, используя тип данных или ограничение, чтобы никто случайно не ввел недопустимую строку в столбце `salutation` (обращение).

31-Flavors/intro/create-table.sql

```
CREATE TABLE PersonalContacts (
  -- другие столбцы
  salutation VARCHAR(4)
    CHECK (salutation IN ('Mr.', 'Mrs.', 'Ms.', 'Dr.', 'Rev.')),
);
```

Это должно решить проблему — если вам действительно не придется поддерживать новые обращения.

К сожалению, руководитель сообщает, что компания открывает филиал во Франции и теперь необходимо поддерживать обращения `M.`, `Mme.` и `Mlle`. Вам поручают изменить таблицу контактов, чтобы она допускала эти значения. Это весьма непростое дело, которое и вовсе может оказаться невыполнимым без ограничения доступности таблицы.

А вы вспоминаете слова шефа о том, что компания собирается в следующем месяце открыть филиал в Бразилии.

Цель: ограничение столбца конкретными значениями

Ограничение значений столбца фиксированным набором чрезвычайно полезно. Уверенность в том, что столбец ни при каких условиях не будет содержать не-

допустимое значение, упростит использование столбца. Например, в таблице `Bugs` базы данных из нашего примера столбец `status` указывает, на какой стадии находится работа над ошибкой: «новая», «в процессе», «исправлена» и т. д. Важность каждого из этих значений зависит от того, как организовано управление ошибками в проекте, но здесь принципиально то, что в столбце могут храниться только значения из этого набора.

В идеале база данных должна отклонять недействительные значения:

31-Flavors/obj/insert-invalid.sql

```
INSERT INTO Bugs (status) VALUES ('NEW'); -- OK

INSERT INTO Bugs (status) VALUES ('BANANA'); -- Ошибка!
```

Антипаттерн: перечисление значений при определении столбца

Многие разработчики указывают допустимые значения данных при определении столбца. Определение столбца является частью *метаданных* — определения структуры самой таблицы.

Например, для столбца можно определить *проверочное ограничение*. Оно блокирует любые операции вставки или обновления, которые приводят к нарушению данного ограничения.

31-Flavors/anti/create-table-check.sql

```
CREATE TABLE Bugs (
  -- другие столбцы
  status VARCHAR(20) CHECK (status IN ('NEW', 'IN PROGRESS', 'FIXED'))
);
```

В MySQL поддерживается нестандартный тип данных `ENUM`, который ограничивает столбец заданным набором значений.

31-Flavors/anti/create-table-enum.sql

```
CREATE TABLE Bugs (
  -- другие столбцы
  status ENUM('NEW', 'IN PROGRESS', 'FIXED'),
);
```

В реализации для MySQL значения объявляются в виде строк, но во внутреннем представлении столбец хранится как порядковый номер строки в перечислении. Таким образом достигается компактность хранения, но при сортировке запроса по этому столбцу по умолчанию результат упорядочивается по порядковым

номерам, а не по алфавиту (для строковых значений). Это может оказаться неожиданным.

Другие возможные решения включают *домены* и *типы, определяемые пользователем* (UDT, User-Defined Types). С их помощью можно ограничить столбец конкретным набором значений и удобно применить тот же домен или тип данных к нескольким столбцам базы данных. К сожалению, эти возможности еще не имеют стандартизированной поддержкки в других разновидностях РСУБД.

Наконец, можно написать триггер, который содержит набор допустимых значений и выдает ошибку, если значение `status` не совпадает с одним из этих значений.

У всех перечисленных решений имеются общие недостатки. В следующих разделах описаны некоторые из них.

«31 вкус» мороженого Baskin-Robbins

В 1953 году знаменитый производитель мороженого предлагал по одному вкусу на каждый день месяца, а в его рекламе еще много лет использовался слоган «31 вкус».

Сегодня, когда прошло уже более 60 лет, Baskin-Robbins предлагает 21 классический вкус, 12 сезонных вкусов, 16 региональных вкусов, а также различные «вкусы месяца». И хотя все вкусы мороженого когда-то составляли неизменяемый набор, который определил бренд, с тех пор компания Baskin-Robbins расширила свое предложение и сделала его изменяемым.

То же самое может произойти с проектом, для которого вы разрабатываете базу данных. Более того, это наверняка произойдет.

А что там было в середине?

Допустим, вы разрабатываете пользовательский интерфейс, через который пользователь может редактировать сообщения об ошибках. Чтобы он мог выбрать одно из допустимых значений статуса, вы решаете создать раскрывающийся список с этими значениями. Для этого необходимо запросить у базы данных список значений, разрешенных для столбца `status`. Первое, что приходит в голову, — получить все актуальные значения простым запросом следующего вида:

31-Flavors/anti/distinct.sql

```
SELECT DISTINCT status FROM Bugs;
```

Но если в базе данных хранятся только новые ошибки, этот запрос вернет только NEW. Если использовать этот результат для заполнения элемента пользовательского интерфейса, представляющего статус ошибки, возникает порочный круг; ошибке можно будет назначить только один из статусов, которые используются в настоящее время.

Чтобы получить полный список допустимых статусов, необходимо запросить определение метаданных этого столбца. Многие базы данных SQL поддерживают *системные представления* для подобных запросов, но пользоваться ими достаточно неудобно. Например, если вы используете тип данных MySQL ENUM, то для получения системных представлений INFORMATION_SCHEMA можно применить следующий запрос:

31-Flavors/anti/information-schema.sql

```
SELECT column_type
FROM information_schema.columns
WHERE table_schema = 'bugtracker_schema'
  AND table_name = 'bugs'
  AND column_name = 'status';
```

Вам не удастся просто получить привычный результирующий набор элементов перечисления из INFORMATION_SCHEMA. Вместо этого вы получите строку, содержащую определение ограничения, или тип данных ENUM. Например, приведенный выше запрос в MySQL возвращает столбец типа LONGTEXT со значением ENUM('NEW', 'IN PROGRESS', 'FIXED'), включая круглые скобки, запятые и одинарные кавычки. Вам придется написать код приложения для парсинга этой строки и извлечь отдельные значения, заключенные в кавычки, прежде чем использовать их для заполнения элемента пользовательского интерфейса.

Сложность запросов, необходимых для проверочных ограничений, доменов или UDT, неуклонно возрастает. Многие разработчики предпочитают действовать более надежно и вручную ведут параллельный список значений в коде приложения. Тем самым они открывают дверь ошибкам, если данные приложения будут рассинхронизированы с метаданными базы данных.

Добавление нового вкуса

Большинство обычных изменений сводится к добавлению или удалению одного из разрешенных значений. Синтаксиса добавления и удаления значений из ENUM или проверочных ограничений не существует; можно только переопределить столбец новым набором значений. В следующем примере в MySQL ENUM добавляется новое значение status — DUPLICATE:

31-Flavors/anti/add-enum-value.sql

```
ALTER TABLE Bugs MODIFY COLUMN status
  ENUM('NEW', 'IN PROGRESS', 'FIXED', 'DUPLICATE');
```

Для этого необходимо знать, что изменяемое определение столбца допускает значения NEW, IN PROGRESS и FIXED. Это возвращает нас к сложности получения текущего набора значений, о которой говорилось выше.

Некоторые базы данных не позволяют изменить определение столбца, если в таблице есть данные. Возможно, вам придется стереть содержимое таблицы, переопределить таблицу, а затем импортировать сохраненные данные; в это время таблица будет недоступной. Эти операции проделывают так часто, что для них даже придумали специальное сокращение: ETL (от «Extract, Transform, Load», то есть «извлечение, преобразование, загрузка»). Другие базы данных поддерживают реструктуризацию заполненной таблицы командами ALTER TABLE, но вносить такие изменения может быть сложно и затратно.

На уровне политики изменение метаданных, то есть изменение определений таблиц и столбцов, не должно выполняться часто, при этом особое внимание необходимо уделять тестированию и контролю качества. Если вы вынуждены изменять метаданные, чтобы добавить или удалить значение из ENUM, тогда вам либо приходится пропускать соответствующее тестирование, либо в сжатые сроки выполнять большой объем работы по внесению изменений. В любом случае такие изменения повышают риски и дезорганизуют проект.

Привычные вкусы навсегда

Удаление неактуальных значений может затронуть исторические данные. Допустим, вы изменяете процесс контроля качества и разбиваете FIXED на две фазы: CODE COMPLETE и VERIFIED:

31-Flavors/anti/remove-enum-value.sql

```
ALTER TABLE Bugs MODIFY COLUMN status
  ENUM('NEW', 'IN PROGRESS', 'CODE COMPLETE', 'VERIFIED');
```

При удалении FIXED из перечисления необходимо решить, что делать с ошибками со статусом FIXED. Одно из возможных изменений — обновить все ошибки со статусом FIXED и заменить его на VERIFIED. Другой возможный вариант — записать на место несуществующих значений NULL или значение по умолчанию. К сожалению, ALTER TABLE не сможет угадать, какое из этих изменений вам требуется.

Возможно, вам придется хранить старые значения, на которые ссылаются старые строки. Только исходя из определений столбцов невозможно установить, какие значения устарели, чтобы исключить их из пользовательского интерфейса. Кто-то все еще сможет выбрать одно из этих значений.

Трудности с портированием

Проверочные ограничения, домены и UDT поддерживаются не во всех базах данных SQL. Тип данных ENUM принадлежит к числу специфических возмож-

ностей MySQL. Каждая база данных может устанавливать разные ограничения на длину списка, который может передаваться в определении столбца. Языки триггеров тоже могут изменяться. Все эти различия затрудняют выбор решения, если требуется поддерживать несколько разновидностей баз данных.

Как распознать антипаттерн

Проблемы с использованием `ENUM` или проверочных ограничений возникают в тех случаях, когда набор значений не фиксирован. Если вы подумываете использовать `ENUM`, сначала спросите себя, будет ли меняться набор значений (и вообще, возможно ли это). Если ответ — да, скорее всего, применять `ENUM` не стоит.

- «Нам придется перевести базу данных в офлайн, чтобы добавить новый вариант в одно из меню приложения. Если все пройдет хорошо, это займет не более 30 минут».

  Это признак того, что набор значений встроен в определение столбца. Такие изменения никогда не должны прерывать работу.
- «Столбец `status` может содержать одно из значений в списке. Список не должен меняться».

  «Не должен» — не значит «не может».
- «Список значений в коде приложения рассинхронизировался с бизнес-правилами в базе данных — опять».

  Такая ситуация возникает из-за того, что информация хранится в двух разных местах.

Допустимые применения антипаттерна

Как уже говорилось, `ENUM` будет создавать меньше проблем, если набор значений остается неизменным. Получить набор значений из метаданных все еще трудно, но можно поддерживать соответствующий список значений в коде приложения без риска рассинхронизации.

Решение с `ENUM`, скорее всего, будет успешным, если нет смысла менять набор разрешенных значений, например, если столбец представляет выбор «или/или» с двумя взаимоисключающими значениями: `LEFT`/`RIGHT`, `ACTIVE`/`IN-ACTIVE`, `ON`/`OFF`, `INTERNAL`/`EXTERNAL` и т. д.

У проверочных ограничений много других возможностей использования, не ограниченных простой реализацией `ENUM`-подобного механизма, например проверка того, что в заданном промежутке времени начальная точка меньше конечной.

Решение: определение значений в данных

Существует более эффективное решение для ограничения значений в столбце: *таблица сопоставления* со строками для каждого значения, разрешенного в столбце `Bugs.status`. После создания такой таблицы необходимо объявить для `Bugs.status` ограничение внешнего ключа со ссылкой на новую таблицу.

31-Flavors/soln/create-lookup-table.sql

```
CREATE TABLE BugStatus (
  status VARCHAR(20) PRIMARY KEY
);

INSERT INTO BugStatus (status) VALUES ('NEW'), ('IN PROGRESS'), ('FIXED');

CREATE TABLE Bugs (
  -- другие столбцы
  status VARCHAR(20),
  FOREIGN KEY (status) REFERENCES BugStatus(status)
    ON UPDATE CASCADE
);
```

При вставке или обновлении строки в таблице `Bugs` необходимо использовать значение `status`, существующее в таблице `BugStatus`. Тем самым обеспечивается использование только допустимых значений, как с `ENUM` или проверочными ограничениями, однако это решение в некоторых случаях обеспечивает большую гибкость.

Запрос набора значений

Набор допустимых значений теперь хранится в данных, а не в метаданных, как с типом данных `ENUM`. Значения данных можно запросить из таблицы сопоставления инструкцией `SELECT`, как из любой другой таблицы. Это значительно упрощает получение значений в виде набора данных для представления в пользовательском интерфейсе. Вы даже можете отсортировать набор значений, которые будут выведены пользователю.

31-Flavors/soln/query-canonical-values.sql

```
SELECT status FROM BugStatus ORDER BY status;
```

Обновление значений в таблице сопоставления

Если вы используете таблицу сопоставления, новое значение можно добавить простой инструкцией `INSERT`. Такие изменения могут вноситься без ограничения доступа к таблице. Вам не нужно переопределять столбцы, планировать время недоступности таблицы или выполнять операции ETL. Кроме того, для

добавления или удаления значений не нужно знать текущий набор значений в таблице.

31-Flavors/soln/insert-value.sql

```
INSERT INTO BugStatus (status) VALUES ('DUPLICATE');
```

Переименовывать значения тоже легко. Если вы объявили внешний ключ с условием `ON UPDATE CASCADE`, то обновление имени в таблице сопоставления `BugStatus` автоматически обновляет любые ссылки внешних ключей на него в других таблицах.

31-Flavors/soln/update-value.sql

```
UPDATE BugStatus SET status = 'INVALID' WHERE status = 'BOGUS';
```

Поддержка неактуальных значений

Нельзя удалить строку из таблицы сопоставления, если на нее ссылается строка из `Bugs`. Внешний ключ для столбца `status` обеспечивает ссылочную целостность, так что значение должно существовать в таблице сопоставления.

Однако в таблицу сопоставления можно добавить еще один столбец атрибута, чтобы помечать некоторые значения как неактуальные. Это позволит хранить исторические данные в столбце `Bugs.status`, но при этом отличать неактуальные значения от значений, которые могут выводиться в пользовательском интерфейсе.

31-Flavors/soln/inactive.sql

```
ALTER TABLE BugStatus ADD COLUMN active
  ENUM('INACTIVE', 'ACTIVE') NOT NULL DEFAULT 'ACTIVE';
```

Чтобы пометить значение неактуальным, используйте `UPDATE` вместо `DELETE`:

31-Flavors/soln/update-inactive.sql

```
UPDATE BugStatus SET active = 'INACTIVE' WHERE status = 'DUPLICATE';
```

При извлечении набора значений, доступных для выбора пользователем в пользовательском интерфейсе, ограничьте запрос значениями `status` с пометкой `ACTIVE`:

31-Flavors/soln/select-active.sql

```
SELECT status FROM BugStatus WHERE active = 'ACTIVE';
```

Такое решение более гибкое, чем `ENUM` или проверочное ограничение, потому что в них не поддерживаются дополнительные атрибуты уровня значений.

Простая портируемость

В отличие от типа данных `ENUM`, проверочных ограничений, доменов или UDT, решение с таблицей сопоставления полагается только на стандартную функциональность SQL — декларативную ссылочную целостность с использованием ограничений внешнего ключа. Такое решение улучшает портируемость. Кроме того, в таблице сопоставления можно хранить практически неограниченное количество значений, так как каждое значение хранится в отдельной строке.

Используйте метаданные при проверке по фиксированному набору значений. Используйте данные при проверке по динамическому набору значений.

Мини-антипаттерн: зарезервированные слова

«Откуда взялась эта ошибка? Я проверил синтаксис по справочной документации и уверен, что все правильно. Я использовал похожий запрос с другой таблицей, и никаких ошибок не было».

```
ERROR 1064: You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use
near 'order'.
```

(Ошибка в синтаксисе SQL; обратитесь к руководству своей версии сервера MySQL за описанием правильного синтаксиса, который должен использоваться с ‘order’.)

Запрос, который привел к появлению ошибки, выглядит так:

31-Flavors/mini/query-order.sql

```
SELECT * FROM Bugs WHERE order = 123;
```

Действительно, с этим синтаксисом все хорошо, если `order` является идентификатором столбца. Но это не так: `ORDER` имеет значение зарезервированного ключевого слова SQL, которое вводится условием `ORDER BY`. SQL по умолчанию нечувствителен к регистру символов, поэтому `order` и `ORDER` считаются одним ключевым словом.

В SQL ключевые слова должны рассматриваться отдельно от идентификаторов, потому что если слово может быть идентификатором или ключевым

словом, некоторые запросы могут стать неоднозначными, и это собьет с толку парсер.

Языки программирования обычно имеют зарезервированные ключевые слова. Например, в Java разработчик не может использовать `class`, `while`, или `try`, или еще несколько десятков других слов в качестве имен методов, переменных, констант или других идентификаторов. В других языках существуют собственные списки зарезервированных ключевых слов, есть они и в SQL.

В некоторых языках, таких как PHP или Perl, перед именами переменных ставится специальный символ (например, `$`), который отличает их от ключевых слов. Функции невозможно присвоить имя `while()`, но переменная вполне может называться `$while`.

В SQL специальные символы не используются, но можно заключить любое слово в ограничители, чтобы показать, что оно является идентификатором. Стандартными ограничителями идентификаторов в SQL служат двойные кавычки:

31-Flavors/mini/query-order-doublequotes.sql

```
SELECT * FROM Bugs WHERE "order" = 123;
```

В MySQL в качестве ограничителя идентификаторов по умолчанию используются обратные апострофы:

31-Flavors/mini/query-order-backticks.sql

```
SELECT * FROM Bugs WHERE `order` = 123;
```

В Microsoft SQL Server для той же цели по умолчанию используются квадратные скобки:

31-Flavors/mini/query-order-brackets.sql

```
SELECT * FROM Bugs WHERE [order] = 123;
```

SQLite для удобства разработчиков, привыкших иметь дело с другими видами баз данных SQL, распознает все три стиля идентификаторов ограничителей.

Идентификаторы в ограничителях также позволяют делать то, чего нельзя в большинстве других языков: записывать идентификаторы с использованием пробелов или знаков препинания.

31-Flavors/mini/query-special.sql

```
SELECT * FROM Bugs WHERE "the order" = 123;

SELECT * FROM Bugs WHERE "the-order" = 123;
```

Зарезервированные ключевые слова могут использоваться в качестве идентификаторов, но нужно помочь парсеру SQL и ясно обозначить, что вы имели в виду именно идентификатор. В предыдущих примерах `order` является ключевым словом. Возможно, это самое распространенное ключевое слово, которое случайно выбирают разработчики, не сознавая, что оно зарезервировано. Вот еще ключевые слова, которые вызывают путаницу у разработчиков: `by`, `default`, `from`, `rows`, `row_number`, `table`, `to` и `year_month`. Нетрудно представить, что кто-то будет использовать эти или другие зарезервированные ключевые слова в качестве имен таблиц или имен столбцов.

Наконец, будьте внимательны при обновлении ПО РСУБД. Так как новая версия часто включает новые возможности синтаксиса SQL, новые слова потребуется добавить в список зарезервированных ключевых слов. Может оказаться, что идентификатор, который вы использовали годами, необходимо заключить в ограничители, прежде чем вы сможете обновить базу данных.

Когда теория предстает перед вами как единственно возможная, воспринимайте это как знак того, что вы не поняли теорию или проблему, которую она должна решить.

➤ *Карл Поппер (Karl Popper)*

ГЛАВА 12

Фантомные файлы

На вашем сервере базы данных катастрофа. При перестановке стойка, набитая жесткими дисками, опрокинулась и упала. К счастью, никто не пострадал, но тяжелые жесткие диски разбились. Даже съемные полы треснули в месте падения. К счастью, IT-отдел был готов к такому повороту: там ежедневно создавали резервные копии всех важных систем, поэтому быстро развернули новый сервер и восстановили базу данных.

Дымовые тесты (или smoke-тесты) быстро обнаружили проблему: приложение связывает графические изображения со многими сущностями базы данных, но все изображения пропали! Вы немедленно обращаетесь к технику.

«Мы восстановили базу данных и проверили ее полноту на момент последнего резервного копирования, — ответил он. — Где хранились изображения?»

Вы помните, что изображения хранились вне базы данных, в файловой системе. В базе данных хранится путь к изображению, и приложение открывает каждый графический файл по этому пути. «Изображения хранились в файлах. Они находились в файловой системе /var, как и базы данных».

Техник качает головой. «Мы не делаем резервных копий файлов в файловой системе /var, если только вы не указываете нам конкретные файлы. Конечно, мы создаем резервные копии всех баз данных, но остальные файлы /var — это обычно журналы, данные кэша и другие временные файлы. По умолчанию мы не делаем их резервное копирование».

У вас темнеет в глазах. В базе данных каталога продуктов хранилось более 11 000 изображений. Скорее всего, многие из них есть и в других местах, но на то, чтобы их собрать, отформатировать и сделать миниатюры для веб-поиска, уйдут недели.

Цель: хранение графики или других больших данных

Графика и другие мультимедиа используются во многих приложениях. Иногда они связываются с сущностями, хранимыми в базе данных. Например, можно разрешить пользователю выбрать портрет или аватар, который будет отображаться при отправке комментария. В базе данных ошибок часто сохраняются скриншоты экрана с обстоятельствами возникновения ошибки.

В этой главе рассматривается механизм хранения изображений и связывания их с сущностями базы данных (например, учетными записями или ошибками). При запросе этих сущностей из базы данных нам понадобится функциональность чтения связанных изображений в приложении.

Антипаттерн: а что, если мне нужны файлы

Концептуально изображение является атрибутом таблицы. Например, в таблице `Accounts` может присутствовать столбец `portrait_image`.

Phantom-Files/anti/create-accounts.sql

```
CREATE TABLE Accounts (
  account_id     SERIAL PRIMARY KEY,
  account_name   VARCHAR(20),
  portrait_image BLOB
);
```

Точно так же можно хранить несколько изображений одного типа в зависимой таблице. Например, с ошибкой можно связать несколько поясняющих ее скриншотов.

Phantom-Files/anti/create-screenshots.sql

```
CREATE TABLE Screenshots (
  image_id          SERIAL NOT NULL,
  bug_id            BIGINT UNSIGNED NOT NULL,
  screenshot_image  BLOB,
  caption           VARCHAR(100),
  PRIMARY KEY (image_id, bug_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

Пока все выглядит достаточно просто, но с выбором типа данных для изображения полной ясности уже нет. Низкоуровневые двоичные данные для изображения можно хранить в типе данных `BLOB`, как показано выше. Однако многие хранят изображения в файле файловой системы и сохраняют в базе данных только пути к файлам в формате `VARCHAR`.

Phantom-Files/anti/create-screenshots-path.sql

```
CREATE TABLE Screenshots (
  image_id SERIAL NOT NULL,
  bug_id BIGINT UNSIGNED NOT NULL,
  screenshot_path VARCHAR(100),
  caption VARCHAR(100),
  PRIMARY KEY (image_id, bug_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

Среди разработчиков кипят ожесточенные споры на эту тему. В пользу обоих решений есть свои доводы, но некоторые разработчики считают, что изображения всегда лучше хранить вне базы данных. Тем не менее такой способ хранения создает ряд рисков, описанных в следующих разделах.

К файлам, хранящимся вне БД, нельзя применить DELETE

Первая проблема — сборка мусора. Если изображения хранятся вне базы данных и строка пути к ним удаляется, находящийся по этому пути файл не удалится автоматически.

Phantom-Files/anti/delete.sql

```
DELETE FROM Screenshots WHERE bug_id = 1234 and image_id = 1;
```

Если в приложении отсутствует возможность удаления осиротевших файлов при удалении строк базы данных, которые на них ссылаются, такие файлы начнут накапливаться.

В случае с файлами, хранящимися вне БД, невозможно изолировать транзакции

Обычно обновление или удаление данных не видно другим клиентам, пока транзакция не подтверждена инструкцией `COMMIT`.

Однако с изменениями в файлах, расположенных вне базы данных, все иначе. Если вы удаляете файл, он немедленно становится недоступным для других клиентов. И если изменить содержимое файла, то другие клиенты увидят эти изменения сразу же — вместо предыдущего содержимого файла, отображаемого, пока транзакция не подтверждена.

Phantom-Files/anti/transaction.py

```
import mysql.connector
import os

cnx = mysql.connector.connect(user='scott', database='test')

cursor = cnx.cursor()
```

```
query = "DELETE FROM Screenshots WHERE bug_id = %s AND image_id = %s"
cursor.execute(query, (1234, 1,))

os.unlink('screenshot1234-1.jpg');

# В это время другие клиенты все еще видят строку базы данных,
# а не файл с изображением.

cnx.commit()
```

На практике подобное встречается нечасто. Кроме того, особого значения оно не имеет; отсутствие изображения в веб-приложении — явление вполне привычное. Однако в других ситуациях последствия могут быть достаточно серьезными.

К файлам, хранящимся вне БД, нельзя применить ROLLBACK

Откат транзакций в случае ошибок (или даже если логика приложения требует отмены внесенных изменений) — это нормально.

Допустим, вы удаляете файл скриншота командой `DELETE` вместе с соответствующей строкой в базе данных. Если выполнить откат этого изменения, то удаление строки из базы данных будет отменено, но файл все равно удалится.

Phantom-Files/anti/rollback.py

```
import mysql.connector
import os

cnx = mysql.connector.connect(user='scott', database='test')

cursor = cnx.cursor()

query = "DELETE FROM Screenshots WHERE bug_id = %s AND image_id = %s"
cursor.execute(query, (1234, 1,))

os.unlink('screenshot1234-1.jpg');

cnx.rollback()
```

Строка базы данных восстановлена — но не файл с изображением.

К файлам, хранящимся вне БД, нельзя применить средства резервного копирования баз данных

Многие РСУБД предоставляют клиентские средства, упрощающие резервное копирование используемых баз данных. Например, в MySQL это команда `mysqldump`, в Oracle — `rman`, в PostgreSQL — `pg_dump`, в SQLite — `.dump` и т. д.

Средства резервного копирования важны, потому что если несколько клиентов вносят изменения одновременно, резервная копия может содержать частичные изменения, потенциально нарушающие ссылочную целостность и даже повреждающие резервную копию, так что ее нельзя будет восстановить.

Средства резервного копирования не умеют копировать файлы, на которые ссылается путь в столбце VARCHAR таблицы. Таким образом, создавать резервную копию базы данных придется в два этапа: сначала использовать программу резервного копирования баз данных, а затем — программу резервного копирования файловой системы для внешних файлов.

Даже если вы включаете внешние файлы в резервное копирование, трудно гарантировать, что копии этих файлов будут синхронизированы с транзакцией, которая использовалась для резервного копирования базы данных. Приложения могут добавлять и изменять файлы изображений в любой момент — возможно, сразу же после того, как вы запустите резервное копирование базы данных.

На файлы, хранящиеся вне БД, не действуют привилегии доступа SQL

Внешние файлы обходят любые привилегии, которые назначаются инструкциями SQL GRANT и REVOKE. Привилегии SQL управляют доступом к таблицам и столбцам, но они не распространяются на внешние файлы, имена которых указаны в строковом виде в базе данных.

Файлы, хранящиеся вне БД, не являются типами данных SQL

Путь, хранящийся в screenshot_path, представляет собой обычную строку. База данных не проверяет правильность пути и не может гарантировать, что файл по нему существует. Если файл будет переименован, перемещен или удален, то база данных не обновит строку автоматически. Любая логика, которая работает с этой строкой как с путем, зависит от кода приложения.

Phantom-Files/anti/file-get.py

```
import mysql.connector
import os

cnx = mysql.connector.connect(user='scott', database='test')

cursor = cnx.cursor()

query = "SELECT image_path FROM Screenshots WHERE bug_id = %s AND image_id = %s"
cursor.execute(query, (1234, 1,))
row = cursor.fetchone()
cursor.close()
```

```
if row:
    image_path = row[0]
    with open(image_path) as image_file:
        image_content = image_file.read()
```

Одно из преимуществ базы данных заключается в том, что она помогает сохранить целостность данных. Когда вы сохраняете данные во внешних файлах, вы лишаетесь этого преимущества, и вам придется писать код приложения для выполнения проверок, обеспечиваемых базой данных.

Как распознать антипаттерн

Признаки антипаттерна обнаружить непросто. Если проект содержит документацию для администратора ПО или вы можете поговорить с его разработчиками (даже если это вы сами), узнайте, как система выполняет следующие операции:

- Резервное копирование и восстановление данных, включая графику и вложения.
- Проверка данных после восстановления их на другом сервере (не та том, на котором создавалась резервная копия).
- Удаление изображений, на которые не указывает ни одна ссылка в строках базы данных.
- Предоставление пользователям приложения доступа к изображениям, а также обратная задача: блокировка просмотра изображений пользователями, не обладающими привилегиями для их просмотра.
- Редактирование или замена изображений, включая отмену таких изменений; восстановление исходного изображения после отмены.

Разработчики проектов, в которых задействован этот антипаттерн, не продумали некоторые (или все) из приведенных вопросов. Не каждому приложению требуется ошибкоустойчивое управление транзакциями или управление доступом к файлам изображений в SQL. Возможно, вы обнаружите, что перевод базы данных в офлайн во время резервного копирования — оправданный компромисс. Если ответы неясны или их трудно получить, это может указывать на то, что разработчики небрежно отнеслись к проектированию работы с внешними файлами.

Допустимые применения антипаттерна

Есть много убедительных причин хранить изображения и другие большие объекты в файлах вне базы данных:

- Без изображений база данных становится намного компактнее, потому что изображения занимают много места по сравнению с простыми типами данных (такими, как целые числа и строки).
- Без изображений резервное копирование базы данных выполняется быстрее, а результат получается менее объемным. Изображения приходится копировать из файловой системы в отдельной фазе резервного копирования, но это может быть удобнее, чем резервное копирование гигантской базы данных.
- Хранение изображений в файлах, внешних по отношению к базе данных, упрощает предварительный просмотр или редактирование, если вдруг возникнет такая необходимость, особенно если потребуется применить пакетную правку ко всем изображениям.

Если преимущества хранения изображений в файлах для вас важны, а описанные выше проблемы не критичны, этот механизм хранения подойдет для вашего проекта.

Некоторые базы данных поддерживают специальные типы данных SQL, которые ссылаются на внешние файлы более или менее прозрачным образом. В Oracle этот тип данных называется `BFILE`, а в SQL Server 2008 — `FILESTREAM`.

Не исключайте ни одно решение

Однажды я спроектировал приложение, которое хранит изображения вне базы данных. Мой заказчик заключил контракт на разработку регистрационного приложения для технической конференции. Когда участник приезжал на конференцию, приложение делало его фото на камеру, регистрировало в системе и распечатывало идентификационную карточку.

Это мое приложение было довольно простым. Каждое изображение могло вставляться и обновляться только для одной клиентской заявки (если участник моргнул или ему не нравилось фото, его можно было заменить в ходе регистрации). Не было никаких требований к сложной обработке транзакций, параллельному доступу со стороны нескольких клиентов или возможности отката. Привилегии доступа SQL не использовались. Так как изображения не нужно было загружать из базы данных, это упрощало их предварительный просмотр.

Я работал над этим проектом в то время, когда практические возможности для приложений и баз данных были намного ниже современных. С учетом этих ограничений имело смысл хранить изображения в иерархии каталогов и управлять ими из кода приложения.

Чтобы понять, в какой степени проблемы, описанные в разделе «Антипаттерн» данной главы, повлияют на вас, продумайте, как будет организована работа с изображениями в вашем приложении. Не слушайте разработчиков, которые считают, что всегда лучше хранить изображения во внешних файлах, и примите обдуманное решение.

Решение: использование типа данных BLOB при необходимости

Если вы встретились с проблемами, описанными в разделе «Антипаттерн» этой главы, попробуйте хранить изображения в базе данных (а не во внешних файлах). Все базы данных поддерживают тип данных `BLOB`, который можно использовать для хранения любых двоичных данных.

Phantom-Files/soln/create-screenshots.sql

```
CREATE TABLE Screenshots (
  bug_id            BIGINT UNSIGNED NOT NULL,
  image_id          BIGINT UNSIGNED NOT NULL,
  screenshot_image  BLOB,
  caption           VARCHAR(100),
  PRIMARY KEY (bug_id, image_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

Хранение изображений в столбце `BLOB` решает все проблемы:

- Изображения хранятся в базе данных. Дополнительный шаг с загрузкой становится ненужным. Исчезает риск, что имя файла окажется недопустимым.
- При удалении строки изображение удаляется автоматически.
- Другие клиенты не видят изменений в изображении до их подтверждения.
- Откат транзакции восстанавливает предыдущее состояние изображения.
- Обновление строки блокирует изображение, так что другие клиенты не смогут параллельно его обновлять.
- Резервные копии базы данных включают все изображения.
- Привилегии SQL управляют доступом как к изображению, так и к строке.

Максимальный размер `BLOB` зависит от конкретной РСУБД, но обычно его хватает для большинства изображений. Все базы данных должны поддерживать тип `BLOB` или похожий формат. Например, MySQL предоставляет типы данных `MEDIUMBLOB` и `LONGBLOB`, вмещающие до 16 Мбайт или 4 Гбайт соответственно. Oracle поддерживает типы данных `LONG RAW` или `BLOB` с емкостью до 2 и 4 Гбайт соответственно. Похожие типы данных доступны и в других продуктах.

Изображения обычно изначально хранятся в файлах, так что вам понадобится загрузить их в столбец `BLOB` базы данных. Некоторые базы данных предоставляют специальные функции для загрузки внешних файлов. Например,

в MySQL поддерживается функция `LOAD_FILE()`, которая может использоваться для чтения файла — обычно для сохранения содержимого в столбце `BLOB`.

Phantom-Files/soln/load-file.sql

```
UPDATE Screenshots
SET screenshot_image = LOAD_FILE('images/screenshot1234-1.jpg')
WHERE bug_id = 1234 AND image_id = 1;
```

Также можно сохранить содержимое столбца `BLOB` в файле. Например, в MySQL инструкция `SELECT` поддерживает необязательную секцию для хранения результата запроса «as is», без форматирования для обозначения столбцов или завершения строк.

Phantom-Files/soln/dumpfile.sql

```
SELECT screenshot_image
INTO DUMPFILE 'images/screenshot1234-1.jpg'
FROM Screenshots
WHERE bug_id = 1234 AND image_id =1;
```

Также можно извлекать данные изображения из `BLOB` и выводить их напрямую. Ответом веб-запроса может быть двоичный контент (например, изображение), хотя это требует правильного задания типа контента. Ниже приведен пример веб-приложения Python Flask, которое возвращает скриншот.

Phantom-Files/soln/binary-content.py

```
import mysql.connector
from flask import Flask, Response

app = Flask(__name__)

@app.route('/')
def screenshot():
    cnx = mysql.connector.connect(user='scott', database='test')

    cursor = cnx.cursor()
    query = """
      SELECT screenshot_image FROM Screenshots
      WHERE bug_id = %s AND image_id = %s"""
    cursor.execute(query, (1234, 1,))
    row = cursor.fetchone()
    cursor.close()

    if row:
      screenshot_image = row[0]
      return Response(screenshot_image, mimetype='image/jpeg')
```

```
    else:
      return Response(status=404)

if __name__ == '__main__':
    app.run()
```

Прежде чем организовывать хранение изображений или другого контента в файлах за пределами БД, как это принято, рассмотрите возможность использования столбца `BLOB` для хранения этих данных. Возможно, это упростит ваш код и сделает его более устойчивым к ошибкам.

Нарушая связи между данными и базой данных, вы принимаете на себя бремя управления этими данными.

Каждый раз, когда вы прибегаете к ее помощи
для получения результата, возникает вопрос:
при каком пути вычислений этот результат может
быть получен машиной за кратчайшее время?

➤ *Чарльз Бэббидж (Charles Babbage),*
«Отрывки из жизни философа» (1864)

ГЛАВА 13

Индексный дробовик

«Привет! У тебя есть минутка? Мне нужна твоя помощь». Оклахомский акцент в трубке заглушает гудение вентиляции в вычислительном центре. Это ведущий администратор баз данных компании.

«Ну конечно», — отвечаете вы, не совсем понимая, что ему нужно.

«Тут у нас появилась база данных, которая почти парализовала сервер, — продолжает админ. — Я нашел проблему. У одних таблиц индексов нет, а у других лишние. Нам нужно с этим разобраться, или занимайся сам, потому что ни у кого нет времени!»

«Мне очень жаль — вообще-то я не так много знаю о базах данных, — отвечаете вы, пытаясь успокоить администратора. — Мы старались с оптимизацией как могли, но очевидно, такой эксперт, как ты, и сам бы все сделал. Нельзя ли применить какую-нибудь настройку?»

«Сынок, я настроил все, что мог, иначе здесь бы вообще ничего не работало, — отвечает админ. — Последнее, что осталось, — урезать ресурсы твоего приложения, но тебе это вряд ли понравится. Пора четко понять, что конкретно должна делать база данных в приложении».

Кажется, тучи сгущаются. Вы с опаской спрашиваете: «А что ты предлагаешь? Говорю же, в нашей команде нет знатоков баз данных».

«Не проблема, — смеется администратор. — Ты же знаешь свое приложение, не так ли? Именно это важно — и с этим я помочь *не могу*. Я пришлю своего человека, он передаст тебе нужные инструменты, а потом мы исправим найденное узкое место. Немного подсказок, и ты справишься. Вот увидишь».

Цель: оптимизация производительности

Проблемы с производительностью, пожалуй, беспокоят разработчиков баз данных чаще всего. Только послушайте, о чем идет речь на любой технической конференции: там постоянно обсуждают инструменты и приемы, которые позволяют выжать чуть больше сил из базы данных. Когда я выступаю с докладом о способах структурирования баз данных или о том, как повысить надежность, безопасность или правильность запросов SQL, первый (а порой и единственный) вопрос от аудитории звучит так: «Хорошо, но как это повлияет на производительность?»

Самый эффективный метод повышения производительности базы данных — грамотное использование индексов. *Индекс* представляет собой структуру данных, которая используется для связывания значений со строками, в которых эти значения встречаются в заданном столбце. С помощью индекса значение можно находить быстрее, чем методом брутфорса, когда просматривается вся таблица сверху донизу.

Разработчики обычно не понимают, как и когда использовать индекс. В документации и книгах о базах данных крайне редко можно найти понятное руководство по использованию индексов. Разработчики могут только предполагать их эффективность.

Антипаттерн: беспорядочное использование индексов

Выбор индексов наугад неизбежно чреват ошибками. Ошибки, возникающие из-за непонимания того, когда использовать индексы, бывают трех видов:

- индексов нет (или их недостаточно);
- индексов слишком много (или они бесполезны);
- индексы не могут повысить эффективность отправляемых запросов.

Отсутствие индексов

Выше я говорил о том, что обновление индекса базы данных требует лишних расходов ресурсов. Каждый раз, когда вы используете `INSERT`, `UPDATE` или `DELETE`, базе данных приходится обновлять структуры данных индексов для таблицы, чтобы они были актуальны, а последующие поиски используют эти индексы, чтобы находить нужные наборы записей.

Вы уже знаете, что лишние расходы снижают эффективность. Поэтому совершенно не нужно, чтобы обновление индекса создавало такие расходы,

и некоторые разработчики решают вообще отказаться от индексов, считая это выходом из ситуации. Подобный совет встречается довольно часто, но он игнорирует тот факт, что у индексов есть преимущества, компенсирующие накладные расходы.

Не все расходы напрасны. Компания нанимает административный персонал, юристов и бухгалтеров, платит за аренду, хотя все это напрямую не генерирует доход. Бизнес идет на это, потому что эти люди вносят важный вклад в успех компании.

В норме на каждое обновление таблицы приходятся сотни запросов. Каждый раз, когда вы выполняете запрос, использующий индекс, вы компенсируете расходы на сопровождение индекса.

Индекс также может помочь с выполнением инструкций `UPDATE` или `DELETE` за счет быстрого нахождения записи. В частности, индекс для первичного ключа `bug_id` повышает эффективность следующей инструкции:

Index-Shotgun/anti/update.sql

```
UPDATE Bugs SET status = 'FIXED' WHERE bug_id = 1234;
```

Чтобы найти данные в неиндексированном столбце, приходится проводить полное сканирование таблицы:

Index-Shotgun/anti/update-unindexed.sql

```
UPDATE Bugs SET status = 'OBSOLETE' WHERE date_reported < '2000-01-01';
```

Индексы не стандартны

А вы знаете, что в стандарте ANSI/ISO SQL ничего не сказано об индексах? Реализация и оптимизация хранимых данных не определяются в языке SQL, так что каждая РСУБД имеет полное право реализовывать индексы по своему усмотрению.

В большинстве РСУБД используется сходный синтаксис `CREATE INDEX`, но каждый продукт совершенствуется и развивается по-своему. Не существует стандарта для функциональности индексов. Точно так же нет стандартов для сопровождения индексов, автоматической оптимизации запросов, планирования запросов или таких команд, как `EXPLAIN`.

Чтобы извлечь максимальную пользу из индексов, вам придется изучить документацию своей РСУБД. Конкретный синтаксис и функциональность индексов серьезно различаются, но логика остается неизменной.

Слишком много индексов

Индексы эффективны только при выполнении запросов, использующих этот индекс. Нет никакого смысла создавать индексы, которые не используются. Рассмотрим несколько примеров:

Index-Shotgun/anti/create-table.sql

```
CREATE TABLE Bugs (
    bug_id SERIAL PRIMARY KEY,
    date_reported DATE NOT NULL,
    summary VARCHAR(80) NOT NULL,
    status VARCHAR(10) NOT NULL,
    hours NUMERIC(9,2),
❶  INDEX (bug_id),
❷  INDEX (summary),
❸  INDEX (hours),
❹  INDEX (bug_id, date_reported, status)
 );
```

В этом примере есть несколько бесполезных индексов:

❶ `bug_id`: многие базы данных автоматически создают индекс для первичного ключа, так что определение другого индекса избыточно. Он не приносит никакой пользы, кроме лишних накладных расходов. В каждой РСУБД собственные правила автоматического создания индексов. Ознакомьтесь с документацией базы данных, которую вы используете.

❷ `summary`: индекс для длинного строкового типа данных (такого, как `VARCHAR(80)`) занимает больше места, чем индекс для более компактного типа данных. Кроме того, вам вряд ли придется выполнять запросы на выборку или сортировку по всему столбцу `summary`.

❸ `hours`: еще один пример столбца, по которому вам вряд ли понадобится искать конкретные значения.

❹ `bug_id`, `date_reported`, `status`: в пользу составных индексов говорит многое, однако часто разработчики создают составные индексы, которые оказываются избыточными или редко используются. Кроме того, в составном индексе важен порядок столбцов; в критериях поиска, соединения или сортировки столбцы должны перечисляться слева направо.

Когда индекс бесполезен

Следующая категория ошибок — выполнение запросов, которые не могут использовать индексы. Разработчики создают все больше индексов, пытаясь найти волшебное сочетание столбцов или параметров индекса, чтобы их запросы выполнялись быстрее.

Индекс базы данных можно сравнить с телефонной книгой. В ней легко, например, найти всех абонентов с фамилией (last name) `Charles`. Все люди с одинаковыми фамилиями сгруппированы вместе, потому что записи в телефонной книге упорядочены пофамильно.

Index-Shotgun/anti/indexable.sql

```
CREATE INDEX LastNameFirstName ON TelephoneBook(last_name, first_name);

SELECT * FROM TelephoneBook WHERE last_name = 'Charles'; -- OK
```

Но если вам потребуется найти в телефонной книге всех абонентов с *именем* (first name) `Charles`, упорядочение в телефонной книге не принесет никакой пользы. Это имя может быть у любого абонента, так что вам придется просмотреть всю книгу строка за строкой.

Index-Shotgun/anti/not-indexable.sql

```
SELECT * FROM TelephoneBook WHERE first_name = 'Charles'; -- НЕ индексируется
```

Телефонная книга упорядочивается сначала по фамилии, а затем по имени — по аналогии с составным индексом базы данных по столбцам `last_name`, `first_name`. Этот индекс не ускорит поиск по имени.

Ниже приведен еще один пример индекса и запроса, который не может воспользоваться этим индексом:

Index-Shotgun/anti/not-indexable.sql

```
CREATE INDEX LastNameFirstName ON Accounts(last_name, first_name);

SELECT * FROM Accounts ORDER BY first_name, last_name;
```

Запрос демонстрирует сценарий с телефонной книгой. Составной индекс для столбца `last_name` и следующего за ним `first_name` (как в телефонной книге) не поможет с первичной сортировкой по `first_name`.

Index-Shotgun/anti/not-indexable.sql

```
CREATE INDEX DateReported ON Bugs(date_reported);

SELECT * FROM Bugs WHERE MONTH(date_reported) = 4;
```

Даже если вы создадите индекс для столбца `date_reported`, порядок столбцов в индексе не поможет с поиском по месяцу. Индекс упорядочен по всем компонентам даты, начиная с года. В каждом году есть четвертый месяц, так что строки, у которых месяц равен 4, распределены по всему индексу.

Некоторые базы данных поддерживают индексы по выражениям, или индексы по генерируемым столбцам, а также индексы по простым столбцам. Прежде чем

использовать индекс, его необходимо определить, и этот индекс ускорит операции только с выражением, заданным в его определении.

Index-Shotgun/anti/not-indexable.sql

```
CREATE INDEX LastNameFirstName ON Accounts(last_name, first_name);

SELECT * FROM Accounts WHERE last_name = 'Charles' OR first_name = 'Charles';
```

Мы возвращаемся к той же проблеме: строки с интересующим нас именем хаотично распределены в порядке определенного индекса. Результат предыдущего запроса совпадает с результатом следующего:

Index-Shotgun/anti/not-indexable.sql

```
SELECT * FROM Accounts WHERE last_name = 'Charles'
UNION
SELECT * FROM Accounts WHERE first_name = 'Charles';
```

Индекс из примера ускорит поиск фамилии, но не имени.

Index-Shotgun/anti/not-indexable.sql

```
CREATE INDEX Description ON Bugs(description);

SELECT * FROM Bugs WHERE description LIKE '%crash%';
```

Так как шаблон в предикате поиска может находиться в любом месте строки, структура данных отсортированного индекса здесь ничем не поможет.

Индексы с низкой избирательностью

Избирательность (selectivity) — статистическая характеристика индекса базы данных. Она определяется как отношение количества разных значений в индексе к общему количеству строк в таблице:

```
SELECT COUNT(DISTINCT status) / COUNT(status) AS selectivity FROM Bugs;
```

Чем ниже избирательность, тем ниже эффективность индекса. Почему? Рассмотрим аналогию.

В книгах используется алфавитный указатель: для каждого элемента алфавитного указателя книги указываются страницы, на которых встречается данное слово. Если слово встречается часто, номеров страниц может быть достаточно много. Чтобы найти нужную часть книги, придется обратиться к каждой странице в списке, последовательно перебирая их.

В алфавитных указателях не приводятся слова, которые встречаются на слишком многих страницах. Если вам приходится часто переходить от индекса к страницам книги, с таким же успехом можно просто прочитать всю книгу от начала до конца.

Аналогичным образом в индексах баз данных, если слово встречается в слишком большом количестве строк в таблице, читать индекс будет сложнее, чем просто сканировать таблицу. Более того, подобное использование индекса может требовать высоких накладных расходов.

В идеале база данных должна отслеживать избирательность индексов и не использовать бесполезные индексы.

Как распознать антипаттерн

Некоторые признаки антипаттерна «Индексный дробовик»:

- «Вот запрос, как его ускорить?»

  Вероятно, это самый популярный вопрос о SQL, но в нем нет никаких деталей: описания таблиц, индексов, объема данных, метрик производительности и оптимизации. Без них любой ответ будет обычным гаданием.

- «Я определил индекс для каждого поля; почему не стало быстрее?»

  Это классический пример индексного дробовика. Вы испробовали все возможные индексы — но на самом деле это стрельба в молоко.

- «Я читал, что индексы замедляют работу базы данных, поэтому я ими не пользуюсь».

  Как многие разработчики, вы ищете одну стратегию повышения производительности на все случаи жизни. Но ее не существует.

Допустимые применения антипаттерна

Если вы проектируете базу данных общего пользования, не зная, какие запросы важно оптимизировать, нельзя с уверенностью сказать, какие индексы будут наиболее эффективными. Приходится делать обоснованные предположения. Скорее всего, вы упустите какие-то полезные индексы. И, вероятно, создадите лишние. Ваше предположение должно оказаться как можно ближе к истине.

База данных не всегда СЛУЖИТ «узким местом»

Разработчики часто думают, что база данных — всегда самая медленная часть приложения и именно в ней кроется источник проблем с производительностью. Однако это не так.

Например, когда я работал над одним приложением, руководитель попросил меня выяснить, почему оно такое медленное. Он настаивал, что в этом

виновата база данных. После профилирования кода приложения выяснилось, что 80 % времени занимал парсинг вывода HTML для нахождения полей формы, чтобы заполнить форму значениями. Проблемы с производительностью не имели никакого отношения к запросам базы данных.

Прежде чем делать выводы о причинах проблем с производительностью, используйте инструменты измерения, чтобы определить, какой именно код выполняется слишком долго. Иначе вы рискуете нарваться на преждевременную оптимизацию.

Решение: MENTOR

Суть антипаттерна «Индексный дробовик» — в необоснованном создании или удалении индексов, поэтому вам понадобится провести анализ баз данных и найти причины для включения индексов или отказа от них.

Контрольный список анализа базы данных для принятия обоснованных решений, касающихся индексов, часто описывается сокращением MENTOR («Measure, Explain, Nominate, Test, Optimize, Rebuild — измерение, объяснение, поиск кандидатов, тестирование, оптимизация, перестройка»).

Измерение

Невозможно принять обоснованное решение, не имея информации. Многие базы данных предоставляют средства регистрации времени выполнения запросов SQL, чтобы выявлять самые затратные операции. Пример:

- Microsoft SQL Server и Oracle содержат средства и инструменты трассировки SQL для получения информации и анализа результатов трассировки. Microsoft называет этот инструментарий *SQL Server Profiler*, а Oracle использует название *TKProf*.
- MySQL и PostgreSQL могут регистрировать запросы, время выполнения которых превышает заданный порог. MySQL называет этот инструмент *журналом медленных запросов*, а его параметр конфигурации `long_query_time` по умолчанию равен 10 секундам. В PostgreSQL используется сходная переменная конфигурации `log_min_duration_statement`.

Зная, на какие запросы приходится большая часть времени работы приложения, вы поймете, что́ стоит оптимизировать в первую очередь, чтобы получить максимальный эффект. Может оказаться, что все запросы эффективны, кроме одного — критического. Именно с этого запроса следует начинать оптимизацию.

Самая затратная область приложения не всегда занимает больше всего времени, если этот запрос выполняется достаточно редко. Другие простые запросы могут выполняться часто — намного чаще, чем вы ожидаете, поэтому в итоге на них тратится больше времени. Оптимизация таких запросов оправдает затраченные усилия.

На время измерения производительности запросов отключите любое кэширование результатов. Эта разновидность кэширования разработана так, чтобы обходить выполнение запросов и использование индексов, поэтому полученные данные будут неточными.

Чтобы получить более точную информацию, профилируйте приложение после его развертывания. Соберите сводные данные о том, на что уходит основное время, когда с приложением работают реальные пользователи и используется реальная база данных. Время от времени проверяйте данные профилирования, чтобы убедиться, что в приложении не возникло новое узкое место.

Не забывайте отключать или снижать частоту получения данных профилировщиков после завершения измерений, потому что эти инструменты тоже требуют накладных расходов.

Объяснение

Следующий шаг после выявления самого затратного запроса — понять, почему он так медленно выполняется. Каждая база данных применяет оптимизатор при выборе индексов для запроса. У базы данных можно запросить отчет о проведенном анализе, который называется *планом исполнения запроса*, или *QEP* (Query Execution Plan).

Синтаксис запроса QEP в разных РСУБД разный, как показано в таблице ниже.

Не существует стандарта, определяющего, какую информацию должен включать отчет QEP или формат этого отчета. В общем случае QEP показывает, какие таблицы используются в запросе, как оптимизатор решает применять индексы и в каком порядке он будет обращаться к таблицам. Отчет также может включать статистику, например количество строк, генерируемых каждой стадией запроса.

РСУБД	**Средство вывода отчетов QEP**
IBM DB2	EXPLAIN, команда db2expln или Visual Explain
Microsoft SQL Server	SET SHOWPLAN_XML или Display Execution Plan
MySQL	EXPLAIN
Oracle	EXPLAIN PLAN
PostgreSQL	EXPLAIN
SQLite	EXPLAIN

Рассмотрим пример запроса SQL и соответствующий отчет QEP:

Index-Shotgun/soln/explain.sql

```
EXPLAIN SELECT Bugs.*
FROM Bugs
JOIN (BugsProducts JOIN Products USING (product_id))
  USING (bug_id)
WHERE summary LIKE '%crash%'
  AND product_name = 'Open RoundFile'
ORDER BY date_reported DESC;
```

Отчет MySQL QEP:

table	type	possible_keys	key	key_len	ref	rows	filtered	extra
Bugs	ALL	PRIMARY, bug_id	NULL	NULL	NULL	4650	100	Using where; Using temporary; Using filesort
BugsProducts	ref	PRIMARY, product_id	PRIMARY	8	Bugs.bug_id	1	100	Using index
Products	ALL	PRIMARY, product_id	NULL	NULL	NULL	3	100	Using where; Using join buffer

Столбец `key` показывает, что запрос использует только индекс первичного ключа `BugsProducts`. Кроме того, из заметок в последнем столбце видно, что запрос сортирует результат во временную таблицу без использования индекса.

Выражение `LIKE` инициирует полное сканирование таблицы в `Bugs`, а индекса для `Products.product_name` не существует. Запрос можно улучшить созданием нового индекса для `product_name`, а также использованием полнотекстового поиска (см. главу 17 «Поисковая система для бедных»).

Состав информации в отчете QEP зависит от конкретного продукта. В приведенном примере следует изучить страницу руководства MySQL Optimizing Queries with EXPLAIN, чтобы понять, как интерпретировать отчет[1].

Поиск кандидатов

После получения от оптимизатора данных QEP для запроса можно переходить к поиску случаев, в которых запрос обращается к таблице без использования индекса.

---

[1] https://dev.mysql.com/doc/refman/en/using-explain.html

Некоторые базы данных предоставляют средства для решения этой задачи, сбора статистики трассировки запросов и предложения изменений, включая создание новых индексов, которые вы упустили, но которые могут оказаться полезными для запроса.

Примеры:

- IBM DB2 Design Advisor;
- Microsoft SQL Server Database Engine Tuning Advisor;
- MySQL Enterprise Query Analyzer;
- Oracle Automatic SQL Tuning Advisor.

Даже без автоматических рекомендаций вы можете научиться распознавать ситуации, в которых индекс может быть полезен. Чтобы интерпретировать отчет QEP, необходимо изучить документацию базы данных.

Покрывающие индексы

Если индекс предоставляет все необходимые столбцы, то запросу вообще не приходится читать данные из таблицы.

Представьте, что записи телефонной книги содержат только номер страницы; когда вы находите имя, вам приходится открывать указанную страницу, чтобы узнать номер телефона. Логичнее находить нужную информацию за один шаг. Поиск имени выполняется быстро, потому что книга упорядочена, и было бы лучше получать другие необходимые атрибуты из этой записи, например телефон и, возможно, адрес.

Так работает *покрывающий индекс*. Можно определить индекс с дополнительными столбцами, хотя они и не являются необходимыми для поиска или сортировки.

```
CREATE INDEX BugCovering ON Bugs
  (status, bug_id, date_reported, reported_by, summary);
```

Если запрос ссылается только на столбцы, включенные в структуру данных индекса, то для генерирования результатов базе данных достаточно будет прочитать индекс.

```
SELECT status, bug_id, date_reported, summary
FROM Bugs WHERE status = 'OPEN';
```

Покрывающие индексы не могут использоваться для каждого запроса, но там, где это возможно, они обычно обеспечивают значительный выигрыш в производительности.

Тестирование

Этот шаг очень важен; после создания индексов снова профилируйте свои запросы. Важно убедиться, что изменения действительно на что-то повлияли и работу можно считать выполненной.

На этом шаге также можно произвести впечатление на руководство и аргументировать проведенную оптимизацию. Никто не хочет, чтобы его недельный отчет звучал так: «Я испробовал все, что только можно, чтобы решить проблему с производительностью. Надо еще немного подождать, и ...» И все хотят произнести что-то подобное: «Я выяснил, что можно определить новый индекс для таблицы с высокой нагрузкой, и производительность критических запросов повысилась на 38 %».

Оптимизация

Индексы — компактные, часто используемые структуры данных, благодаря чему они отлично подходят для хранения в памяти кэша. Чтение индексов в памяти улучшает производительность на несколько порядков по сравнению с чтением индексов через операции ввода/вывода диска.

Серверы баз данных позволяют настроить объем системной памяти, выделяемой для кэширования. Конкретные настройки зависят от выбранного продукта. Например, в MySQL самым важным параметром в этом отношении является `innodb_buffer_pool_size`.

Многие базы данных устанавливают довольно низкий размер кэширующего буфера по умолчанию, чтобы база данных работала даже на портативных компьютерах с низким энергопотреблением. При развертывании базы данных на реальном сервере размер кэша, скорее всего, следует увеличить.

Сколько памяти выделить для кэша? Единственно правильного ответа не существует, потому что это зависит от размера базы данных, типов выполняемых запросов и объема доступной системной памяти. Обычно рекомендуется отслеживать частоту запросов ввода/вывода к диску; каждый день немного увеличивайте размер кэша, пока это помогает снизить активность дискового ввода/вывода. Не выделяйте больше памяти, чем установлено в системе, и оставьте память для других процессов и целей.

Перестройка

Индексы работают эффективнее всего, если они *сбалансированы*. Со временем при обновлении и удалении записей индексы постепенно разбалансируются

(подобно тому, как фрагментируются файловые системы). На практике можно не заметить различий между оптимальным и разбалансированным индексами. Индексы должны быть максимально эффективными, поэтому базу необходимо регулярно обслуживать.

Как часто бывает с индексами, каждый продукт использует собственную терминологию, синтаксис и функциональность.

РСУБД	**Команда технического обслуживания индекса**
IBM DB2	REORG INDEX
Microsoft SQL Server	ALTER INDEX ... REORGANIZE, ALTER INDEX ... REBUILD или DBCC DBREINDEX
MySQL	OPTIMIZE TABLE
Oracle	ALTER INDEX ... REBUILD
PostgreSQL	VACUUM or ANALYZE
SQLite	VACUUM

Как часто следует перестраивать индекс? Иногда можно услышать общие ответы из серии «раз в неделю», но универсального ответа, который бы подходил для всех приложений, не существует. Все зависит от того, насколько часто вы коммитите изменения в таблице, что также может привести к разбалансировке. Ответ зависит также от размера таблицы и от того, насколько важен выигрыш от индексов в таблице. Стоит ли тратить часы на перестройкуу индексов для большой, но редко используемой таблицы, если прирост производительности составит лишь 1 %? Это придется решать вам, потому что именно вы знаете свои данные и операционные требования лучше, чем кто-либо другой.

Механизмы извлечения максимума из индексов привязаны к конкретному продукту, так что вам придется изучить свою РСУБД. Основные источники информации — руководство по базе данных, книги и журналы, блоги и рассылки, а также самостоятельные эксперименты. Важнейшее правило: стратегия «наобум» при индексировании обычно не работает.

Знайте свои данные, знайте свои запросы и применяйте технику MENTOR при работе с индексами.

Мини-антипаттерн: индексирование каждого столбца

Некоторые разработчики создают индексы по каждому столбцу (и по каждой комбинации столбцов), если не знают, какие из индексов будут полезными для запросов. Пример приведен в разделе «Слишком много индексов», с. 170.

На самом деле создать все индексы, которые теоретически могут использоваться, намного сложнее, чем создать индекс по каждому столбцу. Вариант с использованием индексов по всем столбцам не является исчерпывающим. Запросы могут требовать составных индексов для оптимизации поиска по нескольким столбцам, или же `ORDER BY` или `GROUP BY`, или даже дополнительных столбцов для создания покрывающего индекса. Порядок столбцов в индексе тоже важен. Таким образом, чтобы указать все возможные индексы, необходимо создать столько индексов, сколько существует *перестановок* столбцов в таблице.

Index-Shotgun/mini/permutations.sql

```
CREATE TABLE Bugs (
  bug_id SERIAL PRIMARY KEY,
  date_reported DATE NOT NULL,
  summary VARCHAR(80) NOT NULL,
  status VARCHAR(10) NOT NULL,
  INDEX (bug_id, date_reported, summary, status),
  INDEX (date_reported, bug_id, summary, status),
  INDEX (summary, date_reported, bug_id, status),
  INDEX (bug_id, date_reported, summary, status),
  INDEX (summary, bug_id, date_reported, status),
  INDEX (bug_id, summary, date_reported, status),
  INDEX (date_reported, bug_id, summary, status),
  INDEX (summary, date_reported, bug_id, status),
  INDEX (status, date_reported, bug_id, summary),
  INDEX (date_reported, status, bug_id, summary),
  ...
);
```

Количество индексов, необходимых для всех перестановок, равно факториалу количества столбцов в таблице. Иначе говоря, для четырех столбцов оно равно $4 \times 3 \times 2 = 24$. При пяти столбцах для всех перестановок потребуется 120 индексов. Некоторые реализации баз данных SQL не позволяют создать столько индексов для заданной таблицы. Каждый индекс увеличивает объем памяти, необходимый для таблицы, и расходы на обновления. Создавая таблицу базы данных с таким большим количеством индексов, вы рискуете чрезмерно увеличить расходы и не получить гарантированной отдачи.

Создавайте только те индексы, которые необходимы для поддержки актуальных запросов. Если в будущем вам понадобится другой запрос, проанализируйте запросы, актуальные на тот момент, и решите, добавлять ли новый индекс.

ЧАСТЬ III

Антипаттерны запросов

Вы заполняете базу данных информацией, а затем извлекаете ее. Запросы SQL составляются на языке манипулирования данными и представляют собой такие инструкции, как SELECT, UPDATE *и* DELETE.

Есть известные известные — вещи, о которых мы знаем, что знаем их. Есть также известные неизвестные — вещи, о которых мы знаем, что не знаем. Но еще есть неизвестные неизвестные — это вещи, о которых мы не знаем, что не знаем их.

➤ *Дональд Рамсфелд (Donald Rumsfeld)*

ГЛАВА 14

Страх неизвестного

В базе данных ошибок таблица `Accounts` содержит столбцы `first_name` и `last_name`. Можно воспользоваться выражением и отформатировать полное имя пользователя в один столбец при помощи оператора конкатенации строк:

Fear-Unknown/intro/full-name.sql

```
SELECT first_name || ' ' || last_name AS full_name FROM Accounts;
```

Представим, что ваш шеф требует изменить базу данных и добавить в таблицу средний инициал отчества (например, у двух пользователей могут быть одинаковые имена и фамилии, и средний инициал помогает избежать путаницы). Это довольно простое изменение; вы вручную добавляете средние инициалы для нескольких пользователей.

Fear-Unknown/intro/middle-name.sql

```
ALTER TABLE Accounts ADD COLUMN middle_initial CHAR(2);

UPDATE Accounts SET middle_initial = 'J.' WHERE account_id = 123;
UPDATE Accounts SET middle_initial = 'C.' WHERE account_id = 321;

SELECT first_name || ' ' || middle_initial || ' ' || last_name AS full_name
FROM Accounts;
```

Неожиданно приложение перестает выводить имена. Разобравшись, вы выясняете, что ошибка происходит не всегда. Нормально выводятся имена только тех пользователей, у которых был задан средний инициал; все остальные имена неожиданно пропали.

Что происходит с остальными именами? Можно ли исправить ситуацию, пока шеф не заметил и не нагнал панику, думая, что вы потеряли все данные?

Цель: отделить отсутствующие значения

Часть полей в базе данных неизбежно будут пустыми. Например, если вы добавили запись до того, как стали известны значения всех столбцов, или некоторые столбцы не имеют смысла в каких-то случаях. SQL поддерживает специальное пустое (null) значение, выражаемое ключевым словом `NULL`. Пустые значения в таблицах и запросах SQL используются в разных ситуациях:

- Можно использовать `NULL` вместо значения, недоступного на момент создания строки, например даты расторжения договора с сотрудником, который еще работает.
- В столбце может использоваться значение `NULL`, если оно не имеет допустимого значения в заданной строке, например эффективность расхода топлива для полностью электрического автомобиля.
- Функция может возвращать значение `NULL` для неверного ввода, например `DAY('2021-12-32')`.
- Внешнее соединение использует значения `NULL` для заполнения столбцов таблицы, не имеющих соответствий.

Цель — написание запросов к столбцам, содержащим `NULL`.

Антипаттерн: использование NULL в качестве обычного значения и наоборот

Многих разработчиков поведение `NULL` в SQL застает врасплох. В отличие от большинства языков программирования, SQL рассматривает `NULL` как специальное значение, отличное от нуля, `false` или пустой строки. Это справедливо для стандартного SQL и большинства конкретных продуктов баз данных. Однако в Oracle и Sybase `NULL` эквивалентно строке нулевой длины. Значение `NULL` также имеет некоторые особенности.

Использование NULL в выражениях

Одна из нестандартных ситуаций — выполнение арифметических вычислений со столбцом или выражением, дающее результат `NULL`. Например, многие считают, что следующий запрос вернет `10` для ошибок, не имеющих оценки в столбце `hours`, но вместо этого запрос вернет `NULL`.

Fear-Unknown/anti/expression.sql

```
SELECT hours + 10 FROM Bugs;
```

NULL не то же самое, что ноль. Число, которое на 10 больше неизвестной величины, все еще остается неизвестной величиной.

NULL не то же самое, что строка нулевой длины. Объединение любой строки с NULL в стандартном SQL возвращает NULL (несмотря на поведение Oracle и Sybase).

NULL не то же самое, что false. Результаты логических выражений с AND, OR и NOT тоже многим кажутся странными.

Поиск столбцов, содержащих NULL

Следующий запрос возвращает только строки, в которых assigned_to содержит значение 123, а не строки с другими значениями или строки, в которых столбец содержит NULL:

Fear-Unknown/anti/search.sql

```
SELECT * FROM Bugs WHERE assigned_to = 123;
```

Можно подумать, что следующий запрос возвращает дополняющий набор строк (то есть все строки, *не возвращенные* предыдущим запросом):

Fear-Unknown/anti/search-not.sql

```
SELECT * FROM Bugs WHERE NOT (assigned_to = 123);
```

Однако ни один из результатов запроса не включает строки, в которых assigned_to содержит NULL. Любое сравнение с NULL возвращает *неизвестный* результат, не true и не false. Даже отрицание NULL все еще дает NULL.

Одна из типичных ошибок — поиск значений NULL или отличных от NULL:

Fear-Unknown/anti/equals-null.sql

```
SELECT * FROM Bugs WHERE assigned_to = NULL;

SELECT * FROM Bugs WHERE assigned_to <> NULL;
```

Условие в секции WHERE выполняется только в том случае, если выражение истинно, но результат сравнения с NULL никогда не бывает истинным; это неизвестное значение. Неважно, проверяет ли сравнение равенство или неравенство; оно все равно неизвестно, что, безусловно, не является истиной. Ни один из предыдущих запросов не возвращает строки, в которых assigned_to содержит NULL.

Использование NULL в параметрах запросов

Также трудно использовать NULL в параметризованных выражениях SQL, как если бы NULL являлся обычным значением.

Fear-Unknown/anti/parameter.sql

```
SELECT * FROM Bugs WHERE assigned_to = ?;
```

Предыдущий запрос возвращает предсказуемые результаты, когда вы передаете обычное целое значение параметра, но литерал NULL не может использоваться как параметр.

Предотвращение проблемы

Если обработка NULL усложняет запросы, многие разработчики решают запретить NULL в базе данных. Вместо него они выбирают целое значение, которое обозначает «неизвестное» или «неприменимое» значение.

> **«Мы ненавидим NULL!»**
>
> Один разработчик по имени Джек рассказал мне о просьбе клиента избегать значений NULL в базе данных. Объяснение было простым: «Мы ненавидим NULL!», и присутствие NULL приводит к ошибкам в коде приложения. Джек спросил меня, что можно использовать для представления отсутствующего значения.
>
> Я сказал Джеку, что значение NULL существует как раз для представления отсутствующих значений. Какое бы другое значение он ни выбрал для этой цели, ему придется менять код приложения, чтобы это значение интерпретировалось как специальное.
>
> Подход клиента Джека к NULL был ошибочным; с таким же успехом клиент мог сказать, что ему не нравится писать код, предотвращающий ошибки деления на ноль, но отсюда не следует, что нужно полностью отказаться от использования нуля.

Что же не так с использованием вместо NULL значений, отличных от NULL? В следующем примере столбцы assigned_to и hours, ранее допускавшие NULL, объявляются с NOT NULL:

Fear-Unknown/anti/special-create-table.sql

```
CREATE TABLE Bugs (
  bug_id SERIAL PRIMARY KEY,
  -- другие столбцы
  assigned_to BIGINT UNSIGNED NOT NULL,
  hours NUMERIC(9,2) NOT NULL,
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id)
);
```

Вы можете попытаться использовать -1 для представления неизвестных значений.

Fear-Unknown/anti/special-insert.sql

```
INSERT INTO Bugs (assigned_to, hours) VALUES (-1, -1);
```

Столбец hours является числовым, так что для представления «неопределенного» значения должно использоваться число. Оно не должно иметь смысла в этом столбце, поэтому выбираем отрицательное значение. К сожалению, значение -1 нарушает такие вычисления, как SUM() или AVG(). Придется исключать строки с этим значением при помощи выражений особого случая, а ведь именно этого вы пытались избежать, запрещая NULL.

Fear-Unknown/anti/special-select.sql

```
SELECT AVG( hours ) AS average_hours_per_bug FROM Bugs
WHERE hours <> -1;
```

В другом столбце значение -1 может иметь смысл, поэтому придется выбирать разные значения в каждом конкретном случае для каждого столбца. И все специальные значения придется запоминать или записывать. Все это добавляет нудной и лишней работы.

Теперь взгляните на столбец assigned_to. Он является внешним ключом для таблицы Accounts. Если ошибка уже зарегистрирована в системе, но еще не назначена исполнителю, какое отличное от NULL значение можно использовать в этом столбце? Любое значение, отличное от NULL, должно ссылаться на строку в Accounts, поэтому в Accounts придется создать фиктивную строку, которая означает «никто» или «не назначено». Немного странно создавать учетную запись, на которую будет ссылаться другая таблица и которая будет использоваться для представления отсутствия ссылки на учетную запись реального пользователя.

Объявление столбца с NOT NULL должно означать, что существование строк данных без значения в этом столбце не имеет смысла. Например, столбец Bugs.reported_by column должен содержать значение, потому что о каждой ошибке кто-то должен сообщить. С другой стороны, может существовать ошибка, которая еще не назначена исполнителю. Отсутствующие значения исполнителей должны быть объявлены NULL.

Как распознать антипаттерн

Если вы слышите, как ваши коллеги произносят следующие фразы, это может быть связано с некорректной обработкой NULL:

- «Как найти строки, в которых столбцу assigned_to (или другому) не присвоено значение?»

  Для NULL невозможно использовать оператор равенства. Вы узнаете, как использовать предикат IS NULL, далее в этой главе.
- «Полные имена некоторых пользователей в приложении выводятся пустыми, но я вижу их в базе данных».

Проблема в том, что строковые значения конкатенируют с `NULL`, что дает результат `NULL`.

- «В отчет о затратах рабочего времени вошли только несколько обработанных ошибок! Попали только ошибки, которым был назначен приоритет».

  Скорее всего, в агрегатный запрос для суммирования затрат времени было включено выражение, в котором условие `WHERE` не равно `true`, если `priority` содержит `NULL`. Результаты использования выражений *неравенства* могут быть неожиданными. Например, для строк, в которых `priority` содержит `NULL`, выражение `priority <> 1` не сработает.
- «Оказывается, для представления "неизвестно" мы не можем использовать ту же строку, что и в таблице `Bugs`. Нужно собраться и решить, какое специальное значение использовать, а также оценить, сколько времени потребуется для миграции данных и преобразования кода на использование нового значения».
- Обычно такие проблемы возникают из-за выбора специального значения флага, которое может оказаться допустимым в предметной области столбца. Со временем оказывается, что это значение должно использоваться с буквальным, а не специальным смыслом.

Обнаружить ошибки, связанные с обработкой `NULL`, бывает сложно. Проблемы могут быть не видны в ходе тестирования приложения, особенно если вы упустили некоторые граничные случаи при выборе данных для тестов. Но когда приложение развертывается в продакшен, данные начинают принимать множество непредвиденных форм. Если `NULL` может проникнуть в данные, не сомневайтесь — это непременно случится.

Соответствует ли NULL реляционной теории?

По поводу NULL в SQL существуют разные мнения. Эдгар Ф. Кодд (E. F. Codd) — ученый, разработавший реляционную теорию, — признал необходимость NULL для обозначения отсутствующих данных. Однако Кристофер Дж. Дейт (C. J. Date) показал, что у поведения NULL, определенного в стандарте SQL, имеются граничные случаи, противоречащие реляционной логике.

Приходится признать, что многие языки программирования не идеальны в реализации компьютерных теорий. В языке SQL поддерживается NULL, нравится нам это или нет. Некоторые риски этого были описаны выше, но вы можете научиться учитывать такие случаи и эффективно использовать NULL.

Допустимые применения антипаттерна

Использование NULL не может считаться антипаттерном. Антипаттерном является использование NULL как обычного значения или использование обычного значения как NULL.

Одна из ситуаций, в которых приходится работать с NULL как с обычным значением, встречается при импортировании и экспортировании внешних данных. В текстовом файле с полями, разделенными запятыми, все значения должны быть представлены в текстовом виде. Например, в инструменте MySQL `mysqlimport`, предназначенном для загрузки данных из текстового файла, `\N` представляет NULL.

Аналогичным образом NULL не может иметь прямого представления в пользовательском вводе. Любое приложение, в котором предусмотрен пользовательский ввод, должно соотносить определенную последовательность вводимых символов с NULL. Например, Microsoft .NET 2.0 и более поздних версий поддерживает свойство `ConvertEmptyStringToNull` для пользовательских интерфейсов веб-приложений. Параметры и связанные поля с этим свойством автоматически преобразуют пустую строку (`""`) в NULL.

Наконец, NULL не работает, если требуется различать виды отсутствующих значений. Допустим, необходимо отличать ошибки, которые ни за кем не закреплены, от ошибок, которые были закреплены за сотрудником, покинувшим проект, и использовать отдельное значение для каждого состояния.

Решение: использование NULL как уникального значения

Многие проблемы с NULL являются следствием частых ошибочных представлений о поведении тройственной логики значений SQL. Для разработчиков, привыкших к традиционной логике `true/false`, реализованной в большинстве других языков, она может представлять сложность. Чтобы успешно работать со значениями NULL в запросах SQL, необходимо разобраться в их механике.

NULL в скалярных выражениях

Допустим, Стэну 30 лет, а возраст Оливера неизвестен. Если вы спрашиваете: «Правда ли, что Стэн старше Оливера?», возможен только один ответ: «Я не знаю». Если вы спросите: «Правда ли, что Стэн одного возраста с Оливером?», ответ будет тем же: «Я не знаю». Если вы спросите, какая сумма получится при сложении возрастов Стэна и Оливера, ответ не изменится.

Возраст Чарли тоже неизвестен. Если вы спросите: «Правда ли, что Оливер и Чарли одного возраста?», ответ остается прежним: «Я не знаю». Это объясняет, почему результатом сравнений вида `NULL = NULL` также будет NULL.

В следующей таблице представлены некоторые ситуации, в которых разработчики ожидают одного результата, а получают совсем другой.

Выражение	Ожидается	На самом деле	Потому что
`NULL = 0`	`TRUE`	`NULL`	NULL не равно 0
`NULL = 12345`	`FALSE`	`NULL`	Невозможно определить, равно ли неизвестное значение заданному значению
`NULL <> 12345`	`TRUE`	`NULL`	Также невозможно определить, равны ли они
`NULL + 12345`	`12345`	`NULL`	NULL не равно 0
`NULL || 'string'`	`'string'`	`NULL`	NULL не равно пустой строке
`NULL = NULL`	`TRUE`	`NULL`	Невозможно определить, равно ли одно неизвестное значение другому
`NULL <> NULL`	`FALSE`	`NULL`	Также невозможно определить, различны ли они

Конечно, эти примеры справедливы не только при использовании ключевого слова `NULL`, но и для любого столбца или выражения со значением `NULL`.

NULL в булевых выражениях

Значение `NULL` определенно не равно `true`, но оно и не равно `false`. Если бы оно было равно `false`, то применение `NOT` к `NULL` давало бы результат `true`. Однако на практике это не так; результатом `NOT (NULL)` снова будет `NULL`. Это смущает тех, кто пытается использовать булевы выражения с `NULL`.

В следующей таблице представлены некоторые ситуации, в которых разработчики ожидают одного результата, а получают совсем другой.

Выражение	Ожидается	На самом деле	Потому что
`NULL AND TRUE`	`FALSE`	`NULL`	NULL не равно false
`NULL AND FALSE`	`FALSE`	`FALSE`	Любое значение `AND FALSE` дает FALSE
`NULL OR FALSE`	`FALSE`	`NULL`	NULL не равно false
`NULL OR TRUE`	`TRUE`	`TRUE`	Любое значение `OR TRUE` дает TRUE
`NOT (NULL)`	`TRUE`	`NULL`	NULL не равно false

Верный результат по неверным причинам

Рассмотрим ситуацию, в которой столбец, допускающий NULL, по чистой случайности обладает интуитивно понятным поведением.

```
SELECT * FROM Bugs WHERE assigned_to <> 'NULL';
```

Здесь столбец assigned_to, допускающий NULL, сравнивается со строковым значением 'NULL' (обратите внимание на кавычки) вместо ключевого слова NULL.

Когда assigned_to содержит NULL, сравнение со строкой 'NULL' не будет истинным. Строка исключается из результатов запроса, что соответствует намерениям разработчика.

Другой случай: столбец, содержащий целое число, сравнивается со строкой 'NULL'. В большинстве РСУБД интерпретация строки (такой, как 'NULL') как целого числа дает ноль. Целое значение assigned_to почти наверняка больше нуля. Оно не равно строке, поэтому строка включается в результат запроса.

Таким образом, совершая еще одну распространенную ошибку с заключением NULL в кавычки, некоторые разработчики случайно получают нужный результат. К сожалению, эта случайность не соблюдается в других видах поиска, например WHERE assigned_to = 'NULL'.

Поиск NULL

Так как ни проверка равенства, ни проверка неравенства не возвращает `true` при сравнении значения с `NULL`, для поиска `NULL` потребуется другая операция. В более старых стандартах SQL определяется предикат `IS NULL`, который возвращает `true`, если его единственным операндом является `NULL`. Обратный предикат `IS NOT NULL` возвращает `false`, если его операндом является `NULL`.

Fear-Unknown/soln/search.sql

```
SELECT * FROM Bugs WHERE assigned_to IS NULL;

SELECT * FROM Bugs WHERE assigned_to IS NOT NULL;
```

Кроме того, стандарт SQL-99 определяет другой предикат сравнения — `IS DISTINCT FROM`. Он работает как обычный оператор проверки неравенства `<>`, не считая того, что он всегда возвращает `true` или `false` — даже в том случае, если все операнды содержат `NULL`. Это избавляет от утомительного написания выражений, которые должны проверять `IS NULL` перед сравнением значения. Следующие два запроса эквивалентны:

Fear-Unknown/soln/is-distinct-from.sql

```
SELECT * FROM Bugs WHERE assigned_to IS NULL OR assigned_to <> 1;

SELECT * FROM Bugs WHERE assigned_to IS DISTINCT FROM 1;
```

Этот предикат может использоваться с параметрами запроса, которому должно передаваться литеральное значение или `NULL`:

Fear-Unknown/soln/is-distinct-from-parameter.sql

```
SELECT * FROM Bugs WHERE assigned_to IS DISTINCT FROM ?;
```

`IS DISTINCT FROM` не имеет последовательной поддержки в базах данных. В PostgreSQL, IBM DB2 и Firebird конструкция поддерживается, а в Oracle и Microsoft SQL Server пока нет. MySQL предлагает собственный оператор `<=>`, который работает как `IS NOT DISTINCT FROM`.

Объявление столбцов с ограничением NOT NULL

Рекомендуется объявлять ограничение `NOT NULL` для столбца, в котором `NULL` нарушает политику приложения или не имеет смысла по другим причинам. Лучше поручить контроль за ограничениями базе данных, а не полагаться на код приложения.

Например, разумно предположить, что любая запись в таблице `Bugs` должна содержать отличные от `NULL` значения для столбцов `date_reported`, `reported_by` и `status`. Точно так же строки дочерних таблиц (таких, как `Comments`) должны включать отличное от `NULL` значение `bug_id`, ссылающееся на существующую ошибку. Такие столбцы следует объявлять с ограничением `NOT NULL`.

Некоторые разработчики рекомендуют определять значение по умолчанию (`DEFAULT`) для каждого столбца. Если такой столбец будет опущен в инструкции `INSERT`, то он получит некоторое значение вместо `NULL`. Этот совет хорошо подходит для некоторых столбцов, хотя и не для всех. Например, столбец `Bugs.reported_by` не должен содержать `NULL`. Это должен быть идентификатор учетной записи пользователя, сообщившего об ошибке, но его невозможно объявить как значение по умолчанию. Столбцы часто объявляются с `NOT NULL`, даже в том случае, если они не имеют разумного значения по умолчанию.

Динамические значения по умолчанию

Иногда для упрощения логики запроса требуется, чтобы столбец или выражение были отличны от `NULL`, но при этом значение не должно храниться в таблице. Необходимо каким-то образом присвоить отличное от `NULL` значение, которое будет использоваться, если заданное выражение возвращает результат `NULL`. Для

этого можно воспользоваться функцией COALESCE(). Функция получает переменное количество аргументов и возвращает первый аргумент, отличный от NULL.

В сценарии с конкатенацией имен пользователей в начале главы можно воспользоваться функцией COALESCE() и создать выражение, которое использует один пробел вместо среднего инициала, чтобы результат всего выражения не превращался в NULL из-за среднего инициала со значением NULL.

Fear-Unknown/soln/coalesce.sql

```
SELECT first_name || COALESCE(' ' || middle_initial || ' ', ' ') || last_name
  AS full_name
FROM Accounts;
```

COALESCE() принадлежит к числу стандартных функций SQL. Некоторые РСУБД поддерживают похожую функцию с другим именем, например NVL() или ISNULL().

Используйте NULL для представления отсутствующего значения любого типа данных.

Мини-антипаттерн: NOT IN (NULL)

Если логика NULL вдруг покажется вам недостаточно запутанной, существуют граничные случаи, при наступлении которых еще труднее не потеряться в происходящем.

Надеюсь, вы достаточно хорошо разбираетесь в логике и понимаете, что следующие два запроса эквивалентны:

Fear-Unknown/mini/in-null.sql

```
SELECT * FROM Bugs WHERE status IN (NULL, 'NEW');

SELECT * FROM Bugs WHERE status = NULL OR status = 'NEW';
```

Вы знаете, что проверка на равенство с NULL дает неизвестный результат, а не true, так что первая часть этого сравнения никогда не выполняется. И это нормально, потому что запрос все равно будет находить строки с 'NEW'.

Самое интересное начинается при инвертировании запроса:

Fear-Unknown/mini/not-in-null.sql

```
SELECT * FROM Bugs WHERE status NOT IN (NULL, 'NEW');
```

Можно подумать, что этот запрос просто находит дополнение для набора строк, найденных предыдущим запросом, другими словами, все строки, кроме строк со статусом `'NEW'`. Но на самом деле не будет найдена *ни одна* строка. Почему?

Запрос с предикатом `NOT IN` можно переписать одним из следующих способов:

Fear-Unknown/mini/not-in-null.sql

```
SELECT * FROM Bugs WHERE NOT (status = NULL OR status = 'NEW');

SELECT * FROM Bugs WHERE NOT (status = NULL) AND NOT (status = 'NEW');
```

Первый вариант выглядит знакомо, так как предикат `IN` эквивалентен проверкам на равенство с каждым соответствующим значением, объединенным операциями `OR`. Затем к выражению применяется отрицание `NOT`. Вы уже знаете, что проверка столбца на равенство с `NULL` дает неизвестный результат, а отрицание неизвестного результата остается неизвестным.

Второй вариант является применением *закона де Моргана* — преобразования из области булевой алгебры. При отрицании выражения отрицается каждая его составляющая, а `OR` превращается в `AND` и наоборот.

Теперь вы видите, что результат `NOT (status = NULL)` остается неизвестным, а использование `AND` для объединения его с другим подвыражением делает все выражение неизвестным для любой проверяемой строки. Поэтому запрос SQL не находит ни одной записи независимо от значения в столбце `status`.

Интеллект отличает возможное от невозможного;
здравый смысл отличает целесообразное
от бессмысленного. Даже возможное может
быть бессмысленным.

➤ *Макс Борн (Max Born)*

ГЛАВА 15

Неоднозначные группы

Представим, что ваш руководитель хочет знать, какие проекты в базе данных `bugs` активны, а какие решено закрыть. В частности, он просит вас сгенерировать отчет о последних ошибках, зарегистрированных для каждого продукта. Вы пишете запрос к базе данных MySQL, вычисляющий наибольшее значение в столбце `date_reported` в группе ошибок с заданным значением `product_id`. Отчет выглядит так:

product_name	**последняя**	**bug_id**
Open RoundFile	2010-06-01	1234
Visual TurboBuilder	2010-02-16	3456
ReConsider	2010-01-01	5678

Шеф внимателен к деталям и проверяет все ошибки, указанные в отчете. Он замечает, что в строке, отмеченной как самая недавняя для Open RoundFile, значение `bug_id` не соответствует последней ошибке. Чтобы убедиться в этом, достаточно просмотреть полные данные:

product_name	**date_reported**	**bug_id**	
Open RoundFile	2009-12-19	1234	Это значение bug_id...
Open RoundFile	2010-06-01	2248	не соответствует этой дате
Visual TurboBuilder	2010-02-16	3456	
Visual TurboBuilder	2010-02-10	4077	
Visual TurboBuilder	2010-02-16	5150	
ReConsider	2010-01-01	5678	
ReConsider	2009-11-09	8063	

Что происходит? Почему ошибка возникла в одном продукте, а не во всех? Как получить нужный отчет?

Цель: получить строку с наибольшим значением в группе

Многие разработчики, изучающие SQL, в какой-то момент добираются до использования `GROUP BY` в запросах, применяют агрегатную функцию к группам записей и получают результат, содержащий одну строку на группу. Это мощный механизм, который позволяет легко получать разнообразные сводные отчеты относительно небольшим объемом кода.

Например, запрос для получения последней ошибки, зарегистрированной для каждого продукта в базе данных `bugs`, может выглядеть так:

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

Естественным расширением этого запроса становится получение идентификатора конкретной ошибки с последней датой регистрации:

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest, bug_id
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

Однако этот запрос возвращает либо ошибку, либо недостоверный ответ. Разработчики, использующие SQL, часто в нем путаются.

Цель — написать запрос, который не только возвращает наибольшее значение в группе (или наименьшее, или среднее), но и включает другие атрибуты строки, в которой было найдено это значение.

Антипаттерн: ссылка на столбцы, не входящие в группу

Исходная причина этого антипаттерна проста — типичное и частое непонимание того, как работают группирующие запросы в SQL.

Правило одного значения

В каждую группу входят строки, содержащие одинаковые значения в столбце или столбцах, указанных после GROUP BY. Например, в следующем запросе создается одна группа строк для каждого отдельного значения в product_id.

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

Каждый столбец в SELECT-списке запроса должен содержать одно и то же значение на группу строк. Это называется *правилом одного значения*. Столбцы, указанные в секции GROUP BY, гарантированно содержат ровно одно значение на группу независимо от того, сколько строк включает группа.

Выражение MAX() также гарантированно дает одно значение для каждой группы: наибольшее значение, встречающееся в аргументе MAX() по всем строкам группы.

Однако сервер базы данных не дает такой гарантии относительно любого другого столбца, указанного в SELECT-списке. Невозможно гарантировать, что в других столбцах во всех строках группы содержится одно значение.

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest, bug_id
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

В этом примере для заданного product_id существует много разных значений bug_id, поскольку таблица BugsProducts связывает несколько ошибок с заданным продуктом. В группирующем запросе, который сокращает набор данных до одной строки на продукт, нет возможности представить все значения bug_id.

Так как для «дополнительных» столбцов гарантий одного значения на группу нет, база данных предполагает, что они нарушат правило одного значения. Большинство РСУБД выдают ошибку при попытке выполнить любой запрос, который пытается вернуть другой столбец, кроме перечисленных в секции GROUP BY или в аргументах агрегатных функций.

MySQL и SQLite ведут себя иначе, чем другие РСУБД, как мы увидим в разделе «Допустимые применения антипаттерна» данной главы, с. 199.

Запросы «сделай то, о чем я думаю»

Многие разработчики ошибочно думают, что SQL определит, какое значение bug_id включить в отчет, на основании того, что MAX() используется с другим столбцом. Они предполагают, что если запрос извлекает наибольшее значение,

то другие перечисленные столбцы естественно примут свое значение из строки, в которой оно находится.

К сожалению, в некоторых случаях SQL не может сделать такой вывод:

- Если две ошибки имеют одинаковое значение `date_reported`, которое является наибольшим значением в группе, какое значение `bug_id` должно быть включено в отчет?
- Если в запросе используются две разные агрегатные функции (например, `MAX()` и `MIN()`), вероятно, они соответствуют двум разным строкам в группе. Какое значение `bug_id` должен вернуть запрос для этой группы?

Groups/anti/maxandmin.sql

```
SELECT product_id, MAX(date_reported) AS latest,
  MIN(date_reported) AS earliest, bug_id
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

- Если ни одна из строк в таблице не совпадает со значением, возвращенным агрегатной функцией, каким должно быть значение `bug_id`? Обычно это относится к функциям `AVG()`, `COUNT()` и `SUM()`.

Groups/anti/sumbyproduct.sql

```
SELECT product_id, SUM(hours) AS total_project_estimate, bug_id
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

GROUP BY и DISTINCT

SQL поддерживает модификатор запроса DISTINCT, который выполняет свертку результатов запроса, чтобы каждая строка была уникальной. Например, следующий запрос сообщает, кто и в какие дни регистрировал ошибки, но только по одной строке на дату и человека:

```
SELECT DISTINCT date_reported, reported_by FROM Bugs;
```

Группирующим запросом можно получить тот же результат без агрегатных функций. Результат запроса сокращается до одной строки на каждую отдельную пару значений в столбцах, указанных в секции GROUP BY:

```
SELECT date_reported, reported_by FROM Bugs
GROUP BY date_reported, reported_by;
```

Оба запроса дают одинаковые результаты, их оптимизация и выполнение должны выполняться примерно одинаково. В данном примере уместнее использовать DISTINCT, потому что запрос не содержит агрегатных функций.

Все эти примеры показывают, почему правило одного значения так важно. Не каждый запрос, который нарушает это правило, выдает неоднозначный результат, но это происходит довольно часто. Умная база данных могла бы отличать неоднозначные вопросы от однозначных и выдавать ошибку, только если данные неоднозначны. Однако это снижает надежность приложения, так как один и тот же запрос может оказаться действительным или недействительным в зависимости от состояния данных.

Как распознать антипаттерн

Во многих РСУБД нарушение правила одного значения должно приводить к ошибке сразу же при написании запроса. Вот примеры сообщений об ошибках, выводимых в некоторых РСУБД:

- IBM DB2:

```
BAn expression starting with "BUG_ID" specified in a SELECT clause, HAVING
clause, or ORDER BY clause is not specified in the GROUP BY clause or it is
in a SELECT clause, HAVING clause, or ORDER BY clause with a column function
and no GROUP BY clause is specified.

(Выражение, начинающееся с “BUG_ID”, указанное в секции SELECT, секции HAVING
или секции ORDER BY, не указано в секции GROUP BY или присутствует в секции
SELECT, секции HAVING или секции ORDER BY со столбцовой функцией и без
указания секции GROUP BY.)
```

- Microsoft SQL Server:

```
Column 'Bugs.bug_id' is invalid in the select list because it is not
contained in either an aggregate function or the GROUP BY clause.

(Столбец 'Bugs.bug_id' недопустим в списке select, поскольку он не содержится
ни в агрегатной функции, ни в секции GROUP BY.)
```

- MySQL, начиная с версии 5.7, включает режим SQL `ONLY_FULL_GROUP_BY` по умолчанию для предотвращения неоднозначных запросов.

```
ERROR 1055 (42000): Expression #3 of SELECT list is not in GROUP BY
clause and contains nonaggregated column 'test.Bugs.bug_id'
which is not functionally dependent on columns in GROUP BY clause;
this is incompatible with sql_mode=only_full_group_by.

(Выражение #3 списка SELECT не входит в секцию GROUP BY и содержит неагреги-
рованный столбец 'test.Bugs.bug_id', который не связан функциональной
зависимостью со столбцами в секции GROUP BY; это несовместимо
с sql_mode=only_full_group_by.)
```

- Oracle:

```
not a GROUP BY expression

(Не является выражением GROUP BY)
```

- PostgreSQL:

```
column "bp.bug_id" must appear in the GROUP BY clause or be used in an
aggregate function.

(Столбец "bp.bug_id" должен входить в секцию GROUP BY или использоваться
в агрегатной функции.)
```

В SQLite и в MySQL, если режим SQL `ONLY_FULL_GROUP_BY` не включен, неоднозначные столбцы могут содержать непредвиденные и недостоверные значения. В MySQL возвращаемое значение берется из первой строки в группе — «первой» в порядке физического хранения. SQLite возвращает обратный результат: значение из последней строки группы. В обоих случаях поведение не документировано, и эти базы данных не обязательно будут работать так же в следующих версиях. Вам придется следить за подобными случаями и составлять запросы так, чтобы избежать неоднозначности.

Допустимые применения антипаттерна

Как было показано выше, MySQL и SQLite не гарантируют надежного результата для столбца, не соответствующего правилу одного значения. Однако иногда можно извлечь пользу из того факта, что эти базы данных соблюдают указанное правило менее жестко, чем другие продукты.

Groups/legit/functional.sql

```
SELECT b.reported_by, a.account_name
FROM Bugs b JOIN Accounts a ON (b.reported_by = a.account_id)
GROUP BY b.reported_by;
```

В приведенном запросе столбец `account_name` формально нарушает правило одного значения, так как он не указан ни в секции `GROUP BY`, ни в агрегатной функции. Тем не менее в каждой группе существует только одно возможное значение для `account_name`; группы определяются на основании `Bugs.reported_by`, внешнего ключа для таблицы `Accounts`. Таким образом, у групп существует однозначное соответствие со строками таблицы `Accounts`.

Иначе говоря, если вам известно значение `reported_by`, то также однозначно известно и значение `account_name`, как если бы запрос проводился по первичному ключу таблицы `Accounts`.

Подобные однозначные отношения называются *функциональной зависимостью*. Самый распространенный пример такого рода существует между первичным ключом таблицы и ее атрибутами: `account_name` является функциональной зависимостью первичного ключа `account_id`. Если сгруппировать запрос по столбцу (столбцам) первичного ключа таблицы, то группы будут соответствовать

одной строке этой таблицы, так что другие столбцы той же таблицы обязательно будут содержать одно значение на группу.

Атрибут `Bugs.reported_by` связывается аналогичным отношением с зависимыми атрибутами таблицы `Accounts`, поскольку он ссылается на первичный ключ таблицы `Accounts`. Когда запрос группируется по столбцу `reported_by`, который является внешним ключом, атрибуты таблицы `Accounts` становятся функционально зависимыми, а результат запроса не содержит неоднозначности.

Тем не менее большинство РСУБД все равно возвращает ошибку. Дело не только в том, что этого требует стандарт SQL; программе сложно найти все функциональные зависимости. Если вы используете MySQL или SQLite и тщательно составляете запросы так, чтобы в них использовались только функционально зависимые столбцы, то сможете избежать проблем с неоднозначностью.

Решение: выборка однозначных столбцов

В этом разделе описаны некоторые способы исправления этого антипаттерна и написания однозначных запросов.

Запросы только к функционально зависимым столбцам

Самое простое решение — удалить неоднозначные столбцы из запроса.

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

Запрос возвращает дату последней ошибки для каждого продукта, пусть и не возвращая соответствующее значение `bug_id` последней ошибки. Иногда этого достаточно, так что не пренебрегайте простыми решениями.

Использование оконной функции

Современные продукты SQL реализуют *оконные функции*, которые могут использоваться для фильтрации по первой (или последней) строке группы. Чтобы убедиться, что ваша версия поддерживает эти функции, обратитесь к документации своей РСУБД. Например, для поддержки функций ниже необходима версия MySQL 8.0.

Groups/soln/window-function.sql

```
SELECT t.product_id, t.date_reported, t.bug_id
FROM (
  SELECT bp.product_id, b.date_reported, b.bug_id,
```

```
    ROW_NUMBER() OVER (PARTITION BY bp.product_id
      ORDER BY b.date_reported DESC) AS rownum
  FROM Bugs b JOIN BugsProducts bp USING (bug_id)
) AS t
WHERE t.rownum = 1;
```

Столбец `rownum`, возвращаемый подзапросом, заново начинает нумерацию с единицы для каждого раздела, то есть каждого набора строк данных, сгруппированных по `product_id`. В соответствии с условием во внешнем запросе в результат будет включена только первая строка каждого раздела.

Использование коррелированного подзапроса

Коррелированный подзапрос содержит ссылку на внешний запрос и, как следствие, генерирует разные результаты для каждой строки внешнего запроса. Можно воспользоваться этой возможностью для нахождения последней ошибки по продукту, выполняя подзапрос для поиска ошибок с тем же продуктом и более поздней датой. Если подзапрос ничего не находит, значит, ошибка во внешнем запросе является последней.

Groups/soln/notexists.sql

```
SELECT bp1.product_id, b1.date_reported AS latest, b1.bug_id
FROM Bugs b1 JOIN BugsProducts bp1 USING (bug_id)
WHERE NOT EXISTS
  (SELECT * FROM Bugs b2 JOIN BugsProducts bp2 USING (bug_id)
   WHERE bp1.product_id = bp2.product_id
     AND b1.date_reported < b2.date_reported);
```

Это простое решение понятно и легко реализуемо. Однако стоит учесть, что его производительность вряд ли будет высока, потому что коррелированные подзапросы выполняются по одному разу для каждой строки внешнего запроса.

Использование производной таблицы

Подзапрос может использоваться как *производная таблица*; в этом случае генерируется промежуточный результат, который содержит только `product_id` и соответствующую дату сообщения об ошибке для каждого продукта. Затем этот результат используется для соединения таблиц, чтобы результат запроса содержал только ошибки с последней датой для каждого продукта.

Groups/soln/derived-table.sql

```
SELECT m.product_id, m.latest, b1.bug_id
FROM Bugs b1 JOIN BugsProducts bp1 USING (bug_id)
  JOIN (SELECT bp2.product_id, MAX(b2.date_reported) AS latest
   FROM Bugs b2 JOIN BugsProducts bp2 USING (bug_id)
   GROUP BY bp2.product_id) m
  ON (bp1.product_id = m.product_id AND b1.date_reported = m.latest);
```

product_id	latest	bug_id
1	2010-06-01	2248
2	2010-02-16	3456
2	2010-02-16	5150
3	2010-01-01	5678

Стоит отметить, что можно получать несколько строк на продукт, если последняя дата, возвращенная подзапросом, соответствует нескольким записям. Если вам нужно, чтобы для `product_id` возвращалась только одна строка, можно использовать другую группирующую функцию во внешнем запросе:

Groups/soln/derived-table-no-duplicates.sql

```
SELECT m.product_id, m.latest, MAX(b1.bug_id) AS latest_bug_id
FROM Bugs b1 JOIN
  (SELECT product_id, MAX(date_reported) AS latest
   FROM Bugs b2 JOIN BugsProducts USING (bug_id)
   GROUP BY product_id) m
  ON (b1.date_reported = m.latest)
GROUP BY m.product_id, m.latest;
```

product_id	latest	bug_id
1	2010-06-01	2248
2	2010-02-16	5150
3	2010-01-01	5678

Используйте решение с производной таблицей как более масштабируемую альтернативу коррелированному подзапросу. Производная таблица не коррелирована, поэтому большинство РСУБД должны выполнять подзапрос однократно. Однако база данных должна хранить промежуточный результат, сгенерированный во временной таблице, так что у этого решения будет не лучшая производительность.

Использование JOIN

Можно создать соединение, находящее совпадение с набором строк, которые не существуют. Такие соединения называются *внешними соединениями*. Если совпадающие строки не существуют, для всех столбцов этой несуществующей строки используется `NULL`. Таким образом, если запрос возвращает `NULL`, мы знаем, что подходящая строка не найдена.

Groups/soln/outer-join.sql

```
SELECT bp1.product_id, b1.date_reported AS latest, b1.bug_id
FROM Bugs b1 JOIN BugsProducts bp1 ON (b1.bug_id = bp1.bug_id)
```

```
LEFT OUTER JOIN
  (Bugs AS b2 JOIN BugsProducts AS bp2 ON (b2.bug_id = bp2.bug_id))
  ON (bp1.product_id = bp2.product_id
    AND (b1.date_reported < b2.date_reported
    OR b1.date_reported = b2.date_reported AND b1.bug_id < b2.bug_id))
WHERE b2.bug_id IS NULL;
```

product_id	latest	bug_id
1	2010-06-01	2248
2	2010-02-16	5150
3	2010-01-01	5678

Многим, чтобы понять, как работает этот запрос, придется тщательно его поизучать и, возможно, расписать на бумаге. Но когда вы поймете, что он делает, этот прием может стать ценным инструментом.

Используйте решение с `JOIN`, когда важна масштабируемость запроса для больших наборов данных. Хотя эта концепция сложнее для понимания и, как следствие, ее сложнее поддерживать, она часто масштабируется лучше решений, основанных на подзапросах. Обязательно измерьте производительность разных форм запроса; недостаточно просто предположить, что какой-то один будет эффективнее другого.

Использование агрегатной функции для дополнительных столбцов

Чтобы обеспечить соблюдение правила одного значения дополнительным столбцом, можно применить к нему еще одну агрегатную функцию.

Groups/soln/extra-aggregate.sql

```
SELECT product_id, MAX(date_reported) AS latest,
  MAX(bug_id) AS latest_bug_id
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

Используйте это решение, только если вы твердо уверены, что последнее значение `bug_id` соответствует ошибке с последней датой, иначе говоря, если ошибки расположены точно в хронологическом порядке.

Конкатенация всех значений в группе

Наконец, можно использовать другую агрегатную функцию с `bug_id`, чтобы предотвратить нарушение правила одного значения. MySQL и SQLite поддерживают функцию `GROUP_CONCAT()`, которая выполняет конкатенацию всех зна-

чений в группе в одно значение. По умолчанию результат выводится в строке, разделенной запятыми.

Groups/soln/group-concat-mysql.sql

```
SELECT product_id, MAX(date_reported) AS latest
  GROUP_CONCAT(bug_id) AS bug_id_list,
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

product_id	latest	bug_id_list
1	2010-06-01	1234,2248
2	2010-02-16	3456,4077,5150
3	2010-01-01	5678,8063

Этот запрос не показывает, какое значение `bug_id` соответствует последней дате; `bug_id_list` включает все значения `bug_id` в каждой группе.

Другой недостаток этого решения в том, что оно не стандартизировано и другие РСУБД не поддерживают эту функцию. Некоторые РСУБД поддерживают специальные функции и специальные агрегатные функции. Например, решение для PostgreSQL выглядит так:

Groups/soln/group-concat-pgsql.sql

```
CREATE AGGREGATE GROUP_ARRAY (
  BASETYPE = ANYELEMENT,
     SFUNC = ARRAY_APPEND,
     STYPE = ANYARRAY,
  INITCOND = '{}'
);

SELECT product_id, MAX(date_reported) AS latest,
  ARRAY_TO_STRING(GROUP_ARRAY(bug_id), ',') AS bug_id_list
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

Некоторые РСУБД не поддерживают специальные функции, так что, возможно, придется написать хранимую процедуру для перебора негруппированного результата запроса и ручной конкатенации значений.

Используйте это решение, если предполагается, что дополнительный столбец будет содержать одно значение на группу, но при этом нарушает правило одного значения.

Использование проприетарных решений

MySQL содержит еще одну проприетарную функцию `ANY_VALUE()`, которая позволяет отправить запрос к столбцу, нарушающему правило одного значения.

Функция подавляет проверку неоднозначных столбцов, поэтому она возвращает значение столбца из произвольной строки в группе.

Groups/soln/extra-aggregate-any.sql

```
SELECT product_id, MAX(date_reported) AS latest,
  ANY_VALUE(assigned_to) AS any_developer
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

Эта функция может использоваться и в том случае, если столбец содержит только одно значение на группу, но MySQL не может вычислить функциональную зависимость. Также эту функцию можно применять, если вас просто не интересует, какое значение будет возвращено, важно лишь то, что это значение входит в соответствующую группу.

Запросы лучше делать логически предсказуемыми и однозначными, потому что результаты запросов, зависящих от нестандартного поведения, могут неожиданно изменяться.

Следуйте правилу одного значения, чтобы избежать неоднозначных результатов.

Мини-антипаттерн: портируемый SQL

Не все базы данных реализуют SQL одинаково. SQL представляет собой стандартизированный язык, для которого существует подробная спецификация, но языковая спецификация допускает различные «уровни» соответствия. Любая компания, выпускающая РСУБД на основе SQL, может принять решение о реализации подмножеств функциональных аспектов спецификации. Разные компании выбирают разные подмножества функциональности, описанной в языке.

Разработчики стараются писать «портируемый» код SQL, для чего они ограничиваются функциональностью, общей для всего ПО баз данных на основе SQL. Они хотят быть уверенными, что их код SQL будет работать даже при смене базы данных.

Это создает как минимум две проблемы.

- Во-первых, вы лишаетесь возможности использовать все проприетарные возможности любой РСУБД. Кроме стандартной функциональности SQL,

все компании-разработчики добавляют собственную расширенную функциональность. Некоторые из этих расширенных возможностей весьма полезны, хотя формально они и не являются частью стандартного языка SQL. Если ограничиваться только стандартным синтаксисом SQL, поддерживаемым всеми реализациями, вы не сможете использовать расширения.

- Во-вторых, вы все равно не добьетесь желаемой цели. Компании-разработчики даже стандартные средства SQL реализуют по-разному. Они слегка лукавят в вопросах соответствия спецификации или по-разному интерпретируют ее. Стандарт составлен довольно подробно, но он написан на английском языке, который не настолько точен, как программная реализация. В результате даже если ваши запросы будут написаны только стандартными средствами языка SQL, они могут по-другому работать в других РСУБД.

Несколько лет назад я разработал учебный курс по базам данных для фреймворка PHP. Предполагалось, что в нем будет создан единый интерфейс для поддержки кода SQL для шести РСУБД: MySQL, PostgreSQL, Microsoft SQL Server, IBM DB2, Oracle и SQLite. Как ни странно, даже на таком базовом уровне, как стандартные типы данных, ни один тип данных не был абсолютно одинаково реализован во всех шести РСУБД.

Эти проблемы сильно затрудняют задачу написания портируемого SQL. Даже если бы это было возможно, то вы не смогли бы использовать расширенные возможности любой конкретной реализации.

Пишите код с использованием паттерна проектирования «Адаптер» (см. *Design Patterns: Elements of Reusable Object-Oriented Software [GHJV95]*), чтобы иметь в наличии реализации для всех разновидностей баз данных SQL, которые потребуется поддерживать.

Генерирование случайных чисел — слишком важное дело, чтобы оставлять его на волю случая.

➤ *Роберт Р. Кавью (Robert R. Coveyou)*

ГЛАВА 16

Случайный выбор

Вы пишете веб-приложение, в котором выводится реклама. При каждом просмотре должна выбираться случайная реклама, чтобы у всех рекламодателей были равные шансы на показ, а читателям не надоедало постоянно видеть одну и ту же рекламу.

Первые несколько дней все идет хорошо, но приложение постепенно начинает подтормаживать. Через несколько недель люди начинают жаловаться, что ваш сайт слишком медленный. Вы понимаете, что дело не в психологии; вы измеряете реальный прирост времени загрузки страницы. Пользователи начинают терять интерес, и трафик снижается.

Руководствуясь опытом, вы сначала пытаетесь найти узкое место в производительности. Для этого вы используете профилировщик и тестовую версию базы данных с выборкой данных. Вы измеряете время загрузки веб-страницы, но как ни странно, с производительностью запросов SQL, используемых для заполнения страницы, проблем нет. Тем временем реальный сайт работает все медленнее и медленнее.

Наконец, вы понимаете, что база данных на реальном сайте намного больше выборки из тестов. Вы повторяете тесты для базы данных с размером, сходным с размером реальных данных, и обнаруживаете, что дело в запросе на выбор рекламы. С увеличением количества рекламы, из которой делается выбор, производительность запроса резко снижается. Определение виновника — важный первый шаг.

Как изменить структуру запроса, который выбирает случайную рекламу, пока сайт не потерял аудиторию и, как следствие, — спонсоров?

Цель: получить образец строки данных

Удивительно, насколько часто возникает необходимость в запросах SQL, которые возвращают случайный результат. Может показаться, что они противоречат

принципам повторяемости и детерминированного программирования. Однако иногда важно запрашивать выборку данных из большого набора. Вот несколько примеров:

- Вывод ротационного контента (например, рекламы или новостей).
- Аудит подмножества записей.
- Распределение входящих вызовов между свободными операторами.
- Генерирование тестовых данных.

Целесообразно запросить выборку из базы данных, а не загружать весь набор данных в приложение для получения случайного результата.

Цель — написать эффективный запрос SQL, который возвращает только случайную выборку данных. В математике и в информатике существует различие между полноценной случайностью и *псевдослучайностью*. На практике компьютеры могут генерировать только псевдослучайные значения.

Антипаттерн: случайная сортировка данных

Самый распространенный прием SQL для выбора случайной строки из запроса основан на случайной сортировке запроса и выборе первой строки из результата. Этот прием легко понятен и реализуем:

Random/anti/orderby-rand.sql

```
SELECT * FROM Bugs ORDER BY RAND() LIMIT 1;
```

Хотя это решение популярно, оно быстро проявляет свою слабость. Чтобы ее понять, сначала сравним его с традиционной сортировкой, когда мы сравниваем значения в столбце и упорядочиваем строки в соответствии с тем, какая запись содержит большее или меньшее значение в этом столбце. Такая сортировка воспроизводима, то есть она генерирует одинаковые результаты при многократном выполнении. К ней также полезно применить индекс, потому что индекс фактически представляет собой заранее отсортированный набор значений заданного столбца.

Random/anti/indexed-sort.sql

```
SELECT * FROM Bugs ORDER BY date_reported;
```

Если критерий сортировки представляет собой функцию, которая возвращает случайное значение для строки, ответ на вопрос, будет ли заданная строка больше или меньше другой, тоже будет случайным. Таким образом, порядок не связан со значениями в каждой строке. При такой сортировке порядок также будет различным. Пока все хорошо — именно такой результат нам и нужен.

Сортировка по недетерминированному выражению (`RAND()`) означает, что применять индексы к ней бесполезно. Не существует индекса со значениями, возвращаемыми случайной функцией. Именно поэтому они делаются случайными: результаты получаются разными и непредсказуемыми при каждом выборе.

И это создает проблемы с производительностью запросов, потому что использование индекса — один из лучших способов ускорения сортировки. Отказ от использования индекса приводит к тому, что итоговый набор запроса базе данных приходится сортировать «вручную». Это называется *сканированием таблицы* и часто представляет собой сохранение всего результата во временной таблице и сортировку его физической перестановкой строк. Сортировка со сканированием таблицы выполняется намного медленнее сортировки на базе индексов, и различия в производительности увеличиваются с увеличением размера набора данных.

Проблема незаметна при малом количестве записей, например, во время разработки и тестирования. К сожалению, объем данных в базе со временем увеличивается, и запрос не может нормально масштабироваться.

Как распознать антипаттерн

Прием, содержащийся в антипаттерне, весьма прост, и многие его используют, либо прочитав о нем в статье, либо придя к нему самостоятельно. Следующие фразы указывают на то, что ваши коллеги применяют этот антипаттерн:

- «SQL очень долго выводит случайную строку».

  Запрос на получение случайной выборки хорошо работал с простыми данными во время разработки и тестирования, но с увеличением объема реальных данных он начинает работать все медленнее. Никакие усилия по настройке сервера БД, индексированию или кэшированию не улучшат масштабирование.

- «Как выделить больше памяти приложению? Мне нужно загрузить все строки, чтобы выбрать одну случайную».

  Загружать все строки в приложение совершенно не обязательно; более того, это напрасная трата ресурсов, особенно если учесть, что базы данных часто разрастаются до размеров, с которыми память приложения просто не справится.

- «А вам не кажется, что некоторые элементы встречаются чаще, чем следует? Случайный выбор получается не очень случайным».

  Случайные числа не синхронизируются с пропусками в значениях первичного ключа в базе данных (см. «Выбор большего значения ключа», с. 210).

Допустимые применения антипаттерна

Неэффективное решение со случайной сортировкой приемлемо, если набор данных остается малым. Например, можно использовать случайный выбор для назначения конкретной ошибки разработчику. Можно с уверенностью предположить, что разработчиков никогда не будет столько, что вам понадобится механизм выбора случайного сотрудника с высокой степенью масштабируемости.

Другой прием — выбор случайного штата США из списка 50 штатов. Этот список имеет умеренный размер и вряд ли сильно увеличится на нашем веку.

Решение: не упорядочивать…

Метод случайной сортировки является примером запроса, выполняющего сканирование таблицы и затратную ручную сортировку. Проектируя решения в SQL, избегайте подобных неэффективных запросов. Не тратьте время на бесполезные поиски возможностей оптимизировать неоптимизируемое, а подойдите к решению с другой стороны. Чтобы получить случайную строку из результирующего набора, используйте один из альтернативных способов, описанных ниже.

Выбор случайного значения ключа между MIN и MAX

Один из приемов, позволяющих избежать сортировки таблицы, — выбор случайного значения между наименьшим и наибольшим значением первичного ключа.

Random/soln/rand-min-to-max.sql

```
SELECT MIN(bug_id), MAX(bug_id) INTO @min_bug_id, @max_bug_id FROM Bugs;

SELECT * FROM Bugs
WHERE bug_id = ROUND(RAND() * (@max_bug_id - @min_bug_id)) + @min_bug_id;
```

Решение предполагает, что значения первичного ключа непрерывны. Иначе говоря, между наименьшим и наибольшим значением нет неиспользуемых значений. Если между ними встречаются пропуски, то случайно выбранное значение может не соответствовать ни одной строке в таблице.

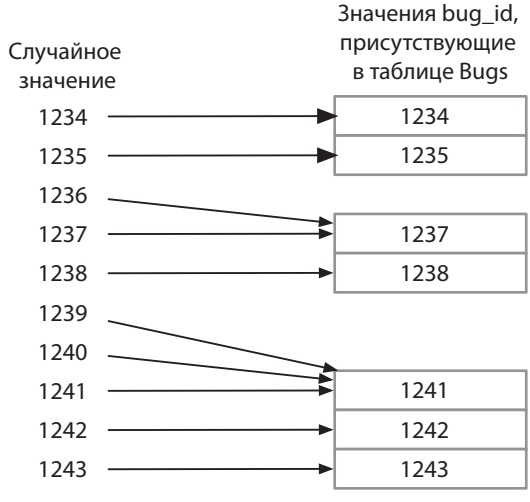
Выбор большего значения ключа

Это решение похоже на предыдущее, но если между наименьшим и наибольшим значением первичного ключа встречаются интервалы неиспользуемых значений, этот запрос получает первое найденное значение ключа, большее либо равное случайному значению.

Random/soln/rand-next-greater.sql

```
SELECT MIN(bug_id), MAX(bug_id) INTO @min_bug_id, @max_bug_id FROM Bugs;

SELECT * FROM Bugs
WHERE bug_id >= ROUND(RAND() * (@max_bug_id - @min_bug_id)) + @min_bug_id;
ORDER BY bug_id LIMIT 1;
```

Это решает проблему случайных чисел, не совпадающих ни с одним значением ключа, но зато значение ключа, следующее за интервалом неиспользуемых значений, будет выбираться чаще других. Все случайные значения выбираются с приблизительно равной частотой, но значения `bug_id` распределены неравномерно. Чем больше размер пропуска, предшествующего заданному значению первичного ключа, тем большее количество случайно выбранных значений будет с ним соотноситься.



Используйте это решение, если вам не важно, чтобы значения ключа выбирались с одинаковой частотой.

Получение списка всех значений ключа, случайный выбор одного значения

Также можно использовать код приложения, чтобы выбрать одно значение из первичных ключей результирующего набора. Затем из базы данных запрашивается полная строка с этим первичным ключом. Этот прием продемонстрирован в следующем коде Python:

Random/soln/rand-key-from-list.py

```
import mysql.connector
import random
```

```
cnx = mysql.connector.connect(user='scott', database='test')
cursor = cnx.cursor()

cursor.execute("SELECT bug_id FROM Bugs")
bug_ids = cursor.fetchall()
rand_bug_id = random.choice(bug_ids)[0]

cursor.execute("SELECT * FROM Bugs WHERE bug_id = %s", (rand_bug_id,))
for bug in cursor:
    print(bug)

cnx.commit()
```

Такое решение обходится без сортировки таблицы, и вероятности выбора всех значений ключа приблизительно равны, но у него есть и затраты:

- Выборка всех значений `bug_id` из базы данных может вернуть список огромного размера. Он даже может превысить ресурсы памяти приложения.
- Запрос должен выполняться дважды: один раз для получения списка первичных ключей и второй — для получения случайной строки. Для сложных и затратных запросов это проблема.

Используйте это решение, если вы выбираете случайную строку из простого запроса с результирующим набором умеренного размера. Решение хорошо подойдет для выбора из списка несмежных значений.

Выбор случайной строки по смещению

Еще один прием, который позволяет избежать проблем, присущих предыдущим вариантам, — подсчет строк в наборе данных и получение случайного числа от нуля до подсчитанного размера. Это число используется как смещение.

Random/soln/rand-limit-offset.py

```
import mysql.connector

cnx = mysql.connector.connect(user='scott', database='test')
cursor = cnx.cursor()

cursor.execute("SELECT ROUND(RAND() * (SELECT COUNT(*) FROM Bugs))")
for row in cursor:
    offset = int(row[0])
cursor.execute("SELECT * FROM Bugs LIMIT 1 OFFSET %s", (offset,))
for bug in cursor:
    print(bug)

cnx.commit()
```

В этом решении используется нестандартная конструкция `LIMIT`, поддерживаемая MySQL, PostgreSQL и SQLite. Следующий вариант использует стандартную оконную функцию `ROW_NUMBER()`:

Random/soln/rand-row-number.py

```
import mysql.connector

cnx = mysql.connector.connect(user='scott', database='test')
cursor = cnx.cursor()

cursor.execute("SELECT 1 + ROUND(RAND() * COUNT(*)) FROM Bugs")
for row in cursor:
    offset = row[0]

cursor.execute("""
    WITH NumberedBugs AS (
      SELECT *, ROW_NUMBER() OVER (ORDER BY bug_id) AS rownum FROM Bugs
    )
    SELECT * FROM NumberedBugs WHERE rownum = %s""", (offset,))
for bug in cursor:
    print(bug)

cnx.commit()
```

Используйте это решение, если задача не предполагает непрерывности значений ключа, но при этом все строки должны выбираться с равной вероятностью.

Проприетарные решения

Разные СУБД предлагают собственные решения для задач такого рода. Например, в Microsoft SQL Server 2005 добавлена секция `TABLESAMPLE`:

Random/soln/tablesample-sql2005.sql

```
SELECT * FROM Bugs TABLESAMPLE (1 ROWS);
```

В Oracle используется слегка отличающаяся секция `SAMPLE`, например, для возвращения 1 % строк таблицы:

Random/soln/sample-oracle.sql

```
SELECT * FROM (SELECT * FROM Bugs SAMPLE (1)
ORDER BY dbms_random.value) WHERE ROWNUM = 1;
```

Подробнее об этих или других решениях можно узнать из документации проприетарного решения, поддерживаемого вашей СУБД.

Сокращение, переработка, повторное использование

Возможно, вам понадобится использовать тот или иной вариант случайного выбора несколько раз. В примере с показом рекламы на веб-странице, описанном в начале этой главы, одно случайное объявление может показываться всем посетителям в течение пяти минут, после чего выбирается новая случайная реклама для следующего периода времени.

В приведенном ниже приложении Python Flask используется декоратор функции `@lru_cache`, который обеспечивает кэширование результата вызова функции. При следующем вызове код функции пропускается и просто возвращается кэшированная реклама.

Random/soln/rand-cached.py

```
import mysql.connector
from flask import Flask, Response, jsonify, request
from functools import lru_cache

app = Flask(__name__)

@app.route('/advert')
def advert():
    if request.args.get("reset"):
        get_random_advert.cache_clear()
    return jsonify(get_random_advert())

@lru_cache(maxsize=1)
def get_random_advert():
    cnx = mysql.connector.connect(user='scott', database='test')
    cursor = cnx.cursor()
    cursor.execute("SELECT * FROM Adverts ORDER BY RAND() LIMIT 1")
    columns = [col[0] for col in cursor.description]
    advert = dict(zip(columns, cursor.fetchone()))
    cnx.commit()
    return advert

if __name__ == '__main__':
    app.run()
```

Веб-страница проверяет, присутствует ли необязательный параметр запроса `reset`. В таком случае кэш очищается, а функция запускает запрос SQL для выбора новой случайной рекламы. Владелец веб-страницы запускает скрипт для сброса кэша через каждые пять минут. В этом случае только скрипт столкнется с замедлением загрузки страницы.

Некоторые запросы невозможно оптимизировать; выберите другое решение.

Мини-антипаттерн: запрос нескольких случайных строк

Решения из этой главы возвращают одну случайную строку данных из таблицы, но некоторым приложениям может понадобиться набор из нескольких случайно выбранных строк. Решения такой задачи оптимизировать сложнее.

Наивный вариант основан на возврате к затратному запросу, описанному в начале главы, но с увеличением `LIMIT`:

```
SELECT * FROM Bugs ORDER BY RAND() LIMIT 5;
```

Это простой запрос, но он обходится слишком дорого, если таблица содержит много строк. Альтернатива — использовать любое из решений, приведенных выше. Каждое из них возвращает одну случайную строку, но запрос нужно выполнять многократно, пока не будет получено необходимое количество строк. Выполнить хорошо оптимизированный запрос пять раз все равно лучше, чем выполнить плохо оптимизированный запрос один раз.

Однако у этого способа есть недостатки.

При многократном выборе одной случайной строки вы можете получить одну строку несколько раз. Если вам нужно получить определенное количество случайно выбранных строк, но без дубликатов, придется писать код для проверки результата и повторять запрос, пока вы не получите достаточное количество разных результатов.

Если таблица слишком мала, повторные попытки могут создать бесконечный цикл. Чтобы получить пять разных строк из таблицы, которая содержит всего четыре строки, ждать придется очень долго. Необходимо включить логику для проверки такого случая.

Проблема хорошо решается в наивном варианте, использующем `ORDER BY RAND() LIMIT 5`. Какое решение использовать — более производительное или требующее меньше кода — выбирать вам.

Некоторые, столкнувшись с какой-то проблемой, думают: «Я знаю, что делать, — я воспользуюсь регулярными выражениями». Теперь у них две проблемы.

➤ *Джейми Завински (Jamie Zavinski)*

ГЛАВА 17

Поисковая система для бедных

В 1995 году я работал в службе технической поддержки. В это время компании только начинали осваивать веб-технологии для предоставления информации пользователям. У меня были краткие документы, в которых описывались решения типичных вопросов, и я хотел разместить их в приложении базы знаний с веб-доступом.

С увеличением количества документов понадобилось организовать поиск, потому что клиентам не хотелось просматривать сотни статей в поисках ответов. Как вариант можно было разбить статьи на категории, но даже такие группы были слишком большими, и многие статьи принадлежали сразу к нескольким группам.

В самом гибком и простом интерфейсе клиент вводил произвольный набор слов, после чего приложение выводило список статей, в которых эти слова встречаются. Статье назначался больший вес, если в ней встречались более полные совпадения искомых слов. Кроме того, поисковая система должна была поддерживать поиск по разным формам слов. Например, поиск по слову `crash` также должен находить формы `crashed`, `crashes` и `crashing`. Конечно, поиск должен работать с растущей коллекцией документов достаточно быстро, чтобы им можно было пользоваться в веб-приложении.

Если это описание кажется вам чересчур подробным, ничего удивительного. Текстовый поиск в приложениях стал настолько популярным, что сейчас уже трудно вспомнить времена, когда он был недоступен. К сожалению, использовать SQL для поиска по ключевым словам и при этом обеспечить быстроту и точность решения на удивление непросто.

Цель: полнотекстовый поиск

В любом приложении, хранящем текстовые данные, возникает необходимость в поиске слов или отдельных фраз в тексте. В наше время в базах данных текст

хранится чаще, чем когда-либо, и пользователям требуется высокая скорость поиска. В веб-приложениях особенно актуальны высокопроизводительные и масштабируемые средства баз данных для поиска текста.

Один из фундаментальных принципов SQL (и реляционной теории, на основе которой создавался SQL) — *атомарность* значений столбцов. Иначе говоря, значение можно сравнить с другим значением, но при этом всегда сравнивается *полное* значение. Сравнение подстрок в SQL оказывается неэффективным или неточным.

Антипаттерн: предикаты сопоставления с шаблонами

SQL предоставляет предикаты для поиска по шаблону для сравнения строк, и обычно это первое, что выбирают разработчики при поиске по ключевым словам. Чаще всего используется предикат `LIKE`.

Предикат `LIKE` поддерживает универсальный символ `%`, который сопоставляется с произвольным количеством символов. Если поставить этот символ до и после ключевого слова, шаблон будет сопоставлен с любой строкой, содержащей это слово. Первый символ сопоставляется со всем текстом, предшествующим слову, а второй — с текстом, следующим после слова.

Search/anti/like.sql

```
SELECT * FROM Bugs WHERE description LIKE '%crash%';
```

Регулярные выражения также поддерживаются многими СУБД. Универсальные символы в этом случае не понадобятся, потому что обычно регулярные выражения используются для поиска совпадений внутри любых строк. Хотя в SQL-99 определяется предикат `SIMILAR TO` для применения регулярных выражений, многие СУБД на основе SQL продолжают использовать нестандартный синтаксис. Пример использования предиката регулярных выражений в MySQL:

Search/anti/regexp.sql

```
SELECT * FROM Bugs WHERE description REGEXP 'crash';
```

Самый главный недостаток операторов поиска по шаблону — низкая производительность. Они не могут использовать традиционные индексы, поэтому им приходится сканировать каждую строку в таблице. Так как поиск по шаблону в текстовом столбце является затратной операцией (по сравнению, например, с проверкой двух чисел на равенство), общие затраты на сканирование таблицы при таком поиске будут очень высокими.

Второй недостаток простого поиска по шаблону с `LIKE` или регулярных выражений — вероятность непредвиденных совпадений.

Search/anti/like-false-match.sql

```
SELECT * FROM Bugs WHERE description LIKE '%one%';
```

Этот пример находит совпадение в тексте, содержащем слово `one`, но он также найдет совпадение в строках `money`, `prone`, `lonely` и т. д. Поиск по шаблону, в котором ключевое слово заключено между пробелами, не найдет вхождения слова со знаками препинания или вхождения слова в начале или конце текста. Поддержка регулярных выражений в базе данных может включать специальные обозначения для *границы слова*, как в MySQL 8.0:

Search/anti/regexp-word.sql

```
SELECT * FROM Bugs WHERE description REGEXP '\\bone\\b';
```

С учетом проблем с производительностью и масштабируемостью, а также замысловатых манипуляций, которые приходится проделывать для предотвращения неактуальных совпадений, простой поиск по шаблону плохо подходит для реализации поиска по ключевым словам.

Как распознать антипаттерн

Вот некоторые типичные вопросы, указывающие на возможное применение антипаттерна «Поисковая система для бедных»:

- «Как вставить переменную между двумя универсальными символами в `LIKE`?»

  Вопрос обычно возникает у разработчиков, которым требуется поиск по шаблону в данных, введенных пользователем.
- «Как написать регулярное выражение для проверки того, что строка содержит несколько слов, что строка *не* содержит заданное слово или что строка содержит произвольную форму заданного слова?»

  Если проблема кажется слишком сложной, чтобы ее можно было решить с применением регулярного выражения, то скорее всего, так и есть.
- «После добавления новых документов в базу данных поиск на сайте стал слишком медленным. Что не так?»

  С ростом объема данных решение с антипаттерном демонстрирует плохую масштабируемость.

Допустимые применения антипаттерна

Выражения, приведенные в описании антипаттерна, являются валидными запросами SQL. Они применяются достаточно просто и прямолинейно, а это дорогого стоит.

Производительность часто важна, но некоторые запросы выполняются так редко, что вкладывать слишком много ресурсов на их оптимизацию не имеет смысла. Создание индекса для ускорения редко используемого запроса может быть так же затратно, как неэффективное выполнение запроса. Если запрос ситуативен по своей природе, созданный индекс может оказаться бесполезным.

Использовать операторы поиска по шаблону для сложных вопросов неудобно, но шаблоны для простых случаев позволят получить нужные результаты с минимальными усилиями.

Решение: правильный выбор инструмента для работы

Лучше использовать специализированные технологии поисковых систем вместо поиска по шаблону на уровне базового SQL.

В следующих подразделах описаны некоторые технологии, предоставляемые в виде встроенных расширений разных СУБД, и технологии независимых проектов. Также описывается решение, использующее стандартный SQL, но в среднем более эффективное, чем поиск подстрок по шаблону.

Расширения от компаний-разработчиков

Каждая серьезная СУБД предлагает собственное решение для полнотекстового поиска. Если вы работаете с одной СУБД (или хотите использовать специфическую функциональность конкретной СУБД), эти средства позволят наиболее эффективно реализовать высокопроизводительный текстовый поиск с максимальной интеграцией с запросами SQL.

Ниже кратко описываются решения для полнотекстового поиска в нескольких СУБД на основе SQL. Обратите внимание, что решения, предоставляемые разными СУБД, используют разный синтаксис и по своей функциональности отличаются от решений других СУБД. Изучение одного решения не поможет в изучении других. Кроме того, приведенные описания довольно сжатые. Решения полнотекстового поиска всех СУБД предлагают широкий выбор настроек; обращайтесь к актуальной документации для СУБД, которую вы используете.

Полнотекстовый индекс в MySQL

MySQL предоставляет поддержку простых полнотекстовых индексов. Полнотекстовые индексы могут определяться для столбцов с типом `CHAR`, `VARCHAR` или `TEXT`. В следующем примере определяется полнотекстовый индекс, включающий контент из столбцов `summary` и `description` таблицы ошибок:

Search/soln/mysql/alter-table.sql

```
ALTER TABLE Bugs ADD FULLTEXT INDEX bugfts (summary, description);
```

Используйте функцию `MATCH()` для поиска ключевых слов в индексируемом тексте. Столбцы, указанные в аргументах функции `MATCH()`, должны быть теми же столбцами и следовать в том же порядке, который использовался при создании индекса.

Search/soln/mysql/match.sql

```
SELECT * FROM Bugs WHERE MATCH(summary, description) AGAINST ('crash');
```

Также в шаблоне можно использовать запись с простыми логическими выражениями. Следующий пример показывает, как провести поиск строк, которые содержат слово crash, но не содержат слово save:

Search/soln/mysql/match-boolean.sql

```
SELECT * FROM Bugs WHERE MATCH(summary, description)
  AGAINST ('+crash -save' IN BOOLEAN MODE);
```

Индексирование текста в Oracle

В Oracle индексирование текста поддерживается, начиная с версии 8, выпущенной в 1997 году, когда этот инструмент был частью картриджа подключаемого расширения ConText. Технология несколько раз обновлялась, и сейчас эта функциональность интегрирована в продукт.

Возможности индексирования текста в Oracle сложные и богатые; ниже приводится их упрощенное описание:

- `CONTEXT`

  Создает индекс этого типа для одного текстового столбца. Используйте оператор `CONTAINS()` для поиска по индексу. Индекс не сохраняет целостность при изменениях в данных, если он не был определен с `PARAMETERS ('SYNC (ON COMMIT)')`.

  Search/soln/oracle/create-index.sql

  ```
  CREATE INDEX BugsText ON Bugs(summary) INDEXTYPE IS CTXSYS.CONTEXT;

  SELECT * FROM Bugs WHERE CONTAINS(summary, 'crash') > 0;
  ```

- `CTXCAT`

  Этот тип индекса специализирован для коротких текстовых фрагментов наподобие тех, которые используются в электронных каталогах, наряду со структурированными столбцами из той же таблицы. Индекс остается последовательным при обновлении индексированных данных в транзакциях.

Search/soln/oracle/ctxcat-create.sql

```
CTX_DDL.CREATE_INDEX_SET('BugsCatalogSet');
CTX_DDL.ADD_INDEX('BugsCatalogSet', 'status');
CTX_DDL.ADD_INDEX('BugsCatalogSet', 'priority');

CREATE INDEX BugsCatalog ON Bugs(summary) INDEXTYPE IS CTXSYS.CTXCAT
  PARAMETERS('BugsCatalogSet');
```

Оператор `CATSEARCH()` получает два аргумента, определяющих поиск в текстовом столбце и структурированном множестве столбцов соответственно.

Search/soln/oracle/ctxcat-search.sql

```
SELECT * FROM Bugs
WHERE CATSEARCH(summary, '(crash save)', 'status = "NEW"') > 0;
```

- `CTXXPATH`

  Этот тип индекса специализирован на поиске в документах XML с оператором `existsNode()`.

Search/soln/oracle/ctxxpath.sql

```
CREATE INDEX BugTestXml ON Bugs(testoutput) INDEXTYPE IS CTXSYS.CTXXPATH;

SELECT * FROM Bugs
WHERE testoutput.existsNode('/testsuite/test[@status="fail"]') > 0;
```

- `CTXRULE`

  Предположим, что в базе данных хранится большая коллекция документов и вам требуется классифицировать их по содержанию.

  С индексом `CTXRULE` можно разрабатывать правила анализа документов и вывода их классификации. Также можно предоставить набор документов и свои пожелания относительно их классификации, чтобы Oracle разработал правила, которые должны применяться к остальным документам из коллекции. Процесс даже можно полностью автоматизировать, поручить Oracle проанализировать коллекцию документов и создать для нее набор правил и классификацию.

  Примеры использования индексов `CTXRULE` выходят за рамки этой книги.

В Oracle 12 появился синтаксис создания разновидности текстового индекса для данных JSON. Предположим, что таблица содержит столбец JSON с именем `properties` и в этом столбце хранятся динамические атрибуты.

Search/soln/oracle/create-index-json.sql

```
CREATE SEARCH INDEX BugsJson ON Bugs(properties) FOR JSON;

SELECT * FROM Bugs
WHERE json_textcontains(properties, '$.summary', 'crash');
```

Полнотекстовый поиск в Microsoft SQL Server

SQL Server 2000 и более поздних версий поддерживает полнотекстовый поиск со сложными настройками конфигурации для языков, тезаурусом и автоматической синхронизацией изменений данных. SQL Server предоставляет набор хранимых процедур для создания полнотекстовых индексов, и можно использовать в запросах операторы `CONTAINS()` для применения полнотекстового индекса.

Чтобы реализовать уже упоминавшийся пример с поиском ошибок, содержащий слово crash, сначала следует включить полнотекстовый поиск и определить каталог в базе данных:

Search/soln/microsoft/catalog.sql

```
CREATE FULLTEXT CATALOG BugsCatalog;
```

Затем определите полнотекстовый индекс для таблицы `Bugs`, добавьте столбцы в индекс и активируйте индекс:

Search/soln/microsoft/create-index.sql

```
CREATE FULLTEXT INDEX ON Bugs(summary, description)
  KEY INDEX bug_id ON BugsCatalog
  WITH CHANGE_TRACKING AUTO;
```

Наконец, выполните запрос с оператором `CONTAINS()` или оператором `FREETEXT()`:

Search/soln/microsoft/search.sql

```
SELECT * FROM Bugs WHERE CONTAINS(summary, 'crash');

SELECT * FROM Bugs WHERE FREETEXT(summary, 'crash bug error');
```

Поиск текста в PostgreSQL

PostgreSQL предоставляет нетривиальный, обладающий широкими возможностями конфигурации механизм преобразования текста в коллекции лексических элементов с поддержкой поиска, а также поиск совпадений шаблонов среди таких элементов. Поддерживать синхронизацию текстовых индексов в PostgreSQL 12 стало проще благодаря возможности использования генерируемых столбцов.

Search/soln/postgresql/create-table.sql

```
CREATE TABLE Bugs (
  bug_id SERIAL PRIMARY KEY,
  summary VARCHAR(80),
  description TEXT,
  ts_bug_text TSVECTOR GENERATED ALWAYS AS (to_tsvector('english',
      COALESCE(summary, '') || COALESCE(description, ''))) STORED
  -- другие столбцы
);
```

Также следует создать *обобщенный инвертированный индекс*, или GIN (Generalized Inverted Index), для столбца `TSVECTOR`:

Search/soln/postgresql/create-index.sql

```
CREATE INDEX bugs_ts ON Bugs USING GIN(ts_bug_text);
```

После этого можно использовать оператор PostgreSQL `@@` для эффективного поиска на основе полнотекстового индекса:

Search/soln/postgresql/search.sql

```
SELECT * FROM Bugs WHERE ts_bug_text @@ to_tsquery('crash');
```

Существует много других вариантов настройки контента с поддержкой поиска, поисковых запросов и результатов поиска.

Полнотекстовый поиск (FTS) в SQLite

Стандартные таблицы в SQLite не поддерживают эффективный полнотекстовый поиск, но можно воспользоваться расширением для SQLite и сохранить текст в *виртуальной таблице*, предназначенной для поиска текста. Реализация этого расширения изменялась несколько раз. FTS1 и FTS2 считаются устаревшими. Используйте FTS3 в SQLite 3.5.0 и выше или FTS4 в SQLite 3.7.4 и выше.

Расширения FTS обычно не включаются по умолчанию в сборку SQLite, поэтому вам придется построить ее с включенными расширениями FTS. Включение FTS3 неявно включает FTS4. Добавьте следующие варианты при выполнении `configure`.

Search/soln/sqlite/configure

```
CPPFLAGS="-DSQLITE_ENABLE_FTS3 -DSQLITE_ENABLE_FTS3_PARENTHESIS" ./configure
```

Когда в вашем распоряжении появится версия SQLite с включенной поддержкой FTS, вы сможете создать виртуальную таблицу для текста с возможностью поиска. Любые типы данных, ограничения или другие настройки столбцов игнорируются.

Search/soln/sqlite/create-table.sql

```
CREATE VIRTUAL TABLE BugsText USING fts4(summary, description);
```

Если вы индексируете текст из другой таблицы (как в примере с использованием таблицы `Bugs`), данные необходимо скопировать в виртуальную таблицу. Виртуальная таблица FTS всегда содержит столбец первичного ключа с именем `docid`, чтобы сопоставлять строки данных со строками исходной таблицы.

Search/soln/sqlite/insert.sql

```
INSERT INTO BugsText (docid, summary, description)
  SELECT bug_id, summary, description FROM Bugs;
```

Теперь вы можете обратиться с запросом к виртуальной таблице FTS `BugsText` с использованием эффективного предиката полнотекстового поиска `MATCH`, а также связать найденные строки с исходной таблицей `Bugs`. Если использовать имя таблицы FTS в качестве псевдостолбца, поиск совпадений шаблона будет осуществляться в любом столбце.

Search/soln/sqlite/search.sql

```
SELECT b.* FROM BugsText t JOIN Bugs b ON (t.docid = b.bug_id)
WHERE BugsText MATCH 'crash';
```

В шаблонах также реализована ограниченная поддержка булевых выражений.

Search/soln/sqlite/search-boolean.sql

```
SELECT * FROM BugsText WHERE BugsText MATCH 'crash -save';
```

Сторонние поисковые системы

Если вам нужно, чтобы поиск в тексте проводился одинаково независимо от того, какую СУБД вы используете, вам понадобится поисковая система, работающая независимо от базы данных SQL. В этом разделе кратко описаны два таких продукта, Sphinx Search и Apache Lucene.

Sphinx Search

Sphinx Search[1] — технология поисковой системы с открытым исходным кодом, хорошо интегрирующаяся с MySQL и PostgreSQL.

Индексирование и поиск быстро выполняются в Sphinx Search; также поддерживаются распределенные запросы. Это хороший вариант для крупномасштабных поисковых приложений с данными, которые относительно редко обновляются.

Sphinx Search может использоваться для объявления поискового индекса с множественными полями, для чего задействуется файл конфигурации sphinx.conf. Индекс включает поля, в которых может осуществляться поиск, а также другие, необязательные поля атрибутов.

Search/soln/sphinx/sphinx.conf

```
searchd
{
```

[1] https://sphinxsearch.com/

```
    log = ./data/searchd.log
    pid_file = ./data/searchd.pid
}

index bugs
{
    type = rt
    path = ./data/bugs
    rt_field = summary
    rt_field = description
    stored_fields = summary, description
}
```

После объявления этой конфигурации в sphinx.conf любой клиент, который может подключиться к экземпляру MySQL, сможет подключиться и к Sphinx Search, поскольку Sphinx Search реализует сервис, имитирующий клиент-серверный протокол MySQL.

Search/soln/sphinx/mysql-client.sh

```
mysql -h 127.0.0.1 -P 9306
```

В индекс можно вставлять данные, как если бы он являлся таблицей SQL:

Search/soln/sphinx/insert.sql

```
INSERT INTO Bugs (id, summary, description)
  VALUES (1234, 'crash when I save', '...');
```

Затем в индексе можно выполнять поиск с использованием ограниченного языка SQL:

Search/soln/sphinx/search.sql

```
SELECT * FROM Bugs WHERE MATCH('crash');
```

У Sphinx Search также имеется процесс-демон и API, через который можно запускать поиск из любого языка программирования, имеющего клиентский интерфейс для MySQL.

Apache Lucene

Lucene[1] — развитая поисковая система для приложений Java. Lucene строит индекс в своем закрытом формате для коллекций текстовых документов. Индекс Lucene не синхронизируется с исходными данными, которые он индексирует. При вставке, удалении или обновлении строк в базе данных необходимо соответственно изменить индекс Lucene.

[1] https://lucene.apache.org/

Поисковая система Lucene немного напоминает двигатель автомобиля; чтобы она работала эффективно, ей необходимо технологическое сопровождение. Lucene не читает коллекции данных из БД SQL напрямую; необходимо записывать документы в индекс Lucene. Например, можно запустить запрос SQL и для каждой строки результата создать один документ Lucene и сохранить его в индексе Lucene. Также Lucene можно использовать через его Java API.

К счастью, Apache также предлагает сопутствующий проект Solr[1]. Solr — сервер, предоставляющий шлюз к индексу Lucene. Вы можете добавлять документы в Solr и отправлять поисковые запросы с использованием REST-подобного интерфейса, который совместим с любым языком программирования.

Solr предоставляет средства импортирования данных в формате XML или CSV, а также для индексирования документов в таких форматах, как Microsoft Word, PDF или других закрытых форматах. Также можно приказать Solr подключиться к базе данных SQL, выполнить запрос и проиндексировать результаты запроса при помощи утилиты Solr `DataImportHandler`.

Elasticsearch и OpenSearch

Как и Apache Solr, проекты Elasticsearch[2] и OpenSearch[3] используют ядро Apache Lucene.

Elasticsearch содержит много дополнительных средств аналитики и визуализации. Система интегрируется с Logstash, инструментом для импортирования потоков данных в поисковое ядро, и Kibana, графическим интерфейсом для поиска данных, хранящихся в Elasticsearch. Эти продукты созданы компанией Elastic, Inc. и имеют общее название «стек ELK». Изначально система разрабатывалась как продукт с открытым кодом, но в 2021 году Elastic изменила лицензию для будущих версий продуктов стека ELK, чтобы стимулировать пользователей к использованию только собственного сервиса Elastic.

В ответ компания Amazon создала собственные ответвления технологий Elasticsearch 7.10.2 и Kibana 7.10.2 — последние версии, доступные на условиях лицензии Apache с открытым исходным кодом.

Продукт Amazon называется OpenSearch. Для импортирования данных в индекс OpenSearch можно использовать Logstash или другие сервисы потоков данных Amazon.

---

[1] https://lucene.apache.org/solr/

[2] https://www.elastic.co/elasticsearch/

[3] https://aws.amazon.com/elasticsearch-service/the-elk-stack/what-is-opensearch/

Несомненно, как Elastic, так и Amazon продолжат разработку соответствующих продуктов. Вероятно, будущие версии продолжат расходиться с добавлением новой функциональности и в конечном итоге станут несовместимыми. Это естественный результат ветвления технологий. Если вам нужно выбрать один из этих продкутов, попробуйте оба, оцените, чья функциональность лучше подходит для ваших целей, и выберите оптимальное решение.

Самодельные решения

Если вы не хотите использовать нестандартную поисковую функциональность или устанавливать независимую поисковую систему, вам необходимо эффективное, не зависящее от конкретной базы данных решение для поиска по тексту. В этом разделе описано решение, называемое *инвертированным индексом*. По сути, инвертированный индекс представляет собой список всех слов, которые пользователь может искать в тексте. Во многих отношениях «многие ко многим» индекс связывает эти слова с текстовыми элементами, содержащими соответствующее слово. Так, слово crash может встречаться во многих ошибках, а каждая ошибка может описываться многими другими ключевыми словами. В этом разделе показано, как спроектировать инвертированный индекс.

Начните с определения таблицы `Keywords` со списком ключевых слов, которые будет искать пользователь, и задайте таблицу пересечений `BugsKeywords` для создания отношения «многие ко многим»:

Search/soln/inverted-index/create-table.sql

```
CREATE TABLE Keywords (
  keyword_id  SERIAL PRIMARY KEY,
  keyword     VARCHAR(40) NOT NULL,
  UNIQUE KEY (keyword)
);

CREATE TABLE BugsKeywords (
  keyword_id  BIGINT UNSIGNED NOT NULL,
  bug_id      BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (keyword_id, bug_id),
  FOREIGN KEY (keyword_id) REFERENCES Keywords(keyword_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

Затем добавьте в `BugsKeywords` строку для каждого ключевого слова, которое соответствует тексту описания заданной ошибки. Запрос поиска подстрок определяет эти совпадения при помощи `LIKE` или регулярных выражений. Это решение будет таким же затратным, как наивный метод поиска, описанный в разделе «Антипаттерн», но поиск нужно будет провести только один раз. После сохранения результата в таблице пересечений весь последующий поиск по тому же ключевому слову будет выполняться намного быстрее.

Затем напишите хранимую процедуру MySQL для упрощения поиска заданного ключевого слова. Если поиск уже проводился, запрос будет выполняться быстрее, потому что строки `BugsKeywords` представляют собой список документов, содержащих ключевое слово. Если еще никто не искал это ключевое слово, придется проводить поиск по коллекции текстовых элементов методом брутфорса.

Search/soln/inverted-index/search-proc.sql

```
CREATE PROCEDURE BugsSearch(IN p_keyword VARCHAR(40))
BEGIN
    DECLARE v_keyword_id BIGINT UNSIGNED;

    SELECT MAX(keyword_id) INTO v_keyword_id FROM Keywords
❶   WHERE keyword = p_keyword;

    IF (v_keyword_id IS NULL) THEN
❷     INSERT INTO Keywords (keyword) VALUES (p_keyword);
❸     SELECT LAST_INSERT_ID() INTO v_keyword_id;
      INSERT INTO BugsKeywords (bug_id, keyword_id)
        SELECT bug_id, v_keyword_id FROM Bugs
        WHERE summary REGEXP CONCAT('\\b', p_keyword, '\\b')
❹         OR description REGEXP CONCAT('\\b', p_keyword, '\\b');
    END IF;

    SELECT b.* FROM Bugs b
    JOIN BugsKeywords k USING (bug_id)
❺   WHERE k.keyword_id = v_keyword_id;
END
```

❶ Поиск по ключевому слову, заданному пользователем. Возвращается либо целочисленный первичный ключ из `Keywords.keyword_id`, либо `null`, если слово ранее не встречалось.

❷ Если слово не найдено, оно вставляется как новое слово.

❸ Запрос значения первичного ключа, сгенерированного в `Keywords`.

❹ Таблица пересечений заполняется поиском в `Bugs` строк, содержащих новое ключевое слово.

❺ Наконец, из `Bugs` запрашиваются полные строки с соответствующим `keyword_id` независимо от того, было ли ключевое слово найдено или его пришлось вставлять как новый элемент.

Теперь можно вызвать эту хранимую процедуру и передать нужное ключевое слово. Процедура возвращает набор сопоставленных ошибок независимо от того, пришлось ли ей вычислить подходящие ошибки и внести в таблицу пересечений данные нового ключевого слова или же просто воспользоваться результатом более раннего поиска.

Search/soln/inverted-index/search-proc.sql

```
CALL BugsSearch('crash');
```

В этом решении присутствует еще один компонент — триггер для заполнения таблицы пересечений при вставке каждой новой ошибки. Если вам нужна возможность редактирования описаний ошибок, вы также можете написать триггер для повторного анализа текста и добавления (или удаления) строк таблицы `BugsKeywords`.

Search/soln/inverted-index/trigger.sql

```
CREATE TRIGGER Bugs_Insert AFTER INSERT ON Bugs
FOR EACH ROW
BEGIN
  INSERT INTO BugsKeywords (bug_id, keyword_id)
    SELECT NEW.bug_id, k.keyword_id FROM Keywords k
    WHERE NEW.description REGEXP CONCAT('\\b', k.keyword, '\\b')
      OR NEW.summary REGEXP CONCAT('\\b', k.keyword, '\\b');
END
```

Список ключевых слов заполняется естественным образом, когда пользователи выполняют поиск, так что вам не придется заполнять список всеми ключевыми словами, встречающимися в статьях базы знаний. С другой стороны, если вы можете предположить, какие слова будут чаще всего искать пользователи, есть смысл провести поиск заранее, чтобы уменьшить затраты на первый поиск каждого ключевого слова и избавить пользователей от них.

Я применил инвертированный индекс к своему приложению базы знаний, описанному в начале главы. Я также дополнил таблицу `Keywords` столбцом `num_searches`. Значение этого столбца увеличивалось каждый раз, когда пользователь искал заданное ключевое слово, чтобы я мог отслеживать наиболее популярные сценарии поиска.

Сегодня в вашем распоряжении имеются разнообразные средства индексирования текста: функциональность индексирования встраивается либо в базу данных SQL, которую вы используете, либо в один из специализированных продуктов. Не существует очевидно лучшего решения для всех проектов, так что вам придется попробовать несколько вариантов и оценить их сильные и слабые стороны, прежде чем выбрать подходящее под ваши потребности.

SQL рассматривает столбец как атомарное значение. Если вам потребуется оптимизировать поиск подстроки, придется использовать расширение для SQL или дополнительную технологию.

Entia non sunt multiplicanda praeter necessitatem
(«Не следует множить сущности без необходимости»).

➤ *Ирландский философ-францисканец Джон Панч (John Punch), 1639*

ГЛАВА 18

Спагетти-запросы

Ваш руководитель разговаривает по телефону со своим руководителем. Он подзывает вас взмахом руки, прикрывает трубку рукой и шепчет: «Сейчас проходит обсуждение бюджета, и нам сократят штат, если мы не сможем предоставить вице-президенту статистику, доказывающую, что все наши сотрудники загружены. Мне нужно знать, сколько у нас продуктов в работе, сколько разработчиков занималось исправлением ошибок, среднее количество исправленных ошибок на разработчика и количество исправленных ошибок, о которых сообщили клиенты. Немедленно!»

Вы открываете инструмент SQL и начинаете писать код. Вам нужны все ответы сразу, поэтому вы создаете один сложный запрос, надеясь свести к минимуму дубликаты и быстрее получить результат.

Spaghetti-Query/anti/complex-report.sql

```
SELECT COUNT(bp.product_id) AS how_many_products,
  COUNT(dev.account_id) AS how_many_developers,
  COUNT(b.bug_id)/COUNT(dev.account_id) AS avg_bugs_per_developer,
  COUNT(cust.account_id) AS how_many_customers
FROM Bugs b JOIN BugsProducts bp ON (b.bug_id = bp.bug_id)
JOIN Accounts dev ON (b.assigned_to = dev.account_id)
JOIN Accounts cust ON (b.reported_by = cust.account_id)
WHERE cust.email NOT LIKE '%@example.com'
GROUP BY bp.product_id;
```

Вы получаете числа, но с ними что-то не то. Откуда взялись десятки продуктов? Как среднее количество исправленных ошибок может быть точно равно 1.0? И вашему боссу нужно не количество клиентов, а количество ошибок, о которых сообщали пользователи. Почему все числа абсолютно неправдоподобны? Запрос будет намного сложнее, чем вы думали.

Начальник вешает трубку. «Неважно, — вздыхает он. — Уже слишком поздно. Давай освобождать столы».

Цель: сокращение количества запросов SQL

Многие проблемы разработчиков SQL часто начинаются с вопроса: «А нельзя ли это сделать в одном запросе?». Его задают практически для каждой задачи. Разработчиков учили, что один запрос SQL слишком сложен и дорого обходится, и они делают вывод, что два запроса SQL вдвое хуже. Варианты более чем с двумя запросами SQL обычно вообще не рассматриваются.

Разработчики не могут упростить задачи, но хотят упростить их решение. Они используют такие термины, как «элегантный» или «эффективный», и думают, что этих качеств можно добиться, уместив решение задачи в один запрос.

Антипаттерн: решение сложной задачи за один шаг

Язык SQL чрезвычайно выразителен. Одним запросом или инструкцией можно сделать очень много. Однако это вовсе не означает, что любую задачу нужно решать в одной строке кода (и что вообще стóит так делать). Стремитесь ли вы к этому, когда используете любой другой язык программирования? Наверное, нет.

Непредусмотренные произведения

Одно из самых частых следствий получения всех результатов в одном запросе — *декартово произведение*. Оно получается, когда две таблицы в запросе не имеют условия, ограничивающего отношения между ними. Без такого ограничения при соединении двух таблиц создаются пары из одной строки первой таблицы с *каждой* строкой второй таблицы. Каждая пара становится строкой результирующего набора, и в итоге вы получаете намного больше строк, чем ожидалось. Допустим, вам требуется запросить количество тегов, используемых для пометки заданной ошибки, а также продуктов, в которых возникла эта ошибка. Некоторые разработчики составляют запрос следующего вида:

Spaghetti-Query/anti/cartesian.sql

```
SELECT b.bug_id,
  COUNT(t.tag) AS count_tags,
  COUNT(bp.product_id) AS count_products
FROM Bugs b
LEFT OUTER JOIN Tags t ON (t.bug_id = b.bug_id)
LEFT OUTER JOIN BugsProducts bp ON (bp.bug_id = b.bug_id)
WHERE b.bug_id = 1234
GROUP BY b.bug_id;
```

С учетом того, что вам известно о вашей базе данных, результат сразу же выглядит неправдоподобным.

bug_id	count_tags	count_products
1234	8	8

Как количество продуктов может быть равно 8, если вы знаете, что сейчас в базе данных всего 3 разных продукта? Довольно странно и то, что за ошибкой закреплены 8 тегов. И как насчет совпадения количества продуктов и тегов — 8?

Коллега предлагает отфильтровать результаты по уникальным значениям, прежде чем подсчитывать их. Результаты больше похожи на то, что вы ожидали.

Spaghetti-Query/anti/cartesian-distinct.sql

```
SELECT b.bug_id,
  COUNT(DISTINCT t.tag) AS count_tags,
  COUNT(DISTINCT bp.product_id) AS count_products
FROM Bugs b
LEFT OUTER JOIN Tags t ON (t.bug_id = b.bug_id)
LEFT OUTER JOIN BugsProducts bp ON (bp.bug_id = b.bug_id)
WHERE b.bug_id = 1234
GROUP BY b.bug_id;
```

bug_id	count_tags	count_products
1234	4	2

Подсчет уникальных значений не будет работать с другими запросами, если ожидается, что в данных будет несколько совпадений с одним значением. Лучше понять, из-за чего появились лишние строки, и решить проблему. Можно попробовать удалить `GROUP BY` из запроса и посмотреть, как выглядит итоговый набор без группировки.

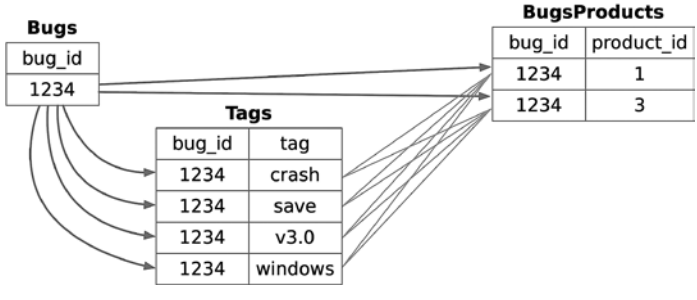
Spaghetti-Query/anti/cartesian-no-group.sql

```
SELECT b.bug_id, t.tag, bp.product_id
FROM Bugs b
LEFT OUTER JOIN Tags t ON (t.bug_id = b.bug_id)
LEFT OUTER JOIN BugsProducts bp ON (bp.bug_id = b.bug_id)
WHERE b.bug_id = 1234;
```

bug_id	tag	product_id
1234	crash	1
1234	crash	3
1234	save	1
1234	save	3
1234	v3.0	1
1234	v3.0	3

bug_id	tag	product_id
1234	windows	1
1234	windows	3

И хотя кажется, что запрос правильно соединяет `Bugs` и `Tags`, а также `Bugs` и `BugsProducts`, в запросе «прячется» декартово произведение, потому что к `Tags` и `BugsProducts` не применяется условие соединения. Другими словами, результаты умножаются, потому что каждая строка в `Tags` сопоставляется с *каждой* строкой в `BugsProducts`.



При попытке написать запрос двойного назначения легко случайно создать декартово произведение. Если в том же запросе снова выполнить другое несвязанное соединение без условия, ограничивающего результаты, то общее количество строк снова увеличится с созданием очередного декартова произведения.

А если этого недостаточно…

Кроме риска получить неверные результаты, важно учитывать, что такие запросы попросту сложно писать, сложно изменять и сложно отлаживать. Готовьтесь, что вам придется постоянно вносить изменения в приложения базы данных. Руководству нужны более сложные отчеты и больше полей в пользовательском интерфейсе. Изменение заковыристых монолитных запросов SQL обходится дороже и отнимает больше времени. Ваше время тоже обходится недешево — как для вас, так и для вашего проекта.

Не забывайте и о времени выполнения. Подробный запрос SQL, в котором используется множество соединений, коррелирующих подзапросов и других операций, ядру SQL будет труднее оптимизировать и выполнить быстрее, чем более простой запрос. Разработчики считают, что чем меньше в приложении запросов SQL, тем лучше его производительность. Возможно, это так, но только если запросы SQL одинаковые по степени сложности. Затраты на один мегазапрос могут расти по экспоненте, а использование нескольких простых запросов может оказаться более экономичным.

Как распознать антипаттерн

Следующие фразы участников проекта могут указывать на присутствие антипаттерна «Спагетти-запрос»:

- «Почему все суммы и счетчики запредельно велики?»

  Непредвиденное декартово произведение двух соединенных наборов данных привело к созданию множества строк.

- «Я весь день писал убойный запрос SQL».

  Язык SQL не так уж сложен, правда. Если вы слишком долго возитесь с одним запросом, вам стоит пересмотреть свое решение. Этот совет справедлив и для других языков программирования. Если задача кажется слишком сложной, разбейте ее на несколько более простых. Сделайте шаг назад и снова подумайте над предположениями, заложенными в основу архитектуры приложения. Может, другой алгоритм или другая модель данных упростят решение?

- «Мы не сможем ничего сделать с отчетом, потому что придется слишком долго разбираться, как переписать запрос SQL».

  Разработчик, написавший запрос, всегда отвечает за его обслуживание, даже если его переводят на другой проект. И этим разработчиком можете оказаться вы, так что не пишите слишком сложный код SQL, который никто не сможет изменить!

- «Попробуем добавить в запрос еще одну секцию `DISTINCT`».

  Чтобы компенсировать размножение строк в декартовом произведении, разработчики исключают дубликаты при помощи ключевого слова `DISTINCT` как модификатора запроса или агрегатной функции. Это скрывает присутствие неверно сформулированного запроса, но создает дополнительную работу для РСУБД, которой приходится генерировать промежуточный результирующий набор только для сортировки и устранения дубликатов.

Другим признаком «Спагетти-запроса» может быть слишком долгое время выполнения. Низкая производительность может свидетельствовать и о других проблемах, но при анализе таких запросов подумайте, не пытаетесь ли вы сделать слишком много в одной инструкции SQL.

Допустимые применения антипаттерна

Самая частая причина, по которой обычно выполняют сложную задачу с одним запросом, заключается в использовании программного фреймворка или библиотеки визуальных компонентов, которая подключается к источнику данных

и представляет данные в приложении. Простые средства бизнес-аналитики и визуализации данных также относятся к этой категории, хотя более сложные программные продукты бизнес-аналитики могут объединять результаты от нескольких источников данных.

Компонент или программа визуализации, которая допускает в качестве источника данных один запрос SQL, могут быть более простыми в использовании, но для них приходится создавать монолитные запросы, чтобы собрать все данные в отчете. При использовании таких приложений иногда приходится писать запросы SQL более сложные, чем при написании кода для обработки результирующего набора.

Если требования приложения визуализации данных слишком сложны, чтобы их можно было удовлетворить одним запросом SQL, возможно, лучше создать несколько отчетов. Если вашему руководству это не нравится, напомните ему о связи сложности отчета и затрат времени на его создание.

Иногда требуется получить сложный результат в одном запросе, потому что вам нужно объединить все результаты в определенном порядке. Задать порядок сортировки в запросе SQL легко. Обычно эффективнее поручить эту работу базе данных, чтобы писать в приложении меньше кода для сортировки результатов нескольких запросов.

Бывает ли польза от декартова произведения?

Почему же SQL допускает соединения без ограничивающих условий? Конечно, было бы лучше запретить декартовы произведения, если они создают столько проблем.

SQL поддерживает декартовы произведения в синтаксисе CROSS JOIN. Соединения такого типа могут намеренно применяться для генерирования всех возможных комбинаций двух наборов строк.

Представьте, что вам нужно быстро сгенерировать множество целых чисел от 0 до 99 на SQL. Для этого можно создать небольшую таблицу из 10 строк, содержащую целые числа от 0 до 9. Затем сгенерировать запрос с декартовым произведением, который соединяет таблицу с ней самой. Каждое число в таблице соединяется со всеми десятью числами, и получается результирующий набор из 10×10 строк. Используйте выражение, которое прибавляет число из первой таблицы к числу из второй таблицы, умноженному на 10. В результате вы получите 100 строк с числами от 0 до 90+9, то есть 99.

Spaghetti-Query/soln/cartesian-good.sql

```
CREATE TABLE integers (
  num INT PRIMARY KEY
);

INSERT INTO integers (num) VALUES (0), (1), (2), (3), (4), (5), (6), (7),
(8), (9);

SELECT 10*digit10.num + digit1.num AS num
FROM integers AS digit1
CROSS JOIN integers AS digit10;
```

Чтобы получить числа от 0 до 999, примените еще одно перекрестное соединение с той же таблицей. Используйте числа этой таблицы для определения сотенного разряда в значениях, возвращаемых выражением.

Spaghetti-Query/soln/cartesian-good.sql

```
SELECT 100*digit100.num + 10*digit10.num + digit1.num AS num
FROM integers AS digit1
CROSS JOIN integers AS digit10
CROSS JOIN integers AS digit100;
```

И это всего лишь один пример. У CROSS JOIN существуют и другие полезные применения. Они встречаются нечасто, но когда такая необходимость возникнет, будет полезно иметь ее поддержку на уровне синтаксиса SQL.

Решение: разделяй и властвуй

Цитата в начале главы по духу близка известному *принципу экономии*:

> **Принцип экономии**
>
> Если существуют две конкурирующие теории, которые выдают абсолютно одинаковые прогнозы, более простая теория будет лучшей.

В контексте SQL это означает, что если нужно выбрать между двумя запросами, выдающими одинаковые результирующие наборы, выбирайте более простой вариант. Помните об этом, исправляя применение этого антипаттерна.

Шаг за шагом

Если вы не видите условия логического соединения между таблицами, попавшими в непреднамеренное декартово произведение, это может объяснять-

ся тем, что такого условия попросту нет. Чтобы избежать декартова произведения, следует разбить «Спагетти-запрос» на несколько более простых запросов.

В простом примере, приведенном выше, достаточно всего двух запросов:

Spaghetti-Query/soln/split-query.sql

```
SELECT b.bug_id, COUNT(t.tag) AS count_tags
FROM Bugs b
LEFT OUTER JOIN Tags t ON (b.bug_id = t.bug_id)
WHERE b.bug_id = 1234
GROUP BY b.bug_id;

SELECT b.bug_id, COUNT(bp.product_id) AS count_products
FROM Bugs b
LEFT OUTER JOIN BugsProducts bp ON (b.bug_id = bp.bug_id)
WHERE b.bug_id = 1234
GROUP BY b.bug_id;
```

Эти два запроса выводят значения 4 и 2, как и ожидалось.

bug_id	count_tags
1234	4

bug_id	count_products
1234	2

Возможно, вас немного огорчит «неэлегантное» решение с разбиением на несколько запросов, но огорчение должно смениться облегчением, когда вы поймете все его преимущества для разработки, обслуживания и производительности:

- Запрос не создает нежелательное декартово произведение, как в предыдущих примерах, поэтому точность результатов проще проверить.
- При добавлении новых требований в отчет проще добавить еще один простой запрос, чем встраивать дополнительные вычисления в и без того сложный запрос.
- Ядро SQL обычно лучше и надежнее оптимизирует и выполняет простые, а не сложные запросы. Даже если вам кажется, что разбиение запроса приводит к дублированию работы, в целом оно все равно может оказаться более эффективным.
- В процессе проверки и ревизии кода или в ходе обучения команды вам будет проще объяснить, как работают несколько простых запросов, чем описывать логику одного сложного запроса.

Решение задачи руководителя

Как же следовало подойти к срочному требованию статистики по проекту? Шеф сказал: «Мне нужно знать, сколько у нас продуктов в работе, сколько разработчиков занималось исправлением ошибок, среднее количество исправленных ошибок на разработчика и количество исправленных ошибок, о которых сообщили клиенты».

Лучше всего разбить работу на части:

- Сколько продуктов:

Spaghetti-Query/soln/count-products.sql

```
SELECT COUNT(*) AS how_many_products
FROM Products;
```

- Сколько разработчиков исправляло ошибки:

Spaghetti-Query/soln/count-developers.sql

```
SELECT COUNT(DISTINCT assigned_to) AS how_many_developers
FROM Bugs
WHERE status = 'FIXED';
```

- Среднее количество исправленных ошибок на разработчика:

Spaghetti-Query/soln/bugs-per-developer.sql

```
SELECT AVG(bugs_per_developer) AS average_bugs_per_developer
FROM (SELECT dev.account_id, COUNT(*) AS bugs_per_developer
      FROM Bugs b JOIN Accounts dev
        ON (b.assigned_to = dev.account_id)
      WHERE b.status = 'FIXED'
      GROUP BY dev.account_id) t;
```

- О каком количестве исправленных ошибок сообщили клиенты:

Spaghetti-Query/soln/bugs-by-customers.sql

```
SELECT COUNT(*) AS how_many_customer_bugs
FROM Bugs b JOIN Accounts cust ON (b.reported_by = cust.account_id)
WHERE b.status = 'FIXED' AND cust.email NOT LIKE '%@example.com';
```

Некоторые из этих запросов достаточно сложны сами по себе. Попытки объединить их в один проход обернутся сущим кошмаром.

Автоматическое написание кода SQL

Результатом разбиения сложного запрос SQL может быть набор похожих запросов, возможно, слегка изменяющихся в зависимости от значений данных. Писать такие запросы утомительно и однообразно, и для таких задач уместно воспользоваться кодогенерацией.

Кодогенерация — метод написания кода, который выдает на выходе новый код, пригодный для компиляции или выполнения. Кодогенерация оправданна, если, чтобы написать код вручную, требуется слишком много усилий. Генератор кода избавит вас от однообразной работы.

Многотабличные обновления

Когда я работал консультантом, мне поручили решить сложную проблему SQL для одного из менеджеров.

Заглянув к нему в офис, я обнаружил измученного человека, который явно находился на грани. Мы едва обменялись приветствиями, когда он начал вываливать на меня свои беды. «Надеюсь, вы быстро разберетесь с этим; наша система управления запасами не работает *целый день*». Он не был новичком в SQL, но упомянул, что не один час проработал над инструкцией для обновления большого набора строк.

Проблема заключалась в том, что он не мог использовать последовательное выражение SQL в команде UPDATE для всех значений в строках. Собственно, изменения, которые нужно было применить, были разными для каждой строки. В его базе данных хранились данные об оборудовании компьютерной лаборатории и о загруженности каждого компьютера. Требовалось записать в столбец last_used дату последнего использования каждого компьютера.

Менеджер слишком старался решить эту сложную задачу в одной инструкции SQL — очередной пример антипаттерна «Спагетти-запрос». За те часы, которые он потратил на написание идеальной инструкции UPDATE, все изменения можно было внести вручную.

Вместо того чтобы писать одну инструкцию SQL для выполнения сложного обновления, я написал скрипт, который генерировал серию более простых команд SQL с желаемым эффектом:

Spaghetti-Query/soln/generate-update.sql

```
SELECT CONCAT('UPDATE Inventory '
  ' SET last_used = ''', MAX(u.usage_date), '''',
  ' WHERE inventory_id = ', u.inventory_id, ';') AS update_statement
FROM ComputerUsage u
GROUP BY u.inventory_id;
```

На выходе этого запроса создавалась серия инструкций UPDATE вместе с символами «;», готовая к выполнению в качестве скрипта SQL:

```
update_statement
UPDATE Inventory SET last_used = '2002-04-19' WHERE inventory_id = 1234;
UPDATE Inventory SET last_used = '2002-03-12' WHERE inventory_id = 2345;
UPDATE Inventory SET last_used = '2002-04-30' WHERE inventory_id = 3456;
UPDATE Inventory SET last_used = '2002-04-04' WHERE inventory_id = 4567;
...
```

Так мне удалось за считаные минуты сделать то, что не получилось у менеджера за несколько часов.

Выполнение многочисленных запросов или инструкций SQL может оказаться не лучшим способом решения вопроса. Постарайтесь выдержать баланс между эффективностью и необходимостью выполнить поставленную задачу.

Хотя SQL кажется достаточно мощным, чтобы решить сложную задачу в одном запросе, не поддавайтесь искушению построить карточный домик.

Как я могу сказать, что думаю,
пока не пойму, что говорю?

➤ *Э. М. Форстер (E. M. Forster)*

ГЛАВА 19

Неявные столбцы

Программисту PHP нужна помощь в отладке странного результата простого на первый взгляд запроса SQL к библиотечной базе данных:

Implicit-Columns/intro/join-wildcard.sql

```
SELECT * FROM Books b JOIN Authors a ON (b.author_id = a.author_id);
```

Запрос возвращал `NULL` вместо всех названий книг. Что было еще более странным, при выполнении другого запроса без соединения с `Authors` результат включал реальные названия книг, как и ожидалось.

Проблемы возникли из-за того, что расширение базы данных для PHP, которое он использовал, возвращало все результирующие строки в виде ассоциативного массива. Например, для обращения к столбцу `Books.isbn` можно было использовать синтаксис `$row["isbn"]`. В его таблицах `Books` и `Authors` присутствовал столбец с именем `title` (во втором случае — для титулов вроде `Dr.` или `Rev.`). В элементе массива `$row["title"]` может храниться только одно значение; в данном случае этот элемент массива занимает `Authors.title`. Для многих авторов в базе данных нет соответствия `title`, поэтому в результате `$row["title"]` оказывается равным `NULL`. Когда в запросе пропускалось соединение с `Authors`, конфликта между именами столбцов не было и название книги занимало элемент массива, как и следовало ожидать.

Проблема решается объявлением псевдонима для столбца, чтобы у одного из столбцов `title` было другое имя и каждый из них служил отдельным элементом массива.

Implicit-Columns/intro/join-alias.sql

```
SELECT b.title, a.title AS salutation
FROM Books b JOIN Authors a ON (b.author_id = a.author_id);
```

Второй вопрос звучал так: «Как назначить псевдоним одному столбцу, но при этом запросить другие столбцы?» Он хотел продолжать использовать шаблон (`SELECT *`), но применить псевдоним к столбцу, на который действует шаблон.

Цель: компактность кода

Разработчики не любят много печатать. В каком-то смысле это делает их выбор профессии парадоксальным, как в неожиданных концовках рассказов О'Генри.

Один из случаев, в котором, по мнению многих разработчиков, приходится вводить много лишних символов, — перечисление всех столбцов, используемых в запросе SQL:

Implicit-Columns/obj/select-explicit.sql

```
SELECT bug_id, date_reported, summary, description, resolution,
  reported_by, assigned_to, verified_by, status, priority, hours
FROM Bugs;
```

Неудивительно, что разработчики с удовольствием пользуются универсальными символами в SQL. Символ `*` обозначает *каждый столбец*, так что список столбцов здесь задается неявно. Так запросы становятся более компактными.

Implicit-Columns/obj/select-implicit.sql

```
SELECT * FROM Bugs;
```

Точно так же при использовании `INSERT` кажется логичным применять значения к столбцам по умолчанию: в том порядке, в каком они определяются в таблице.

Implicit-Columns/obj/insert-explicit.sql

```
INSERT INTO Accounts (account_name, first_name, last_name, email,
  password, portrait_image, hourly_rate)
VALUES ('bkarwin', 'Bill', 'Karwin', 'bill@example.com',
  SHA2('xyzzy', 256), NULL, 49.95);
```

Инструкция будет более компактной без явного перечисления столбцов.

Implicit-Columns/obj/insert-implicit.sql

```
INSERT INTO Accounts
VALUES (DEFAULT, 'bkarwin', 'Bill', 'Karwin', 'bill@example.com',
  SHA2('xyzzy', 256), NULL, 49.95);
```

Если использовать инструкцию `INSERT` с неявными столбцами, количество значений в секции `VALUES()` должно совпадать с количеством столбцов в таблице. Вы можете использовать ключевое слово `DEFAULT` в секции `VALUES()`, чтобы заполнить столбец значением по умолчанию для новой строки.

Антипаттерн: сокращение, которое сбивает с толку

Хотя использование универсальных символов и столбцов без имени призвано снизить количество вводимых символов, оно таит в себе ряд опасностей.

Нарушение рефакторинга

Предположим, вам необходимо добавить в таблицу `Bugs` новый столбец, например `date_due` для планирования задач.

Implicit-Columns/anti/add-column.sql

```
ALTER TABLE Bugs ADD COLUMN date_due DATE;
```

Теперь инструкция `INSERT` приводит к ошибке, потому что вы указали 11 значений вместо 12, которые ожидает получить таблица.

Implicit-Columns/anti/insert-mismatched.sql

```
INSERT INTO Bugs
VALUES (DEFAULT, CURDATE(), 'New bug', 'Test T987 fails...',
    NULL, 123, NULL, NULL, DEFAULT, 'Medium', NULL);

-- SQLSTATE 21S01: Количество столбцов не соответствует количеству значений
в строке 1.
```

В инструкции `INSERT`, использующей неявные столбцы, необходимо задать значения всех столбцов в том порядке, в каком они определяются в таблице. Если столбцы изменяются, инструкция выдает ошибку или даже присваивает значения не тем столбцам.

Допустим, вы выполняете запрос `SELECT *`, а поскольку имена столбцов неизвестны, при обращении к столбцам указывается их исходная позиция:

Implicit-Columns/anti/ordinal.py

```
import mysql.connector

cnx = mysql.connector.connect(user='scott', database='test')
cursor = cnx.cursor()

query = "SELECT * FROM Bugs WHERE bug_id = %s"
cursor.execute(query, (1234,))
for (row) in cursor:
    print(row[10])
```

Но вы не знаете, что другой участник команды удалил столбец из таблицы:

Implicit-Columns/anti/drop-column.sql

```
ALTER TABLE Bugs DROP COLUMN verified_by;
```

Столбец hours уже не находится в позиции 10. Приложение ошибочно использует значение из другого столбца. При переименовании, добавлении или удалении столбца результат запроса может измениться, причем может не поддерживаться кодом. При использовании универсальных символов невозможно предсказать, сколько столбцов вернет запрос. Ошибка может распространиться в коде, и к тому времени, когда вы заметите проблему, будет трудно вернуться к строке, в которой она возникла.

Скрытые затраты

Удобство использования универсальных символов в запросах может вредить производительности и масштабируемости. Чем больше столбцов возвращает запрос, тем больше данных должно передаваться по сети между приложением и сервером базы данных.

В реальной рабочей среде приложения одновременно должно выполняться множество запросов. Все они требуют пропускной способности канала связи. Даже высокоскоростная сеть может быть перегружена сотней клиентских приложений, запрашивающих тысячи строк за раз.

Технологии объектно-реляционного отображения (ORM), такие как Active Record, часто по умолчанию используют SELECT * для заполнения полей объекта, представляющего строку в базе данных. Даже если ORM предоставляет средства для переопределения этого поведения, многим разработчикам не хочется этим заниматься.

Что попросишь, то и получишь

«Есть ли краткая запись для запроса всех столбцов, кроме нескольких, которые мне не нужны?» — типичный вопрос разработчиков, использующих универсальный символ SQL. Возможно, они пытаются сократить расходы на выборку громоздких столбцов TEXT, которые им не нужны, но при этом хотят пользоваться удобными универсальными символами. А может, в таблице очень много столбцов, и им кажется, что проще перечислить исключаемые столбцы.

В SQL не поддерживается синтаксис, который бы означал: «Все столбцы, которые нужны, без тех, которые не нужны». Либо вы используете универсальный символ для запроса всех столбцов в таблице, либо вам придется явно перечислять все нужные столбцы.

Представьте, как бы выглядел синтаксис запросов SQL для получения всех столбцов с *, кроме конкретных исключаемых столбцов. В SQL его не существует, но теоретически он мог бы выглядеть так:

Implicit-Columns/anti/wildcard-except-fake.sql

```
SELECT * EXCEPT reported_by, assigned_to, verified_by, priority, hours
FROM Bugs;
```

SQL не может угадать, какие столбцы вам не нужны. Эти столбцы придется перечислять явно. Таким образом, на объеме ввода особо не сэкономишь. А может быть, он даже *увеличится*, если вашему запросу требуется меньше половины столбцов.

У любого другого разработчика, который читает ваш код, возникнет естественный вопрос: «Если эти столбцы исключены, то какие столбцы возвращает этот запрос?» А может быть, они пропустят ключевое слово `EXCEPT` и подумают, что указанные столбцы включаются в результаты, а не исключаются из них.

Как распознать антипаттерн

Следующие сценарии могут указывать на то, что в вашем проекте неявные столбцы используются некорректно, и это создает проблемы:

- «Приложение сломалось, потому что оно все еще ссылается на столбцы в результирующем наборе базы данных по старым именам. Мы пытались обновить код, но, видимо, что-то упустили».

  Вы изменили таблицу в базе данных (добавление, удаление, переименование или изменение порядка столбцов), но не изменили код приложения, который ссылается на таблицу. Отслеживать все эти ссылки довольно сложно.
- «Мы несколько дней искали узкое место сети и в итоге обнаружили избыточный трафик к серверу баз данных. По статистике, средний запрос извлекает более двух мегабайт данных, но выводит менее десятой части этого объема».

  Вы получаете много данных, которые вам не нужны.

Допустимые применения антипаттерна

Универсальные символы подходят для ситуативных быстрых запросов SQL для тестирования решения или для диагностической проверки текущих данных. Обслуживание одноразовых запросов не так актуально.

В примерах, приведенных в книге, универсальные символы используются для экономии места и для того, чтобы не отвлекать читателя от более важных частей запросов. В реальном коде лучше избегать универсальных символов SQL.

Если приложению нужно выполнить запрос, который адаптируется к добавлению, удалению, переименованию или перестановке столбцов, возможно, в нем стоит использовать универсальные символы. Обязательно запланируйте работу по диагностике возникающих проблем.

Вы можете использовать универсальные символы для каждой отдельной таблицы в запросе с соединением. Поставьте перед универсальным символом имя таблицы или псевдоним. Это позволит определить короткий список конкретных столбцов, которые неообходимо извлечь из одной таблицы, и при этом использовать универсальный символ для выборки остальных столбцов из другой таблицы. Пример:

Implicit-Columns/legit/wildcard-one-table.sql

```
SELECT b.*, a.first_name, a.email
FROM Bugs b JOIN Accounts a
  ON (b.reported_by = a.account_id);
```

Отслеживание длинного списка имен столбцов может занимать много времени. Для кого-то эффективность разработки важнее эффективности во времени выполнения.

Кроме того, можно назначить более высокий приоритет более коротким и удобочитаемым запросам. Универсальные символы сокращают длину вводимого текста и делают запросы компактнее, поэтому если это для вас важно, используйте универсальные символы.

Некоторые разработчики думают, что длинные запросы SQL, передаваемые от приложения к серверу базы данных, расходуют слишком много сетевых ресурсов. Теоретически длина запроса может оказаться важной. Но обычно строки данных, возвращаемые запросом, расходуют больше ресурсов канала, чем строка запроса SQL. Уделяйте внимание исключительным случаям, но не тратьте время на мелочи.

Решение: явное указание столбцов

Всегда явно перечисляйте все столбцы, которые вам нужны, не полагаясь на универсальные символы или неявные списки столбцов.

Implicit-Columns/soln/select-explicit.sql

```
SELECT bug_id, date_reported, summary, description, resolution,
  reported_by, assigned_to, verified_by, status, priority, hours
FROM Bugs;
```

Implicit-Columns/soln/insert-explicit.sql

```
INSERT INTO Accounts (account_name, first_name, last_name, email,
  password_hash, portrait_image, hourly_rate)
VALUES ('bkarwin', 'Bill', 'Karwin', 'bill@example.com',
  SHA2('xyzzy'), NULL, 49.95);
```

Все это выглядит довольно муторно, но оно того стоит по нескольким причинам.

Защита от ошибок

Вспомните *poka-yoke*, практику, применяемую японским производителем для проектирования систем, защищенных от ошибок (глава 5 «Сущность без ключа»). Чтобы повысить устойчивость запросов SQL к ошибкам и путанице, описанным выше, указывайте столбцы явно в `SELECT`-списке запроса.

- Если столбец будет перемещен, его позиция в результатах запроса не изменится.
- Если столбец будет добавлен в таблицу, он не появится в результатах запроса.
- Если столбец удален из таблицы, запрос выдает ошибку: но это полезная ошибка, потому что вы сразу видите код, который нужно исправить, и вам не приходится искать ее первопричину.

Аналогичные преимущества дает явное указание столбцов в командах `INSERT`. Заданный порядок столбцов переопределяет порядок в определении таблицы, и значения присваиваются, как и предполагалось. Добавленные столбцы, не указанные в инструкции, получают значения по умолчанию или `NULL`. При ссылке на удаленный столбец вы получите ошибку, но диагностика при этом упрощается. Это типичный пример принципа *раннего отказа*.

Вам это не понадобится (YAGNI[1])

Если вас беспокоит масштабируемость и пропускная способность продукта, возможно, где-то неэффективно используется канал связи. Нагрузка от запросов SQL может показаться безобидной в фазе разработки и тестирования, но она преподнесет сюрпризы, если в рабочей среде выполняются тысячи запросов SQL в секунду.

Когда вы отказываетесь от универсальных символов в SQL, вам, естественно, приходится продумывать, какие столбцы действительно нужны в конкретном запросе, и опускать ненужные. Это позволяет вводить меньше текста и более эффективно использовать канал связи.

Implicit-Columns/soln/yagni.sql

```
SELECT date_reported, summary, description, resolution, status, priority
FROM Bugs;
```

От универсальных символов все равно придется отказаться

Купив пакетик леденцов в автомате, вы сможете легко отнести их на стол, потому что они находятся в упаковке. Но после того как вы откроете пакетик, вам

[1] You Ain't Gonna Need It.

придется иметь дело с каждым леденцом по отдельности. Они могут выпасть из пакетика и рассыпаться. Если вы будете невнимательны, леденцы упадут под стол и привлекут насекомых. Однако чтобы съесть леденец, пакетик придется открыть.

Если в запросе SQL вам нужно использовать функцию или выражение в SELECT-списке, использовать псевдоним столбца или исключать столбцы, вам неизбежно придется открыть «упаковку», предоставляемую универсальным символом. При этом вы лишаетесь удобства работы со столбцами как с единым целым, но получаете доступ ко всему их содержимому.

Implicit-Columns/soln/select-expr.sql

```
SELECT bug_id, SUBSTRING(summary FROM 1 FOR 16) AS summary_shortened, ...
FROM Bugs;
```

Вам все равно приходится работать со столбцами по отдельности. Если вы с самого начала обойдетесь без универсальных символов, позже вам будет проще изменить запрос.

Берите все, что хотите, но съешьте все, что взяли.

ЧАСТЬ IV

Антипаттерны разработки приложений

Язык SQL предназначен для использования в контексте приложений, написанных на других языках: Python, Java, C, C++, C#, JavaScript, Elixir и т. д. Существуют правильные и неправильные способы применения SQL в приложениях, и в этой части книги описаны некоторые часто совершаемые ошибки.

Враг знает систему.

➤ *Принцип Шеннона*

ГЛАВА 20

Незащищенные пароли

Представьте, что вам в службу поддержки звонит клиент, работающий с одним из ваших приложений. У звонящего возникли проблемы со входом в систему.

«Это Пэт Джонсон из отдела продаж. Кажется, я забыл свой пароль. Вы не можете подсказать, какой у меня пароль?» — Пэт говорит робко, но почему-то торопится.

«Извините, я не могу этого сделать, — отвечаете вы. — Я могу сбросить вашу учетную запись и отправить сообщение на электронную почту, зарегистрированную для этой учетной записи. Выполните инструкции в сообщении и выберите новый пароль».

Голос в трубке становится нетерпеливым и напористым. «Но это смешно, — говорит он. — В предыдущей компании служба поддержки могла посмотреть мой пароль. Вы не способны выполнить свою работу? Хотите, чтобы я пошел к вашему руководству?»

Разумеется, вы хотите сохранить нормальные отношения с клиентами, поэтому вы пишете запрос SQL, находите пароль учетной записи Пэта Джонсона в текстовом виде и диктуете его по телефону.

Клиент вешает трубку. Вы замечаете своему коллеге: «Кажется, пронесло. Мне чуть не устроил скандал Пэт Джонсон. Надеюсь, он не будет жаловаться».

Коллега явно озадачен. «*Он*? Пэт Джонсон из отдела продаж — женщина. Похоже, ты только что сообщил ее пароль мошеннику».

Цель: восстановление и сброс паролей

В любом приложении, в котором нужен пароль, пользователь когда-нибудь его забудет. В большинстве современных приложений эта проблема решается про-

сто: пользователь может восстановить или сбросить пароль по электронной почте или SMS. Для этого он должен иметь доступ к электронной почте или мобильному устройству, связанному с его профилем в приложении.

Антипаттерн: хранение паролей в текстовом виде

В подобных решениях для восстановления пароля часто допускается одна ошибка: приложение разрешает пользователю запросить пароль в текстовом виде по электронной почте. Это серьезная брешь в безопасности, связанная с проектированием базы данных, и она создает целый ряд рисков, позволяющих несанкционированным пользователям получить привилегированный доступ к приложению.

Эти риски рассматриваются в следующих подразделах. Предполагается, что в базе данных ошибок из нашего примера присутствует таблица `Accounts`, в которой учетная запись каждого пользователя представлена отдельной строкой данных.

Хранение паролей

Пароль обычно хранится в таблице `Accounts` в столбце атрибута со строковым типом данных:

Passwords/anti/create-table.sql

```
CREATE TABLE Accounts (
  account_id SERIAL PRIMARY KEY,
  account_name VARCHAR(20) NOT NULL,
  email VARCHAR(100) NOT NULL,
  password VARCHAR(30) NOT NULL
);
```

Чтобы создать учетную запись, достаточно добавить в таблицу новую строку и указать пароль в виде строкового литерала:

Passwords/anti/insert-plaintext.sql

```
INSERT INTO Accounts (account_id, account_name, email, password)
  VALUES (123, 'billkarwin', 'bill@example.com', 'xyzzy');
```

Хранение пароля в виде простого текста и даже передача его по сети небезопасны. Если атакующий сможет прочитать команду SQL, которая используется для вставки пароля, он сможет увидеть пароль в исходном виде. Это относится и к инструкциям SQL, которые изменяют пароль или проверяют, что пользовательский ввод совпадает с хранимым паролем. У злоумышленника появляется ряд возможностей похитить пароль, например:

- Перехват сетевых пакетов при передаче инструкции SQL от клиента приложения к серверу базы данных, если коммуникации не зашифрованы с применением TLS. Это проще, чем может показаться; бесплатные программные инструменты (такие, как Wireshark[1] или tcpdump[2]) позволяют третьей стороне «подслушивать» передачу данных TCP/IP и читать пакеты, передаваемые между клиентом и сервером.
- Поиск журналов запросов SQL на сервере базы данных. Атакующему может понадобиться доступ к хосту сервера базы данных, но в этом случае он сможет прочитать файлы журналов, в которых хранятся инструкции SQL, выполненные сервером базы данных.
- Чтение данных из резервных копий БД на сервере или на резервных носителях. Резервные копии иногда даже проще взломать, чем саму базу данных, потому что владельцы не применяют к ним такие же жесткие меры защиты, как к базе данных.

Злоумышленники используют подобные методы для получения информации о системе паролей, поэтому необходимо принять меры для блокировки доступа не только к базе данных, но и к сети, журналам и резервным копиям.

Аутентификация

Когда пользователь пытается войти в систему, приложение сравнивает ввод пользователя с паролем, хранящимся в базе данных. Сравнение выполняется в виде простого текста, так как сам пароль хранится в этом виде. Например, следующий запрос может возвращать 0 (`false`) или 1 (`true`) в зависимости от того, совпадает ли ввод пользователя с паролем из базы данных:

Passwords/anti/auth-plaintext.sql

```
SELECT CASE WHEN password = 'opensesame' THEN 1 ELSE 0 END
  AS password_matches
FROM Accounts
WHERE account_id = 123;
```

В этом примере введенный пользователем пароль `opensesame` оказывается неверным, и запрос возвращает 0.

Как и в предыдущем разделе, где рассматривается хранение паролей, интерполяция введенной строки в запрос SQL в виде простого текста открывает возможности для его перехвата атакующей стороной:

---

[1] https://www.wireshark.org/

[2] https://www.tcpdump.org/

Многие разработчики создают запрос аутентификации с условиями для столбцов `account_id` и `password` в секции `WHERE`:

Passwords/anti/auth-lumping.sql

```
SELECT * FROM Accounts
WHERE account_name = 'bill' AND password = 'opensesame';
```

В этом запросе объединяются два разных случая: он возвращает пустой результирующий набор, если учетная запись не существует или пользователь ввел неверный пароль. Эти две причины неудачной аутентификации лучше обрабатывать по отдельности.

Например, если неудачных попыток ввода пароля подряд слишком много, можно временно заблокировать учетную запись, потому что это может служить признаком подбора пароля. Но если вы не отличаете неверное имя учетной записи от неверного пароля, эту закономерность выявить не получится.

Пересылка паролей по электронной почте

Так как пароль хранится в виде простого текста в базе данных, получить пароль в приложении несложно:

Passwords/anti/select-plaintext.sql

```
SELECT account_name, email, password
FROM Accounts
WHERE account_id = 123;
```

Приложение может отправить электронное письмо по запросу. Вы наверняка встречали такие сообщения как часть функциональности восстановления пароля на многих сайтах. Сообщение может выглядеть примерно так:

Пример сообщения для восстановления пароля:

```
From: daemon
To: bill@example.com
Subject: Запрос пароля
Вы запросили пароль для своей учетной записи "bill".
Ваш пароль "xyzzy".
Нажмите эту ссылку, чтобы войти со своими учетными данными:
https://www.example.com/login
```

Отправка пароля по электронной почте в текстовом виде создает серьезный риск для безопасности. Злоумышленники могут перехватывать и сохранять электронные письма разными способами. Безопасные протоколы для просмотра почты или ответственные системные администраторы, управляющие почтовыми серверами, не гарантируют достаточной защиты. Так как электронная почта

передается по сети, ее можно перехватить в других точках. Безопасные протоколы электронной почты не всегда достаточно широко распространены или находятся под вашим контролем.

Как распознать антипаттерн

Любое приложение, которое может восстановить пароль и отправить его пользователю, скорее всего, хранит его в виде простого текста — или хотя бы использует обратимое шифрование. Это антипаттерн.

Дело даже не в отправке паролей по электронной почте. Если пароль можно каким-либо способом восстановить, это значит, что он хранится неправильно. Если у приложения есть легитимный доступ к паролю, существует риск, что злоумышленники тоже его получат.

Допустимые применения антипаттерна

Иногда приложению приходится использовать пароль для обращения к стороннему сервису, то есть оно выполняет функции клиента. В таком случае пароль должен храниться в формате, пригодном для чтения. Как минимум желательно использовать не простой текст из базы данных, а кодировку, которую приложение сможет изменить.

Следует четко различать *идентификацию* и *аутентификацию*. Пользователь может идентифицировать себя кем угодно, но аутентификация докажет, что он именно тот, кем себя заявляет. Самый популярный способ аутентификации — это ввод пароля.

Если вы не можете обеспечить должную защиту от грамотных и серьезных взломщиков, то по сути, у вас есть механизм идентификации, но нет надежного механизма аутентификации. Это не обязательно критично.

Не каждое приложение подвергается риску атак, и не каждое приложение содержит конфиденциальную информацию, которую необходимо защитить. Например, к интрасетевому приложению могут иметь доступ только несколько человек, чья добропорядочность сомнений не вызывает. В таком случае механизма идентификации может быть достаточно для работы приложения, и для таких неформальных сред вполне может подойти простой вход с вводом учетных данных. Дополнительные усилия по созданию сильной системы аутентификации могут оказаться лишними.

Но будьте осторожны: приложения имеют тенденцию эволюционировать и выходить за рамки исходной среды или роли. Прежде чем открывать доступ к сим-

патичному интрасетевому приложению из-за брандмауэра компании, проконсультируйтесь с квалифицированным экспертом в области безопасности и попросите его оценить возможные риски.

Этика разработки

Если вы разрабатываете приложение с поддержкой паролей и вас просят реализовать функциональность восстановления паролей, лучше тактично возразить, предупредить лиц, принимающих решения, о возможных рисках и предложить альтернативное решение, которое обеспечит почти одинаковый результат: сброс пароля вместо его восстановления.

Подобно тому как электрик должен знать и уметь исправлять схему проводки, которая создает риск пожара, вы как разработчик ПО обязаны знать о проблемах безопасности и предлагать более безопасные решения.

На эту тему написана хорошая книга *24 Deadly Sins of Software Security* [HLV09]. Еще один хороший источник информации — проект Open Web Application Security Project[1].

Решение: хранение соленого хеш-кода пароля

Главная проблема этого антипаттерна заключается в том, что он разрешает чтение исходной формы пароля. Вместо этого следует выполнять аутентификацию пользовательского ввода с паролем без чтения последнего. В этом разделе описано, как реализовать подобную схему безопасного хранения паролей в базах данных SQL.

Хеш-функции

Закодируйте пароль с использованием односторонней *криптографической хеш-функции*. При этом входная строка преобразуется в новую, полностью неузнаваемую последовательность символов, которая называется *хешем*. Скрывается даже длина исходной строки, потому что хеш, возвращаемый хеш-функцией, представляет собой строку фиксированной длины. Например, алгоритм SHA-256 преобразует пароль `xyzzy` из нашего примера в 256-разрядную последовательность битов, которая обычно представляется 64-символьной строкой шестнадцатеричных цифр.

[1] https://owasp.org/

Passwords/soln/sha256.py

```
import hashlib

m = hashlib.sha256()
m.update(b'xyzzy')
print(m.hexdigest())

# Вывод:
# 184858a00fd7971f810848266ebcecee5e8b69972c5ffaed622f5ee078671aed
```

У криптографических хешей есть еще одна характерная особенность: они необратимы. Невозможно восстановить исходную строку по хеш-коду, потому что алгоритм хеширования разработан с расчетом на «потерю» части информации о входных данных. Взлом хорошего алгоритма хеширования должен требовать такого же объема работы, как простой подбор методом проб и ошибок.

В прошлом был популярен алгоритм SHA-1, но исследователи доказали, что 160-разрядный алгоритм хеширования недостаточно надежен; злоумышленники могут вычислить входную строку по строке хеша. Эта процедура занимает очень много времени, и все же это время меньше, чем потребовалось бы для подбора пароля методом проб и ошибок.

С 2010 года Национальный институт по стандартизации и технологии (NIST) исключил SHA-1 из числа алгоритмов хеширования с подтвержденной безопасностью в пользу более устойчивых разновидностей: SHA-224, SHA-256, SHA-384 и SHA-512[1]. Затем, в 2015 году, NIST одобрил семейство алгоритмов хеширования SHA3. Независимо от того, нужно ли обеспечивать соответствие стандартам NIST, для паролей рекомендуется использовать как минимум SHA-256.

`MD5()` — другая популярная функция хеширования, создающая 128-разрядные строки. Криптографическая слабость `MD5()` также была доказана, поэтому ее не стоит использовать для кодирования паролей. Ненадежные алгоритмы можно применять на практике, но только не для важной информации (такой, как пароли).

Работа с хешами в SQL

Ниже показано обновленное определение таблицы `Accounts`. Хеш-код пароля SHA-256 всегда состоит из 64 символов, поэтому столбец определяется как столбец `CHAR` фиксированной длины.

Passwords/soln/create-table.sql

```
CREATE TABLE Accounts (
  account_id SERIAL PRIMARY KEY,
  account_name VARCHAR(20),
```

[1] https://csrc.nist.gov/projects/hash-functions

```
  email VARCHAR(100) NOT NULL,
  password_hash CHAR(64) NOT NULL
);
```

Функции хеширования не являются частью стандартного языка SQL. Поэтому, возможно, чтобы поддерживать хеширование через расширение, вам потребуется воспользоваться возможностями СУБД. Например, MySQL 5.5 с поддержкой SSL включает функцию `SHA2()`.

Passwords/soln/insert-hash.sql

```
INSERT INTO Accounts (account_id, account_name, email, password_hash)
  VALUES (123, 'billkarwin', 'bill@example.com', SHA2('xyzzy', 256));
```

Чтобы проверить пользовательский ввод, примените к нему ту же хеш-функцию и сравните результат со значением, хранящимся в базе данных.

Passwords/soln/auth-hash.sql

```
SELECT CASE WHEN password_hash = SHA2('xyzzy', 256) THEN 1 ELSE 0 END
  AS password_matches
FROM Accounts
WHERE account_id = 123;
```

Предположим, что учетную запись пользователя необходимо заблокировать, чтобы лишить его доступа к системе. В идеале система аутентификации должна использовать отдельный атрибут, который будет помечать учетную запись как заблокированную. Если такая возможность не поддерживается, учетную запись можно заблокировать изменением ее пароля. Существует некоторая вероятность, что пользователь подберет новый пароль, но если вы замените строку, хранящуюся в хеше пароля, строкой, которая никогда не может быть сгенерирована хеш-функцией, то подобрать пароль будет невозможно. Например, строка `noaccess` содержит буквы, которые не являются шестнадцатеричными цифрами.

Добавление соли в хеш

Если вы храните хеши вместо паролей, а злоумышленник получит доступ к базе данных (например, найдет в мусоре флешку с резервной копией), он все равно сможет попробовать подобрать пароль методом проб и ошибок. Перебор всех паролей может занять много времени, но злоумышленник может подготовить собственную базу данных хешей вероятных паролей и сравнить с ними строки хешей, хранящиеся в вашей базе данных. Если хотя бы один пользователь выберет пароль, который представляет собой слово из словаря, взломщик легко найдет его. Для этого он поищет в базе данных паролей хеши, совпадающие с вариантами из подготовленной таблицы хешей. Это даже можно сделать в SQL:

Passwords/soln/dictionary-attack.sql

```
CREATE TABLE DictionaryHashes (
  password VARCHAR(100),
  password_hash CHAR(64)
);

SELECT a.account_name, h.password
FROM Accounts AS a JOIN DictionaryHashes AS h
  ON a.password_hash = h.password_hash;
```

Один из способов борьбы со «словарной атакой» основан на включении так называемой соли в выражение кодирования пароля. *Соль* (salt) представляет собой строку байтов, не имеющих значения, которая объединяется посредством конкатенации с паролем пользователя перед передачей полученной строки хеш-функцией. Даже если пользователь в качестве пароля выбрал слово в словаре, хеш, полученный из соленого пароля, не совпадет с хешем в базе данных хешей у взломщика. Например, если используется пароль `xyzzy`, вы увидите, что хеш слова отличается от хеша того же слова с добавлением нескольких случайных байтов:

Passwords/soln/sha256-salt.py

```
import hashlib

m = hashlib.sha256()
m.update(b'xyzzy')
print(m.hexdigest())

# Вывод:
# 184858a00fd7971f810848266ebcecee5e8b69972c5ffaed622f5ee078671aed

# Соль добавляется к предыдущему содержимому пароля,
# так что шестнадцатеричный дайджест содержит хеш двух строк,
# объединенных посредством конкатенации.
m.update(b'-0xT!sp9')
print(m.hexdigest())

# Вывод:
# 741b8aabe2e615e1c12876e66075199d602116ffb40b822dd5596baa7dafd40e
```

Каждый пароль должен использовать собственное значение соли, чтобы взломщику пришлось генерировать новую словарную таблицу хешей для каждого пароля. Это возвращает его к началу, потому что взлом паролей в базе данных будет занимать столько же времени, что и подбор методом проб и ошибок. Ниже приведен пример хранения соли и объединения ее с паролем перед вычислением хеш-кода, с последующей проверкой введенного пароля, который объединяется с той же солью.

Passwords/soln/salt.sql

```
CREATE TABLE Accounts (
  account_id SERIAL PRIMARY KEY,
  account_name VARCHAR(20),
  email VARCHAR(100) NOT NULL,
  password_hash CHAR(64) NOT NULL,
  salt BINARY(8) NOT NULL
);

INSERT INTO Accounts (account_id, account_name, email,
    password_hash, salt)
  VALUES (123, 'billkarwin', 'bill@example.com',
    SHA2(CONCAT('xyzzy', '-0xT!sp9'), 256), '-0xT!sp9');

SELECT (password_hash = SHA2(CONCAT('xyzzy', salt), 256))
  AS password_matches
FROM Accounts
WHERE account_id = 123;
```

Длина хорошей соли составляет минимум 8 байт, причем соль генерируется случайным образом для каждого пароля. В приведенных примерах строка соли состоит из печатаемых символов, но вы можете (и должны) включать в соль как печатаемые, так и непечатаемые байты.

Похожий, хотя и более сложный метод восстановления паролей по хеш-кодам использует так называемую *радужную таблицу*. Применение соли защищает и от этого метода атак.

Сокрытие пароля из SQL

Итак, теперь вы используете сильную функцию хеширования для кодирования пароля перед его сохранением, а также соль для предотвращения словарных атак. Можно подумать, что этого достаточно для обеспечения безопасности. К сожалению, пароль все еще включается в выражение SQL в виде простого текста; это означает, что его можно будет прочитать, если взломщик сможет перехватывать сетевые пакеты или если запросы SQL сохраняются в журнале, а файл журнала попадет не в те руки.

Чтобы защититься от подобных рисков, следует предотвратить включение паролей в текстовом виде в запросы SQL. Вместо этого вычисляйте хеш в коде приложения и используйте в запросах SQL только хеши. Злоумышленнику перехваченный хеш не принесет особой пользы, потому что он не сможет восстановить по нему пароль.

Перед вычислением хеша необходимо получить соль. Следующий пример на языке Python получает соль, вычисляет хеш и выполняет запрос для проверки пароля по соленому хешу, хранящемуся в базе данных:

Passwords/soln/auth-salt.py

```
import mysql.connector
import hashlib

cnx = mysql.connector.connect(user='scott', database='test')
cursor = cnx.cursor()

# Для простоты в этом примере используются фиксированные значения
# имени и пароля. В реальном приложении они заменяются пользовательским
# вводом.
name = 'bill'
password = 'xyzzy';

query = "SELECT password_hash, salt FROM Accounts WHERE account_name = %s"
cursor.execute(query, (name,))
for (row) in cursor:
    stored_hash = row[0]
    # Соль хранится в двоичном виде, ее необходимо преобразовать в строку.
    salt = row[1].decode()

# Выполнить конкатенацию введенного пароля с хранимой солью
# для получения хеша
m = hashlib.sha256()
m.update(f"{password}{salt}".encode('utf-8'))
input_hash = m.hexdigest()

# Сравнить хеш ввода с хешем, хранящимся в базе данных.
if input_hash == stored_hash:
    print("match successful!")
```

В веб-приложениях также существует риск перехвата данных в сети, между браузером пользователя и веб-сервером. Когда пользователь отправляет заполненную форму входа, браузер отправляет его пароль серверу в текстовом виде. Хорошая защита от сетевых перехватчиков требует использования протокола HTTPS при отправке пароля от браузера к приложению. Некоторые разработчики для лучшей защиты дополнительно хешируют пароль в коде на стороне браузера перед отправкой запроса HTTPS.

Сброс пароля вместо восстановления

Теперь, когда пароль хранится в более защищенном виде, можно вернуться к исходной цели: помочь пользователю, забывшему свой пароль. Восстановить пароль не удастся, потому что в базе данных хранится хеш вместо пароля. Хотя вам обратить хеш ничуть не проще, чем взломщику, открыть доступ пользователю к приложению можно другими способами. Ниже описаны две возможные реализации.

Первый вариант: когда пользователь, забывший пароль, обращается за помощью, вместо отправки пароля по электронной почте приложение отправляет сообще-

ние с временным паролем. Код сброса пароля в приложении знает пароль в текстовом виде, поэтому он может отправить его в сообщении. В базе данных хранится только хешированная версия пароля.

Для обеспечения дополнительной безопасности временный пароль может становиться недействительным по истечении некоторого времени, чтобы в случае перехвата сообщения снизить риск несанкционированного доступа злоумышленника.

Пример сообщения с временным паролем, сгенерированным системой

```
From: daemon
To: bill@example.com
Subject: Сброс пароля

Вы запросили сброс пароля для своей учетной записи.
Ваш временный пароль "p0trz3b1e".
Пароль действителен только в течение часа.
Перейдите по ссылке, чтобы войти в свою учетную запись
и установить новый пароль:
https://www.example.com/login
```

Второй вариант: вместо включения нового временного пароля в сообщение запрос на сброс пароля сохраняется в таблице базы данных, и ему присваивается уникальный идентификационный маркер:

Passwords/soln/reset-request.sql

```
CREATE TABLE PasswordResetRequest (
  account_id BIGINT UNSIGNED PRIMARY KEY,
  token CHAR(32) NOT NULL,
  expiration TIMESTAMP NOT NULL,
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

SET @token = MD5('billkarwin' || CURRENT_TIMESTAMP || RAND());

# Используйте REPLACE вместо INSERT в MySQL, чтобы команда
# перезаписывала существующую запись для заданного account_id.
REPLACE INTO PasswordResetRequest (account_id, token, account_id, expiration)
  VALUES (123, @token, CURRENT_TIMESTAMP + INTERVAL 1 HOUR);
```

Затем маркер включается в сообщение. Маркер можно отправить и в другом сообщении (например, SMS) при условии, что он отправляется на адрес, уже связанный с учетной записью, запросившей сброс пароля. Если запрос на сброс пароля поступит от постороннего, сообщение будет отправлено только реальному владельцу учетной записи.

Пример сообщения с временной ссылкой на страницу сброса пароля

```
From: daemon
To: bill@example.com
Subject: Сброс пароля

Вы запросили сброс пароля для своей учетной записи.

Перейдите по этой ссылке в течение часа, чтобы изменить пароль.
Через час ссылка станет недействительной, а ваш пароль останется прежним.

https://www.example.com/reset_password?token=f5cabff22532bd0025118905bdea50da
```

Когда приложение получает запрос специальной страницы `reset_password`, значение в параметре `token` должно соответствовать строке таблицы `PasswordResetRequest`, а срок действия для этой строки еще не должен истечь. Значение `account_id` для этой строки ссылается на таблицу `Accounts`, так что маркер ограничивается сбросом пароля только для одной конкретной учетной записи.

В любом из этих двух вариантов пользователь обязательно должен изменить пароль, прежде чем сможет выполнить какое-либо действие на сайте. После изменения пароль будет известен только пользователю, потому что в базе данных хранится только хеш, но не текстовая форма пароля.

Криптография непрерывно совершенствуется, стараясь опережать технологии взлома. Методы, описанные в этой главе, остаются актуальными независимо от типа используемого криптографического алгоритма хеширования, но вам стоит использовать современные рекомендуемые алгоритмы:

- Argon2[1] — алгоритм хеширования паролей, получивший премию на конкурсе PHC (Password Hashing Competition) в 2015 году.
- PBKDF25 — широко применяемый стандарт, *повышающий надежность ключей*.
- Bcrypt[2] реализует функцию *адаптивного хеширования*.

Со временем и этот список устареет. Если вы отвечаете за реализацию системы аутентификации, отслеживайте новейшие стандарты NIST для рекомендуемых алгоритмов.

Если вы можете прочитать пароль, то это сможет сделать и взломщик.

[1] https://www.argon2.com/

[2] https://tools.ietf.org/html/rfc2898

Мини-антипаттерн: хранение строк хешей в VARCHAR

«Какой тип данных использовать для хранения хешей?»

Результат хеширования обычно хранится в виде строки шестнадцатеричных цифр, и многие разработчики по умолчанию используют `VARCHAR` для хранения любых строк. Если они не знают, сколько памяти требует строка, они выбирают `VARCHAR(255)`, потому что это максимальная длина, поддерживаемая большинством реализаций SQL.

Все алгоритмы хеширования разные, но у них есть одно общее свойство: независимо от длины ввода любой алгоритм хеширования всегда возвращает результат постоянной длины. Таким образом определяется размер памяти, необходимой для хранения хеша. Не нужно использовать `VARCHAR`, так как длина фиксирована. Вместо этого используйте столбец `CHAR` фиксированной длины.

В двоичном представлении строка шестнадцатеричных цифр занимает только половину памяти. Например, шестнадцатеричное значение FF использует два символа при хранении в виде строки, но как двоичное значение занимает только один байт. Таким образом, строка хеша с фиксированной длиной может храниться либо в столбце `CHAR`, либо в столбце `BINARY` половинной длины. В каждом языке программирования должна существовать встроенная функция для преобразования шестнадцатеричных цифр в двоичное представление.

Те же байты можно закодировать в формате base64. Если в шестнадцатеричной записи три байта двоичных данных используют шесть символов, base64 кодирует те же три байта четырьмя символами. Это означает, что формат base64 на 50 % компактнее шестнадцатеричного. Формат base64 лучше воспринимается человеком, чем двоичный, поскольку использует только печатаемые символы.

Использование двоичного представления или base64 сокращает затраты памяти как на хранение данных, так и для любых индексов, создаваемых для этих данных.

Алгоритм	Биты	Тип для шестнадцатеричных цифр	Тип для Base64	Тип для байтов
MD5	128	`CHAR(32)`	`CHAR(22)`	`BINARY(16)`
SHA-1	160	`CHAR(40)`	`CHAR(27)`	`BINARY(20)`
SHA-256	256	`CHAR(64)`	`CHAR(43)`	`BINARY(32)`
SHA-512	512	`CHAR(128)`	`CHAR(86)`	`BINARY(64)`

Функции хеширования паролей, такие как Argon2, PBKDF2 или Bcrypt, поддерживают параметры, для которых различия в формате вывода будут незначительными. Формат кодирует выбранные параметры, а также использованную соль. Ниже приведен пример использования инструментария командной строки Argon2. Возвращаемая строка состоит из 84 символов, так что с учетом этой конфигурации для хранения строки следует использовать тип `CHAR(84)`.

```
echo "mypassword" | argon2 "salt1234" -e
$argon2i$v=19$m=4096,t=3,p=1$c2FsdDEyMzQ$61QNJTDZzd7eL6u5HDE0jCyhoLrGmH...
```

Результат следует формату PHC с пятью полями, разделенными символами `$`: алгоритм, версия, параметры, соль в кодировке base6-64 и 256-разрядный результат хеширования в кодировке base-64. Как правило, анализировать содержимое этой строки не придется; достаточно просто проверить новый пароль, хешируя его и сравнивая результат с хранимым результатом.

Процитируйте мои слова о том,
что меня неверно процитировали.

➤ *Граучо Маркс (Groucho Marx)*

ГЛАВА 21

SQL-инъекции

В марте 2010 года один серийный хакер был осужден и приговорен к 20 годам в федеральной тюрьме США за участие в крупнейшей краже персональных данных в истории[1]. Он похитил приблизительно 130 миллионов номеров кредитных и дебетовых карт, взламывая банкоматы и платежные системы с применением так называемой *SQL-инъекции*.

Уязвимости SQL-инъекции регулярно обнаруживаются в популярных программных продуктах и на сайтах, и один такой случай может включать взлом миллионов записей с персональными данными. Например, для блогерской платформы WordPress или одного из ее плагинов неотложные исправления безопасности SQL выходят несколько раз в год[2]. Уязвимости SQL-инъекции обнаруживались в каждой операционной системе, СУБД и языке программирования.

SQL-инъекции были впервые описаны в 1998 году, так почему же они до сих пор так распространены? Дело в том, что это не уязвимость в коде продукта, а ошибка программирования, которую ежедневно совершают многие из около 24,3 миллиона разработчиков по всему миру (по состоянию на 2021 год)[3].

В процессе обучения разработчикам редко преподают практики безопасного программирования (кроме классов, специально посвященных безопасности кода). Предполагается, что они самостоятельно освоят этот навык в ходе работы. Роберт Мартин, автор книги «Clean Code»[4] [Mar08], утверждает, что количество программистов увеличивается вдвое примерно каждые 5 лет, а это значит, что в любой момент времени примерно половина из них имеет менее 5 лет

---

[1] https://www.justice.gov/opa/pr/leader-hacking-ring-sentenced-massive-identity-thefts-payment-processor-and-us-retail

[2] https://www.bleepingcomputer .com/news/security/wordpress-related-vulnerabilities-saw-a-30-percent-uptick-in-2018/

[3] https://www.developernation.net/developer-reports/de20

[4] Мартин Р. «Чистый код. Создание, анализ и рефакторинг». — Санкт-Петербург, издательство «Питер».

практического опыта[1]. Если многие компании-разработчики нанимают в основном джунов, то, скорее всего, эти специалисты не прошли хорошую подготовку в области безопасности. Работодатель в первую очередь заинтересован в том, чтобы как можно быстрее выпустить продукт, а не заниматься долгой ревизией кода или тестированием безопасности, которые «хорошо бы провести». И компании, и их сотрудники обычно начинают заниматься исправлением уязвимостей только после взлома.

SQL-инъекции до сих пор не представляют для злоумышленников трудности, поскольку многие разработчики не понимают природы уязвимости и продолжают создавать ее в коде.

Цель: написание динамических запросов SQL

Язык SQL создавался для интеграции с кодом приложения. Когда вы строите запросы SQL в строковом виде и вставляете переменные приложения в строки, это часто называется *динамическим SQL*. В следующем примере переменная вставляется в форматированный строковый литерал (f-строку) Python. К тому времени, когда база данных получает запрос, значение `bugid` является частью запроса.

SQL-Injection/obj/dynamic-sql.py

```
bugid = 1234
query = f"SELECT * FROM Bugs WHERE bug_id = {bugid}";
cursor.execute(query)
```

Динамические запросы SQL — естественный механизм эффективного использования базы данных. Когда вы используете данные приложения, чтобы указать, как вы собираетесь строить запрос к базе данных, вы используете SQL как двусторонний язык. Приложение ведет своего рода диалог с базой данных.

Заставить программу делать то, что нужно, не так сложно — сложнее заставить ее не делать то, что не нужно. Дефекты внедрения SQL можно считать примерами второй категории.

Антипаттерн: выполнение непроверенного ввода как кода

SQL-инъекция происходит при интерполяции некоторого контента в строку запроса SQL, и этот контент изменяет синтаксис запроса непредвиденным образом. В классическом примере SQL-инъекции значение, интерполируемое в строку, завершает команду SQL и выполняет вторую полную команду. На-

[1] https://www.youtube.com/watch?v=BHnMItX2hEQ

пример, если `bugid` содержит текст `1234; DELETE FROM Bugs`, то полученный код SQL будет выглядеть так:

SQL-Injection/anti/delete.sql

```
SELECT * FROM Bugs WHERE bug_id = 1234; DELETE FROM Bugs
```

Подобная SQL-инъекция может быть довольно эффектной (рисунок Рэндалла Мунро (Randall Munroe)[1], используется с разрешения).

Обычно такие бреши не очевидны, но все равно опасны.

Неприятности случаются

Предположим, вы пишете веб-интерфейс для просмотра базы данных ошибок, и одна из страниц позволяет просматривать проекты по имени. В следующем примере приведена возможная реализация на Python и Flask.

SQL-Injection/anti/ohare.py

```
import mysql.connector
import json
from flask import Flask, Response, request

app = Flask(__name__) ❶

cnx = mysql.connector.connect(user='scott', database='test') ❷
cursor = cnx.cursor()

@app.route('/products', methods = ['GET']) ❸
def get_products(): ❹
    product_name = request.args.get("name") ❺

    # НЕБЕЗОПАСНО!
    sql = f"SELECT * FROM Products WHERE product_name = '{product_name}'" ❻

    cursor.execute(sql) ❼
    return json.dumps(cursor.fetchall())
```

[1] https://xkcd.com/327/

```
if __name__ == '__main__':
    app.run() ❽
```

❶ Создание экземпляра веб-приложения Flask.

❷ Открытие подключения к базе данных MySQL на локальном хосте.

❸ Определение маршрута, чтобы GET-запросы к веб-приложению по указанному пути обрабатывались следующей функцией.

❹ Определение функции-обработчика запроса.

❺ Присваивание значения параметра `name` GET-запроса переменной.

❻ Форматирование строки, содержащей запрос SQL, с использованием форматированного строкового литерала Python для интерполяции переменной.

❼ Выполнение строки как запроса SQL.

❽ Запуск веб-приложения Flask при запуске скрипта Python.

Проблемы начинаются, когда вашей команде поручают разработку ПО для международного аэропорта О'Хара в Чикаго. Естественно, вы присваиваете проекту имя «O'Hare». Для просмотра проекта в веб-приложении может использоваться запрос следующего вида:

```
http://bugs.example.com/project/view?name=O'Hare
```

Код интерполирует значение параметра в запрос SQL, но запрос оказывается недопустимым:

SQL-Injection/anti/ohare.sql

```
SELECT * FROM Products WHERE product_name = 'O'Hare'
```

Так как строка завершается первым обнаруженным символом кавычки, полученное выражение содержит короткую строку `'O'`, за которой следуют лишние символы `Hare'`, из-за чего база данных возвращает синтаксическую ошибку. Это обычная случайность без злого умысла. Вряд ли здесь произойдет что-то плохое, потому что инструкция с синтаксической ошибкой не будет выполнена. Но существует и куда большая опасность: команда может быть выполнена без ошибок, но сделает что-то незапланированное.

Одна из главных угроз веб-безопасности

SQL-инъекция становится значительной угрозой, когда злоумышленник использует ее для манипуляций с инструкциями SQL. Например, приложение может позволять пользователю менять пароль:

SQL-Injection/anti/set-password.py

```
def set_password():
    userid = request.form["userid"]
    password = request.form["password"]

    # НЕБЕЗОПАСНО!
    query = f"""UPDATE Accounts
        SET password_hash = SHA2('{password}', 256)
        WHERE account_id = {userid}"""

    cursor.execute(query)
    cnx.commit()
    return "OK"
```

Умный злоумышленник, способный догадаться, как параметры запроса используются в инструкции SQL, может отправить грамотно подобранную строку для эксплуатации уязвимости:

```
password=xyzzy&userid=123 OR TRUE
```

После интерполяции строки из параметра `userid` в инструкцию SQL синтаксис инструкции изменяется. Теперь инструкция меняет пароль для каждой учетной записи в базе данных, а не только одной указанной:

SQL-Injection/anti/set-password.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy', 256)
WHERE account_id = 123 OR TRUE;
```

SQL-инъекция изменяет синтаксис инструкции SQL до ее парсинга. Если в инструкцию перед парсингом вставляются динамические составляющие, возникает риск SQL-инъекции.

Способов изменить поведение инструкций SQL специально сформированной строкой бесчисленное множество. Они ограничиваются только воображением злоумышленника и вашей способностью защитить инструкции SQL.

Тяжелое лечение

Теперь вы знаете, какую угрозу несет SQL-инъекция, и возникает следующий логичный вопрос: как защитить код от несанкционированного использования? Возможно, вы читали блоги или статьи, в которых описывался какой-то метод и утверждалось, что он универсален против SQL-инъекций. В реальности ни один из этих методов не обеспечивает защиты от всех форм SQL-инъекций, так что придется использовать их все в соответствующих ситуациях.

Экранирование

Самый старый способ защиты запросов SQL от случайного несоответствия кавычек основан на *экранировании* всех символов кавычек, чтобы они не рассма-

тривались как конец строки, заключенной в кавычки. В стандартном SQL можно использовать два символа ' для определения одного литерального символа ':

SQL-Injection/anti/ohare-escape.sql

```
SELECT * FROM Products WHERE product_name = 'O''Hare'
```

Многие СУБД также поддерживают экранирование символом \, как и в большинстве других языков программирования:

SQL-Injection/anti/ohare-escape.sql

```
SELECT * FROM Products WHERE product_name = 'O\'Hare'
```

Идея в том, что данные приложения преобразуются до их интерполяции в строки SQL. Многие программные интерфейсы SQL предоставляют вспомогательные функции для этой цели. Так, в адаптере MySQL для Python доступна функция `escape()`. В следующем примере показано ее использование:

SQL-Injection/anti/ohare-escape.py

```
def get_products():
    product_name = cnx.converter.escape(request.args.get("name"))

    # БЕЗОПАСНО!
    sql = f"SELECT * FROM Products WHERE product_name = '{product_name}'"

    cursor.execute(sql)
    return json.dumps(cursor.fetchall())
```

Этот способ снижает риск SQL-инъекции, происходящей из-за несоответствия кавычек в динамическом контенте. Однако следует экранировать ввод каждый раз, когда вам понадобится интерполировать переменную в запрос SQL, а об этом легко забыть.

Кроме того, он полезен только для значений в строковых литералах SQL, заключенных в кавычки. Следующий пример демонстрирует использование функции экранирования для защиты строкового ввода, но экранирование специальных символов не работает с переменными, используемыми в качестве числовых литералов.

SQL-Injection/anti/set-password-escape.py

```
def set_password():
    userid = request.form["userid"]
    password = cnx.converter.escape(request.form["password"])

    # ВСЕ ЕЩЕ НЕБЕЗОПАСНО!
    query = f"""UPDATE Accounts
        SET password_hash = SHA2('{password}', 256)
        WHERE account_id = {userid}"""
```

```
    cursor.execute(query)
    cnx.commit()
    return "OK"
```

Если ввод `userid` содержит вредоносный контент, то запрос будет выполнен так, как если бы он был записан в следующем виде:

SQL-Injection/anti/set-password-escape.py

```
UPDATE Accounts SET password_hash = SHA2('xyzzy', 256)
WHERE account_id = 123 OR TRUE
```

Числовой столбец не может напрямую сравниваться со строкой, содержащей цифры, во всех СУБД. Некоторые СУБД могут неявно преобразовать строку в значимый числовой эквивалент, но в стандартном SQL придется намеренно использовать функцию `CAST()` для преобразования строки к числовому типу данных.

В некоторых малопонятных граничных случаях строки в кодировках, отличных от ASCII, могут проходить через функцию, предназначенную для экранирования кавычек, оставляя неэкранированные кавычки без изменений[1, 2].

Параметры запросов

Решение, которое чаще всего преподносится как панацея от SQL-инъекции, основано на использовании *параметров запросов*. Вместо интерполяции динамических значений в строку SQL при подготовке запроса в строке оставляются *заполнители для параметров*. Затем значение параметра предоставляется при выполнении подготовленного запроса.

В следующем примере адаптер MySQL для Python использует `%s` в качестве заполнителей. Этот адаптер также поддерживает формат заполнителя `%(`*имя*`)s`, если значения параметра предоставляются в объекте `dict` Python (объект `dict` в Python представляет собой множество пар «ключ — значение» по аналогии с `Hash` в Ruby или Perl либо `HashMap` в Java).

SQL-Injection/anti/parameter.py

```
name = request.args.get("name")
cnx.execute("SELECT * FROM Products WHERE product_name = %s", [name])
```

Многие разработчики рекомендуют это решение, потому что в этом случае не приходится экранировать динамический контент или беспокоиться о несовер-

[1] https://bugs.mysql.com/bug.php?id=8378

[2] https://shiflett.org/blog/2006/addslashes-versus-mysql-real-escape-string

шенных функциях экранирования. Действительно, параметры запроса создают очень сильную защиту от внедрения SQL. К сожалению, их нельзя считать универсальным решением, потому что значение параметра запроса всегда интерпретируется как одиночное литеральное значение. Ниже перечислены случаи, в которых динамический SQL не может использовать параметр.

Список значений не может быть одиночным параметром:

SQL-Injection/anti/parameter.py

```
bugid_list = "1234,3456,5678"
cnx.execute("SELECT * FROM Bugs WHERE bug_id IN ( %s )", [bugid_list])
```

Это выглядит, как будто вы предоставили одно строковое значение, состоящее из цифр и запятых, то есть это совсем не то же самое, что серия целых чисел:

SQL-Injection/anti/parameter.sql

```
SELECT * FROM Bugs WHERE bug_id IN ( '1234,3456,5678' )
```

Идентификатор таблицы не может быть параметром:

SQL-Injection/anti/parameter.py

```
table = "Bugs"
cnx.execute("SELECT * FROM %s WHERE bug_id = 1234", [table])
```

Это выглядит, как будто вы передали строковый литерал вместо имени таблицы (что на самом деле является синтаксической ошибкой).

SQL-Injection/anti/parameter.sql

```
SELECT * FROM 'Bugs' WHERE bug_id = 1234
```

Идентификатор столбца не может быть параметром:

SQL-Injection/anti/parameter.py

```
column = "date_reported"
cnx.execute("SELECT * FROM Bugs ORDER BY %s", [column]);
```

В этом примере сортировка — пустая операция, потому что выражение является строковой константой, одинаковой для каждой строки:

SQL-Injection/anti/parameter.sql

```
SELECT * FROM Bugs ORDER BY 'date_reported';
```

Ключевое слово SQL не может быть параметром:

SQL-Injection/anti/parameter.py

```
keyword = "DESC"
cnx.execute("SELECT * FROM Bugs ORDER BY date_reported %s", [keyword]);
```

Параметр интерпретируется как литеральная строка, а не как ключевое слово SQL. В этом примере результатом будет синтаксическая ошибка.

SQL-Injection/anti/parameter.sql

```
SELECT * FROM Bugs ORDER BY date_reported 'DESC'
```

Каким был полный запрос?

Многие думают, что использование параметров запросов SQL позволяет автоматически экранировать значения в инструкциях SQL. Это не совсем так, и если вы будете так относиться к параметрам запросов, вы не поймете, как они работают.

Сервер РСУБД разбирает код SQL при *подготовке* запроса. После этого ничто не сможет изменить синтаксис запроса SQL.

Вы предоставляете значения при *выполнении* подготовленного запроса. Каждое предоставленное значение подставляется на место очередного заполнителя, одно за другим.

Вы можете снова выполнить подготовленный запрос, подставив новые значения параметров вместо старых. Таким образом, РСУБД должна отслеживать запрос и значения параметров по отдельности; такой подход хорош с точки зрения безопасности.

Это означает, что если вы прочитаете подготовленную строку запроса SQL, она не будет содержать значений параметров. Если вы занимаетесь отладкой или протоколированием запросов, удобно просматривать инструкцию SQL со значениями параметров, но эти значения никогда не объединяются с запросом в удобочитаемую форму SQL.

Если вы выполняете отладку динамического SQL, храните запрос с заполнителями и значениями параметров по отдельности.

Хранимые процедуры

Использование хранимых процедур — еще один метод, который, по мнению многих разработчиков, защищает от уязвимостей SQL-инъекции. Как правило, хранимые процедуры содержат фиксированные инструкции SQL, которые парсятся при определении процедуры.

Тем не менее в хранимых процедурах возможно небезопасное использование динамического SQL. В следующем примере аргумент `input_userid` интерполируется в запрос SQL буквально, что небезопасно.

SQL-Injection/anti/procedure.sql

```
CREATE PROCEDURE UpdatePassword(
  IN input_password VARCHAR(20),
  IN input_userid VARCHAR(20))
BEGIN
  SET @sql = CONCAT('UPDATE Accounts
    SET password_hash = SHA2(', QUOTE(input_password), ', 256)
    WHERE account_id = ', input_userid);
  PREPARE stmt FROM @sql;
  EXECUTE stmt;
END
```

Использование динамического SQL в хранимой процедуре не более и не менее безопасно, чем использование динамического SQL в коде приложения. Аргумент `input_userid` может содержать вредоносный контент и генерировать небезопасную инструкцию SQL:

SQL-Injection/anti/set-password.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy', 256)
WHERE account_id = 123 OR TRUE;
```

Фреймворки доступа к данным

Иногда сторонники фреймворков доступа к данным утверждают, что их библиотека защищает код от рисков SQL-инъекции. Это утверждение не соответствует действительности для любого фреймворка, позволяющего записывать инструкции SQL в виде строк. Тем не менее оно создает ложную уверенность в том, что простой факт использования фреймворка волшебным образом превращает небезопасные привычки программирования в безопасные.

Ни один фреймворк не поможет писать безопасный код SQL. Фреймворк обычно предоставляет полезные вспомогательные функции, но за их использование отвечаете вы сами. При желании можно легко обойти эти функции и использовать стандартную строковую интерполяцию для небезопасного построения инструкций SQL.

Как распознать антипаттерн

Практически каждое приложение базы данных строит команды SQL динамически. Если вы строите любую часть инструкции SQL, объединяя строки конкатенацией или интерполируя переменные в строки, вы делаете приложение потенциально доступным для атак SQL-инъекций. Уязвимости SQL-инъекций встречаются так часто, что нужно исходить из того, что они есть в каждом приложении, использующем SQL, если только вы не провели ревизию кода специально, чтобы найти и исправить эти проблемы.

Допустимые применения антипаттерна

Этот антипаттерн отличается от большинства других описанных в книге тем, что не существует никаких разумных причин, которые бы оправдывали наличие уязвимостей SQL-инъекций в приложениях. Ваша обязанность как разработчика — писать код с учетом безопасности и помогать коллегам делать то же самое. Программный продукт безопасен настолько, насколько безопасно его слабейшее звено, — позаботьтесь о том, чтобы не стать именно таким звеном!

Решение: не доверяйте никому

Не существует единого способа защитить код SQL. Следует освоить все методы, перечисленные ниже, и использовать их в подходящих случаях.

Фильтрация ввода

Вместо того чтобы гадать, содержит ли ввод вредоносный контент, отфильтруйте любые недопустимые символы. Например, если вам нужно получить целое число, используйте такую функцию, как `int()`:

SQL-Injection/soln/casting.py

```
def get_products():
  bugid = int(request.args.get("bugid"))

  # БЕЗОПАСНО!
  sql = f"SELECT * FROM Bugs WHERE bug_id = {bugid}"

  cursor.execute(sql)
  return json.dumps(cursor.fetchall())
```

Другая разновидность фильтрации использует регулярные выражения для выделения безопасных подстрок. Если во вводе нет совпадений регулярного выражения, не разрешайте его. Преобразуйте ввод, используйте значение по умолчанию или верните ошибку.

SQL-Injection/soln/regexp.py

```
def get_bugs():
    o = request.args.get("order") ❶
    if re.search('^\w+$', o): ❷
        sortorder = o
    else:
        sortorder = "date_reported" ❸

    # БЕЗОПАСНО!
    sql = f"SELECT * FROM Bugs ORDER BY {sortorder}" ❹

    cursor.execute(sql)
    return json.dumps(cursor.fetchall())
```

❶ Переменной присваивается значение параметра `order` запроса GET.

❷ Регулярное выражение используется для проверки того, что значение является строкой из одного и более символов — алфавитно-цифровых или _. Если ввод соответствует этому шаблону, он присваивается переменной `sortorder`.

❸ Если значение не соответствует шаблону регулярного выражения, `sortorder` присваивается значение по умолчанию.

❹ Форматирование строки, содержащей запрос SQL, с использованием форматированного строкового литерала Python для интерполяции переменной. Вы уже знаете, что строку можно безопасно использовать как имя столбца, потому что она либо содержит значение по умолчанию, либо как минимум не содержит кавычек и других специальных символов.

Правило № 31: проверьте заднее сиденье

Если вам нравятся фильмы ужасов, вы знаете, что монстры любят прятаться за водителем и хватать героя, когда тот садится в машину. Мораль: нельзя быть уверенным, что даже такое знакомое место, как ваша машина, безопасно.

SQL-инъекции также могут принимать косвенные формы. Любая строка может содержать специальные символы, и ее использование в запросе SQL в неизменном виде будет небезопасным. В следующем примере имя запрашивается из таблицы Accounts, после чего интерполируется в полнотекстовую поисковую функцию.

SQL-Injection/anti/second-order.py

```
sql1 = "SELECT last_name FROM Accounts WHERE account_id = 123"
cursor.execute(sql1)

for row in cursor:
    # НЕБЕЗОПАСНО!
    sql2 = f"SELECT * FROM Bugs WHERE MATCH(description) AGAINST
                      ('{row["last_name"]}')"
    cursor.execute(sql2)
    print(cursor.fetchall())
```

Это может создать проблемы в предыдущем запросе, если пользователь записал свое имя в форме O'Hara. Тот факт, что строка хранится в базе данных, никоим образом не делает ее неуязвимой — строка может вызвать ошибку, если будет использована в последующем запросе SQL.

Такая разновидность SQL-инъекции называется *SQL-инъекцией второго порядка*. Обычно она приводит к ошибкам, но может использоваться и в атаках злоумышленников.

Параметризация динамических значений

Если динамические части запроса представляют собой простые значения, используйте параметры запроса, как объясняется на с. 271 в подразделе «Параметры запросов».

SQL-Injection/soln/parameter.py

```
def set_password():
    userid = request.form["userid"]
    password = request.form["password"]

    # БЕЗОПАСНО!
    sql = """UPDATE Accounts
        SET password_hash = SHA2(%s, 256)
        WHERE account_id = %s"""

    cursor.execute(sql, [password, userid])
    cnx.commit()
    return "OK"
```

Примеры в разделе «Антипаттерн» показывают, что после парсинга инструкции SQL параметр может подставляться только как единое значение. Таким образом, никакая SQL-инъекция не сможет изменить синтаксис в параметризованном запросе. Даже если злоумышленник попытается использовать вредоносное значение параметра (например, `123 OR TRUE`), РСУБД интерпретирует параметр как одиночное строковое значение. В худшем случае запрос не будет применен ни к одной строке данных; применение его к неправильным строкам маловероятно.

Результатом внедрения вредоносного значения станет относительно безопасная инструкция SQL, эквивалентная следующей:

SQL-Injection/soln/parameter.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy', 256)
WHERE account_id = '123 OR TRUE'
```

Сравнение `account_id` со строкой необычно, но не опасно. Так как `account_id` является числовым столбцом, строка преобразуется в ее числовое значение на основании начальных цифр `123`. Остальные символы игнорируются.

Используйте параметры запроса, если переменные приложения должны объединяться как литеральные значения в выражениях SQL.

Экранирование динамических значений

Параметры запроса обычно оказываются лучшим решением, но в редких случаях запросы с плейсхолдерами параметров вынуждают оптимизатор запросов принимать странные решения относительно используемых индексов.

Предположим, в таблице `Accounts` присутствует столбец с именем `is_active`. Для 99 % строк в нем хранится значение `true`, что дает неравномерное распределение значений. К запросу с условием `is_active = false` хорошо применить индекс, но чтение индекса для запроса с условием `is_active = true` будет неэффективным. Однако если использовать параметр в выражении `is_active = %s`, оптимизатор не знает, какое значение будет передано при выполнении подготовленного запроса, поэтому он может выбрать неподходящий план оптимизации.

Непросто узнать заранее, в каком случае произойдет подобная оптимизация. Если вы подозреваете, что запрос выполняется слишком медленно (то есть производительность приложения низкая), протестируйте план выполнения запроса с `EXPLAIN` (см. подраздел «Объяснение» в главе 13) с типичными и нетипичными значениями и посмотрите, изменится ли план оптимизации.

В подобных нестандартных случаях может быть лучше интерполировать значения прямо в команду SQL, несмотря на общую рекомендацию об использовании параметров запросов. В таком случае следите, чтобы строки были экранированы.

SQL-Injection/soln/interpolate.py

```
def get_account():
    account_name_escaped = cnx.converter.escape(request.args.get("name"))

    # БЕЗОПАСНО!
    sql = f"""SELECT * FROM Accounts
        WHERE account_name = '{account_name_escaped}'"""

    cursor.execute(sql)
    return json.dumps(cursor.fetchall())
```

Убедитесь, что используемая функция надежна и протестирована для нетипичных проблем безопасности SQL. Многие библиотеки доступа к данным включают такую функцию экранирования строк. Не пытайтесь реализовать собственную версию, не изучив как следует все риски безопасности. Не используйте функции, не относящиеся к SQL, например кодирование сущностей HTML.

Изоляция пользовательского ввода от кода

Параметры запросов и методы экранирования помогают объединять литеральные значения в выражения SQL, но они не работают с другими частями инструкций, например идентификаторами таблиц или столбцов, или ключевыми словами SQL. Чтобы сделать динамическими эти части запроса, вам понадобится другое решение.

Предположим, пользователям требуется выбрать способ сортировки списков ошибок (например, по статусу или по дате создания), а также направление сортировки.

SQL-Injection/soln/orderby.sql

```
SELECT * FROM Bugs ORDER BY status ASC;

SELECT * FROM Bugs ORDER BY date_reported DESC;
```

Параметризация предиката IN()

Вы уже видели, что элементы строки, разделенной запятыми, не могут передаваться в одном параметре. Для этого понадобится столько параметров, сколько элементов содержит список.

Предположим, вам требуется запросить шесть ошибок по первичным ключам:

SQL-Injection/soln/in-predicate.py

```
bug_list = [123, 234, 345, 456, 567, 678]
sql = "SELECT * FROM Bugs WHERE bug_id IN (%s, %s, %s, %s, %s, %s)"
cursor.execute(sql, bug_list)
```

Решение работает только в том случае, если `bug_list` содержит ровно шесть элементов (по количеству плейсхолдеров для параметров). Предикат SQL `IN` следует строить динамически, с количеством плейсхолдеров, равным количеству элементов в `bug_list`.

Следующий пример строит массив плейсхолдеров с такой же длиной, как у `bug_list`, после чего соединяет элементы массива с запятыми, прежде чем использовать их в выражении SQL.

SQL-Injection/soln/in-predicate.py

```
bug_list = [123, 234, 345, 456, 567, 678]
placeholders = ",".join(["%s"] * len(bug_list))
sql = f"SELECT * FROM Bugs WHERE bug_id IN ({placeholders})"
cursor.execute(sql, bug_list)
```

В следующем примере скрипт Python получает параметры запроса `order` и `dir`, и код наивно применяет интерполяцию для использования этих входных данных в запросе SQL в качестве имени столбца и ключевого слова.

SQL-Injection/soln/mapping.py

```
def get_bugs_unsafe():
  sortorder = request.args.get("order")
  direction = request.args.get("dir")

  # НЕБЕЗОПАСНО!
  sql = f"SELECT * FROM Bugs ORDER BY {sortorder} {direction}"
```

```
  cursor.execute(sql)
  return json.dumps(cursor.fetchall())
```

Скрипт исходит из того, что значение `order` содержит имя столбца, а `dir` — `ASC` или `DESC`. Это предположение небезопасно, потому что пользователь может отправить произвольные значения параметров в веб-запросе.

Вместо этого следует провести поиск значений из ввода в словаре, а затем использовать соответствующие значения в запросе SQL. Пример приведен ниже.

SQL-Injection/soln/mapping.py

```
def get_bugs_safe():
    sortorders = {"status": "status", "date": "date_reported"} ❶
    directions = {"up": "ASC", "down": "DESC"} ❷
    s = request.args.get("order") ❸
    if s in sortorders:
        sortorder = sortorders[s]
    else:
        sortorder = "bug_id"

    d = request.args.get("dir") ❹
    if d in directions:
        direction = directions[d]
    else:
        direction = "ASC"

    # БЕЗОПАСНО!
    sql = f"SELECT * FROM Bugs ORDER BY {sortorder} {direction}" ❺

    cursor.execute(sql)
    return json.dumps(cursor.fetchall())
```

❶ Объявление объекта Python `dict` с именем `sortorders`, который связывает допустимые варианты выбора (ключи) с именами столбцов SQL (значениями).

❷ Объявление объекта Python `dict` с именем `directions`, который связывает действительные варианты выбора (ключи) с ключевыми словами SQL `ASC` и `DESC` (значениями).

❸ Если варианты пользователя совпадают с ключами массива в `sortorders`, используются соответствующие значения. В противном случае используется значение по умолчанию.

❹ Точно так же, если варианты пользователя совпадают с ключами массива в `directions`, используются соответствующие значения. В противном случае используется значение по умолчанию.

❺ Теперь переменные `sortorder` и `direction` могут безопасно использоваться в запросах SQL, потому что они содержат только значения, явно объявленные в коде.

Преимущества этого метода:

- Пользовательский ввод никогда не объединяется с запросом SQL, что снижает риск SQL-инъекции.
- Динамической может быть любая часть инструкции SQL, включая идентификаторы, ключевые слова SQL и даже целые выражения.
- У вас появляется простой и эффективный способ проверки пользовательского ввода.
- Внутренние подробности реализации запросов отделяются от пользовательских интерфейсов.

Варианты ввода жестко закодированы в приложении, но это допустимо для имен таблиц, имен столбцов и ключевых слов SQL. Произвольный ввод обычно разрешается для значений данных, но не для идентификаторов или ключевых слов.

Сложнее ли использовать параметры запросов?

Вы наверняка уже видели примеры кода и учебники, в которых запросы SQL строятся с использованием конкатенации строк или интерполяции переменных. И конечно, все эти книги по программированию и блоги не могут ошибаться? Старые привычки трудно изменить.

Использовать параметры запросов SQL обычно проще, чем традиционный код. Следующий классический пример на PHP показывает, как экранирующая функция может создать больше проблем, чем решить, и как параметры запроса могут это исправить.

SQL-Injection/soln/concat.php

```
$sql = "INSERT INTO Accounts (account_id, account_name, email, password)
  VALUES (".intval($account_id).","
  .mysqli_real_escape_string($conn, $account_name)."', '"
  .mysqli_real_escape_string($conn, $email)."' SHA256('"
  .mysqli_real_escape_string($conn, $password)."', 256)";
mysqli_query($sql);
```

В приведенном примере отсутствуют две кавычки, одна запятая и одна круглая скобка. Сколько времени вам понадобилось, чтобы найти их? Это трудно сделать, потому что синтаксис SQL чередуется с выражениями PHP.

Двойные кавычки многократно открывают и закрывают строки PHP, а внутри строк одинарные кавычки открывают и закрывают синтаксис литеральных строк SQL. В таких ситуациях на отладку кода может уйти не один час. Сравните со следующим кодом PHP, использующим параметры запросов.

SQL-Injection/soln/parameter-mysqli.php

```
$sql = "INSERT INTO Accounts (account_id, account_name, email, password)
  VALUES (?, ?, ?, SHA256(?, 256))";
$stmt = mysqli_prepare($conn, $sql);
mysqli_stmt_bind_param($stmt, "isss", $account_id, $account_name, $email,
$password);
mysqli_stmt_execute($stmt);
```

Такой код намного проще писать, читать и отлаживать. Заключать плейсхолдеры параметров в кавычки не обязательно, так что вы не пропустите кавычку. Вы быстро найдете все пропущенные запятые и скобки. Кроме того, другому разработчику, читающему ваш код, будет проще понять логику запроса.

Расширение PDO для PHP еще больше упрощает работу с параметрами запроса.

SQL-Injection/soln/parameter-pdo.php

```
$sql = "INSERT INTO Accounts (account_id, account_name, email, password)
  VALUES (?, ?, ?, SHA256(?, 256))";
$stmt = $pdo->prepare($sql);
$stmt->execute([$account_id, $account_name, $email, $password]);
```

Научиться использовать параметры запросов не так сложно, а когда вы это сделаете, ваша производительность возрастет в разы. Надежная защита от SQL-инъекции — всего лишь одно из преимуществ.

Код-ревью

Лучший способ найти ошибку — взглянуть на код со стороны. Попросите коллегу, знакомого с рисками SQL-инъекции, проанализировать ваш код. Забудьте о гордости или эго — конечно, вас может смущать пропущенная ошибка, но неужели это хуже, чем нести ответственность позже, когда злоумышленник взломает ваш сайт из-за бреши в безопасности?

При анализе SQL-инъекции соблюдайте следующие правила.

1. Найдите инструкции SQL, в которых используются переменные приложения, конкатенация строк или замена.

2. Отследите источник всего динамического контента инструкций SQL. Найдите любые данные, поступающие извне — из пользовательского ввода, файлов, окружения, веб-сервисов, стороннего кода и даже из строки, извлеченной из базы данных.
3. Любой внешний контент следует считать потенциально опасным. Используйте фильтры, валидаторы и ассоциативные массивы для преобразования небезопасного контента.
4. Объединяйте внешние данные в инструкции SQL с использованием параметров запросов или надежных функций экранирования.
5. Не забудьте проверять хранимые процедуры и другие места, в которых могут присутствовать динамические инструкции SQL.

Код-ревью (инспекция кода) — самый точный и экономичный способ найти SQL-инъекции. Всегда выделяйте на него время и сделайте эту задачу обязательной. А в ответном одолжении проанализируйте код коллеги.

Для отслеживания непредвиденных запросов SQL, которые могут появиться в результате атак SQL-инъекций, также можно воспользоваться журналом запросов SQL или технологией мониторинга производительности приложения (APM).

Разрешайте пользователям вводить данные, но никогда не позволяйте им вводить код.

Мини-антипаттерн: параметры запроса в кавычках

«Почему мой запрос ничего не находит, если он параметризованный? Когда я интерполирую переменную в строку SQL, все работает».

Типичная ошибка — заключение плейсхолдера параметра запроса в кавычки (в следующем примере в качестве плейсхолдера используется символ `?`, принятый по умолчанию в MySQL):

```
SELECT * FROM Bugs WHERE bug_id = '?'
```

В этом случае вопросительный знак используется как строковый литерал, а не как плейсхолдер параметра. Значение `bug_id`, являющееся целым числом, сравнивается с числовым значением строкового литерала, которое равно 0. В этой таблице значения `bug_id` начинаются с 1, так что значение 0 не соответствует никакой строке.

Если подумать, то так и должно быть, потому что в противном случае простой вопросительный знак нельзя было бы использовать в строке.

```
SELECT * FROM Bugs WHERE summary = 'Is this a bug?'
```

Эта проблема также встречается, когда разработчик объединяет параметр запроса с литеральным текстом. В следующем примере представлен случай, когда разработчик намеревался добавить универсальные символы % до и после плейсхолдера параметра запроса в выражениях поиска по шаблону типа LIKE:

```
SELECT * FROM Bugs WHERE summary LIKE '%?%'
```

Плейсхолдер параметра должен интерпретироваться как отдельный маркер в синтаксисе запроса. Он ведет себя как строковый литерал. Его можно использовать в выражениях, например, при конкатенации строк:

```
SELECT * FROM Bugs WHERE summary LIKE CONCAT('%', ?, '%')
```

Кроме того, плейсхолдер можно включить в запрос отдельно и передать значение параметра, которое является результатом выражения форматирования строки в коде клиентского приложения. В следующем примере Python используется форматированный строковый литерал для заключения переменной между универсальными символами %, после чего полученная строка передается как параметр запроса.

```
query = "SELECT * FROM Bugs WHERE summary LIKE %s"
pattern = f"%{word}%"
cursor.execute(query, [pattern])
```

Обратите внимание: плейсхолдер в запросе не заключается в кавычки.

Те, кто нам важен, возражать не станут,
а те, кто станет возражать, нам не важны.

➤ *Бернард Барух (Bernard Baruch)*
(о распределении мест для гостей на званом обеде)

ГЛАВА 22

Чистка псевдоключа

Ваш руководитель подходит к вам, держа в руках две распечатки отчетов. «Бухгалтерия говорит, что у нас расхождения между отчетом за этот и предыдущий квартал. Я посмотрел, они абсолютно правы. Большинство последних активов исчезло. Что случилось?»

Вы просматриваете отчеты, и закономерность наводит вас на одну мысль. «Нет, все на месте. Вы попросили меня почистить базу данных, чтобы в ней не было пропущенных строк. Вы еще сказали, что бухгалтерия постоянно задает вопросы об отсутствующих активах из-за пропусков в нумерации. Тогда я перенумеровал некоторые строки, чтобы все они заняли места пропущенных строк. Теперь пропусков нет — используются все числа от 1 до 12 340. Все они здесь, но некоторые просто изменили номер и сместились вверх. Вы мне сами сказали это сделать».

Шеф качает головой. «Но это не то, что мне нужно. Бухгалтерия отслеживает амортизацию по номерам активов. Номер каждой единицы оборудования должен оставаться одинаковым во всех ежегодных отчетах. Кроме того, все инвентарные номера напечатаны на наклейках на каждой единице. Чтобы поменять наклейки во всей компании, у нас уйдет несколько недель. Нельзя ли вернуть всем строкам исходные значения идентификаторов?»

Ничего не поделаешь, вы возвращаетесь к компьютеру и начинаете работать. Но внезапно вас осеняет: «А что делать с теми активами, которые мы приобрели в этом месяце, после консолидации идентификаторов? Им присвоены идентификаторы, которые использовались до смены нумерации. Если я верну всем идентификаторам прежние значения, то что делать с дубликатами?»

Цель: очистка данных

Некоторых людей нервируют пропуски в последовательностях.

bug_id	status	product_name
1	OPEN	Open RoundFile
2	FIXED	ReConsider
4	OPEN	ReConsider

С одной стороны, такое беспокойство понятно: что случилось со строкой с `bug_id` 3? Почему запрос не вернул эту ошибку? Она потерялась в базе данных? А что в ней было? Может, об этой ошибке сообщил один из важных клиентов? А не случится ли так, что вину за потерю данных возложат на меня?

Разработчики, использующие антипаттерн «Чистка псевдоключа», стремятся избавиться от этих неприятных вопросов. Они отвечают за проблемы с целостностью данных, но, как правило, не очень хорошо понимают технологию базы данных (или не доверяют ей), чтобы уверенно интерпретировать результаты отчетов.

Антипаттерн: заполнение пропусков

Видимые пропуски можно заполнить двумя способами.

Назначение чисел, не входящих в последовательность

Вместо того чтобы выделять новое значение первичного ключа с использованием механизма автоматического генерирования псевдоключа, можно назначить новой строке первое неиспользуемое значение первичного ключа. Таким образом, при вставке данных пропуски будут заполняться естественным образом.

bug_id	status	product_name
1	OPEN	Open RoundFile
2	FIXED	ReConsider
4	OPEN	ReConsider
3	NEW	Visual TurboBuilder

Однако для того, чтобы найти наименьшее неиспользуемое значение, придется выполнить ненужный запрос с соединением таблицы с самой собой:

Neat-Freak/anti/lowest-value.sql

```
SELECT b1.bug_id + 1
FROM Bugs b1
LEFT OUTER JOIN Bugs AS b2 ON (b1.bug_id + 1 = b2.bug_id)
WHERE b2.bug_id IS NULL
ORDER BY b1.bug_id LIMIT 1;
```

В книге уже рассматривалась проблема одновременного доступа, возникающая при попытке выделения уникального значения первичного ключа с выполнением запроса вида `SELECT MAX(bug_id)+1 FROM Bugs` (см. врезку «Специальные контексты для последовательностей идентификаторов» в главе 4, с. 70). Такая же проблема может возникнуть, когда два приложения одновременно пытаются найти наименьшее неиспользуемое значение. Когда оба пытаются использовать одно значение в качестве значения первичного ключа, одна попытка завершается успешно, а другая приводит к ошибке. Этот способ неэффективен и подвержен ошибкам.

Перенумерация существующих строк

Возможно, вы захотите быстрее добиться непрерывности значений первичного ключа и ожидание, пока новые строки заполнят пропуски, вам покажется слишком долгим. Можно воспользоваться стратегией обновления значений ключей существующих строк для устранения пропусков и обеспечения непрерывности значений. Обычно это означает, что нужно найти строку с наибольшим значением первичного ключа и обновить ее наименьшим неиспользуемым значением. Например, можно обновить значение 4 до значения 3:

Neat-Freak/anti/renumber.sql

```
UPDATE Bugs SET bug_id = 3 WHERE bug_id = 4;
```

bug_id	status	product_name
1	NEW	Open RoundFile
2	FIXED	ReConsider
3	DUPLICATE	ReConsider

Чтобы воспользоваться этим методом, необходимо найти неиспользуемое значение ключа способом, аналогичным предыдущему для вставки новых строк. Также необходимо выполнить команду `UPDATE` для переназначения значения первичного ключа. На каждом из этих шагов может возникнуть проблема параллельного доступа. Чтобы заполнить большой пропуск в числовой последовательности, вам придется многократно повторить эти действия.

Кроме того, измененное значение необходимо обновить во всех дочерних записях, ссылающихся на перенумерованные строки. Это проще всего сделать при объявлении внешних ключей с конструкцией `ON UPDATE CASCADE`, иначе придется отключить ограничения, вручную обновить все дочерние записи и восстановить ограничения. Это трудоемкий, рискованный процесс, который может нарушить работоспособность базы данных, и если вам кажется, что лучше обойтись без него, то вам не кажется.

Даже если вы очистите БД таким способом, это ненадолго. Когда псевдоключ генерирует новое значение, оно будет больше последнего сгенерированного (даже если строка с этим значением уже была удалена или изменена), а *не* наибольшим значением, находящимся в настоящий момент в таблице, как считают некоторые разработчики. Допустим, вы обновили строку с наибольшим значением `bug_id 4` наименьшим неиспользуемым значением для заполнения пробела. Следующей строке, добавляемой с использованием генератора псевдоключа по умолчанию, будет выделено значение `5`, и в позиции значения `4` появится новый пропуск.

Как возникает несоответствие данных

В преамбуле главы описаны некоторые проблемы, которые могут возникнуть при изменении нумерации значений первичного ключа. Если другая система, внешняя по отношению к базе данных, зависит от идентификации записей по первичному ключу, то в результате обновлений ссылки в данных этой системы становятся недействительными.

Повторное использование значения первичного ключа строки — плохая идея, потому что пропуск мог возникнуть в результате удаления или отмены создания строки по уважительной причине. Допустим, пользователь с `account_id 789` был заблокирован в системе из-за отправки оскорбительных сообщений. Политика компании требует удаления учетной записи нарушителя, но если вы повторно используете первичные ключи, значение `789` в дальнейшем будет закреплено за другим пользователем. Так как еще не все получатели прочитали эти оскорбительные сообщения, вы будете получать новые жалобы на *учетную запись 789*. И в этом будут обвинять несчастного пользователя, которому был присвоен этот номер, хотя он не сделал ничего плохого.

Не переназначайте значения псевдоключа только потому, что кажется, что они не используются.

Как распознать антипаттерн

Следующие фразы могут указывать на то, что кто-то среди вас собирается применить антипаттерн «Чистка псевдоключа»:

- «Как повторно использовать автоматически сгенерированный идентификатор после отмены вставки?»

  Выделение значения псевдоключа не отменяется; если бы оно отменялось, РСУБД пришлось бы выделять значения псевдоключа в контексте транзакции. Это привело бы либо к ситуации гонки, либо к блокировке при одновременной вставке данных несколькими клиентами.
- «Куда пропал `bug_id` 4?»

  Говорящий беспокоится о том, что в последовательности первичных ключей некоторые числа не используются.
- «Как запросить первый неиспользуемый идентификатор?»

  Цель подобных запросов почти всегда — переназначение идентификаторов.
- «А если не останется свободных чисел?»

  Обычное оправдание переназначения неиспользуемых значений идентификатора. См. «Мини-антипаттерн: хватит ли BIGINT?» в главе 4, с. 75.

Допустимые применения антипаттерна

Нет никаких причин менять значение псевдоключа, так как оно все равно не должно содержать никакой полезной информации. Если значения в столбце первичного ключа имеют какой-то смысл, значит, этот столбец является *естественным* ключом, а не псевдоключом. А изменение значений естественного ключа — тривиальная задача.

Решение: смириться

Значения в любом первичном ключе должны быть уникальными и отличными от `NULL`, поэтому вы можете использовать их для ссылки на конкретные строки, но помните одно правило: для идентификации строк они не обязательно должны быть последовательными числами.

Нумерация строк

Числа, возвращаемые генераторами псевдоключей, выглядят почти как номера строк, поскольку они *монотонно возрастают* (каждое последовательное значение на 1 больше предыдущего), но это всего лишь случайное следствие их реализации. Такое генерирование значений — удобный способ обеспечить их уникальность.

Не путайте номера строк с первичными ключами. Первичный ключ идентифицирует одну строку таблицы, тогда как номера строк идентифицируют строки в результирующем наборе. Номера строк в результатах запроса не соответству-

ют значениям первичного ключа в таблице, особенно при использовании в запросах таких операций, как `JOIN`, `GROUP BY` или `ORDER BY`.

Номера строк целесообразно использовать по многим причинам, например, для получения подмножества строк из результата запроса. Это часто называют разбивкой на страницы, как страницы интернет-поиска. Для выбора подмножества таким способом используются возрастающие и последовательные номера строк независимо от формы запроса.

В SQL:2003 описаны *оконные функции*, включая функцию `ROW_NUMBER()`, которая возвращает последовательные числа для результата запроса. Нумерация строк обычно используется для ограничения результата запроса диапазоном строк:

Neat-Freak/soln/row_number.sql

```
SELECT t1.* FROM
  (SELECT a.account_name, b.bug_id, b.summary,
    ROW_NUMBER() OVER (ORDER BY a.account_name, b.date_reported) AS rn
  FROM Accounts a JOIN Bugs b ON (a.account_id = b.reported_by)) AS t1
WHERE t1.rn BETWEEN 51 AND 100;
```

Сейчас эти функции поддерживаются практически всеми популярными базами данных на основе SQL.

Использование GUID

Также можно генерировать случайные значения псевдоключа — при условии, что каждое число не будет использоваться больше одного раза. В некоторых базах данных для этой цели поддерживаются *глобально-уникальные идентификаторы* (GUID).

GUID — псевдослучайное число из 128 бит (обычно представляемое последовательностью, содержащей не менее 32 шестнадцатеричных цифр). Для практических целей значения GUID уникальны, что позволяет использовать их для генерирования псевдоключа.

В следующем примере используется синтаксис Microsoft SQL Server 2005:

Neat-Freak/soln/uniqueidentifier-sql2005.sql

```
CREATE TABLE Bugs (
  bug_id UNIQUEIDENTIFIER DEFAULT NEWID(),
  -- . . .
);

INSERT INTO Bugs (bug_id, summary)
VALUES (DEFAULT, 'crashes when I save');
```

В результате создается строка следующего вида:

bug_id	summary
0xff19966f868b11d0b42d00c04fc964ff	Crashes when I save

У GUID есть минимум два преимущества перед генераторами псевдоключа:

- Псевдоключи можно генерировать одновременно на нескольких серверах баз данных без использования одинаковых значений.
- Никто не пожалуется на пропуски в значениях — все будут слишком заняты жалобами на необходимость вводить первичные ключи, состоящие из 32 шестнадцатеричных цифр.

Из второго обстоятельства вытекают некоторые недостатки:

- Значения длинные, и их неудобно вводить.
- Значения случайные, так что не получится вывести какую-либо закономерность, а большее значение не обязательно будет соответствовать более поздней записи.
- Для хранения GUID требуется не менее 16 байт. Значения занимают больше места, а операции с ними выполняются медленнее, чем с типичным целым псевдоключом из 4 байт.

Самая важная проблема

Теперь вы знаете, какие проблемы возникают при изменении нумерации псевдоключей и какие могут быть альтернативы этому, но остается решить еще одну большую проблему: что делать с руководством, которое требует навести порядок в базе данных и заполнить пропуски в псевдоключе? Это проблема коммуникаций, а не технологий. Тем не менее нельзя исключать, что вам придется убедить начальника защищать целостность данных в базе.

- *Объясните суть технологии.* Обычно честность — лучшая политика. Будьте вежливы и подтвердите, что вы поняли задачу. Например, скажите следующее:

  «Пропуски выглядят не очень, но никакого вреда в них нет. Строки иногда пропускают или удаляют, и это нормально. Мы назначаем новое число для каждой новой строки базы данных — чтобы не писать код, вычисляющий, какие старые номера можно использовать безопасно. Такой подход удешевляет и ускоряет разработку и снижает количество ошибок».
- *Честно опишите затраты.* Изменить значения первичного ключа кажется простой задачей, но нужно реально оценить объем работы по вычислению

новых значений, написанию и тестированию кода для обработки дубликатов, каскадного распространения изменений в базе данных, анализу влияния на другие системы, а также обучению пользователей и администраторов управлению новыми процедурами.

Большинство руководителей в первую очередь волнуют затраты на решение задачи, и они не станут требовать микрооптимизации, узнав, чего она будет стоить.

Если менеджер все равно настаивает, спросите его, стоит ли отложить ради этого другую работу. Напомните об остальных задачах, которые придется задержать, и спросите, считает ли он изменение нумерации псевдоключей приоритетом.

- *Используйте естественные ключи*. Если менеджер или другие пользователи базы данных настаивают, чтобы значения первичного ключа имели смысл, пусть они имеют смысл. Не используйте псевдоключи — используйте строку или число, в котором кодируется некоторый идентифицирующий смысл. Тогда вам будет проще объяснить любые пропуски в контексте смысла этих естественных ключей.
- Также можно использовать псевдоключ вместе с другим столбцом атрибута, который используется в качестве естественного идентификатора. Скройте псевдоключ из отчетов, если пропуски в числовой последовательности смущают читателей.

Используйте псевдоключи как уникальные идентификаторы строк; они не являются номерами строк данных.

Мини-антипаттерн: автоматическое увеличение в группах

«Мне нужен столбец с автоматическим увеличением, но он должен заново начинаться с 1 в каждой подгруппе строк».

Это требование может принимать разные формы. Ранжирование игроков по командам или по годам либо счетов по клиентам. Независимо от причин, с ним возникают некоторые проблемы.

Во-первых, подбор новых увеличенных значений усложняется. Вставка строки должна проверить самое последнее значение, сгенерированное для той же подгруппы, что означает блокировку одновременных вставок на время проверки

текущего набора строк. Это может привести к появлению узких мест и замедлению операции вставки.

Как вариант, вставки должны создавать новые генераторы последовательностей «на ходу» для каждой новой подгруппы. Это может привести к стремительному росту количества генераторов последовательностей. Если подгруппы состоят только из одной строки, количество генераторов последовательностей будет равно количеству строк в самой таблице.

Во-вторых, может показаться, что нумерация строк в подгруппах работает как простое порядковое ранжирование. Помните, что псевдоключи должны быть уникальными, но сделать так, чтобы они использовали последовательные значения, сложнее. При удалении строк или отмене транзакций последовательные значения приходится переназначать — возможно, для многих строк.

Лучшее решение — нумеровать строки при их запросе, что легко сделать при помощи оконной функции `ROW_NUMBER()`. Это гарантирует, что вы получите последовательность порядковых целых чисел без пропусков, которая будет заново начинаться для каждой подгруппы, определяемой параметром `PARTITION BY`.

```
SELECT bug_id, author, comment,
  ROW_NUMBER() OVER (PARTITION BY bug_id
    ORDER BY comment_date) AS comment_number
FROM Comments;
```

Начинать теоретизировать до того, как имеешь
на руках все улики, — большая ошибка.

➤ *Шерлок Холмс*

ГЛАВА 23

Не вижу зла

«Я нашел *еще одну* ошибку в вашем продукте», — услышал я в трубке телефона.

Это случилось, когда я работал инженером технической поддержки продукта SQL РСУБД. У нас был один известный клиент-скандалист. Практически во всех его заявках проблема оказывалась на его стороне, а не на нашей.

«Доброе утро, мистер Дэвис. Конечно, мы готовы исправить любую проблему, о которой вы сообщили, — ответил я. — Расскажите, что произошло?»

«Я выполнил запрос к вашей базе данных и ничего не получил, — резко ответил мистер Дэвис. — Но я знаю, что данные находятся в базе, — я проверил в тестовом скрипте».

«Может, проблема с запросом? — спросил я. — API возвращает какие-нибудь ошибки или предупреждения?»

«С чего мне проверять возвращаемое значение функции API? Функция должна просто выполнить мой запрос SQL. Если с ней проблемы, значит, в вашем продукте допущена ошибка. Если бы в продукте не было ошибок, то и проблем бы не было. Я не обязан разбираться с вашими ошибками».

Я был слегка ошарашен, но решил обратиться к фактам. «Хорошо, давайте потестируем. Скопируйте и вставьте свой *точный* запрос SQL в инструмент запросов и запустите его. Что там написано?» Я ждал ответа.

«Синтаксическая ошибка в `SELCET`». После недолгой паузы он добавил: «Заявку можно закрывать» — и бросил трубку.

Мистер Дэвис был единственным разработчиком в компании, управляющей воздушным движением. Он писал ПО для сохранения данных о международных полетах. Он звонил нам каждую неделю.

Цель: сокращение объема кода

Все хотят писать *элегантный код*. Это значит, что мы хотим получать отличный результат с меньшим объемом кода. Чем лучше результат и чем меньше кода он требует, тем выше его элегантность. Если выполнить работу еще лучше не получается, остается повышать элегантность, показывая тот же результат, но с меньшим объемом кода.

Возможно, кому-то этот аргумент покажется притянутым за уши, но есть и более веские причины писать краткий код:

- Рабочее приложение создается быстрее.
- Приходится писать меньше кода для тестирования, документирования или совместного анализа.
- Чем меньше строк кода, тем меньше в нем ошибок.

Разработчики всегда стараются удалить весь код, который можно удалить, особенно если этот код не повышает качество их работы.

Антипаттерн: мартышкин труд

Антипаттерн «Не вижу зла» обычно принимает две формы: первая — игнорирование возвращаемых значений API базы данных и вторая — чтение фрагментов кода SQL, слитых с кодом приложения. В обоих случаях разработчики не используют информацию, которая лежит на поверхности.

Диагнозы без диагностики

В коде из следующего примера есть ошибки, но нет проверки ошибок.

See-No-Evil/anti/no-check.py

```
import mysql.connector

cnx = mysql.connector.connect(user='scottt', database='test') ❶

cursor = cnx.cursor()

query = '''SELCET bug_id, summary, date_reported FROM Bugs
    WHERE assigned_to = %s AND status = %s'''

parameters = (1, 'NEW')

cursor.execute(query, parameters) ❷
```

```
for row in cursor:
    print(row)
```

Код получается коротким, но в нем есть места, в которых возвращаемые функциями значения статуса могут указывать на наличие проблем. Игнорируя возвращаемые значения, вы никогда не узнаете о них.

Наверное, самая распространенная ошибка в API баз данных возникает при попытке создания подключения к базе данных, например, в точке ❶. Можно случайно ввести неправильное имя базы данных или хоста сервера, или указать неверное имя пользователя или пароль, или сервер базы данных может быть недоступен. В зависимости от языка и адаптера базы данных в таких случаях могут выдаваться исключения, завершающие выполнение скрипта. В других языках исключения не выдаются, но результатом подключения является недействительный объект.

Ошибка в имени пользователя в предыдущем примере на языке Python приводит к ошибке следующего вида:

```
mysql.connector.errors.ProgrammingError: 1045 (28000):
Access denied for user 'scottt'@'localhost'
```

(Доступ пользователю 'scottt'@'localhost' запрещен)

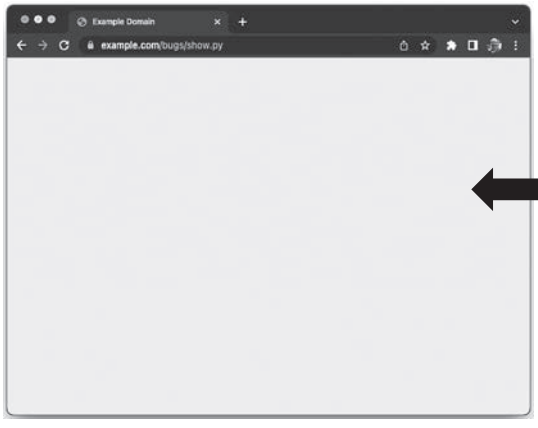
Если исправить ошибку в имени пользователя, вызов `execute()` в точке ❷ может выдать исключение из-за простой синтаксической ошибки, возникшей из-за опечатки, непарных скобок или неверно записанного имени столбца:

```
mysql.connector.errors.ProgrammingError: 1064 (42000):
You have an error in your SQL syntax; check the manual that corresponds to your
MySQL server version for the right syntax to use near 'SELCET bug_id, ...

(Ошибка в синтаксисе SQL; обратитесь к документации своей версии MySQL Server за
описанием синтаксиса, который должен использоваться с 'SELCET bug_id, ...)
```

Такие разработчики, как мистер Дэвис, отнюдь не редкость. Им кажется, что проверка возвращаемых значений и исключений не приносит никакой пользы, потому что подобные ситуации не должны возникать. Кроме того, лишний код повторяется, что портит приложение и хуже читается. Проверка ошибок, безусловно, не повышает качество кода, но увеличивает его длину, так что, к сожалению, общая элегантность снижается.

Пользователи не видят кода; они видят только вывод. Если критическая ошибка остается необработанной, пользователь может увидеть непонятное сообщение об исключении или пустой экран.

В подобной ситуации вас вряд ли утешит тот факт, что код аккуратный и краткий.



Пользователи увидят абсолютно пустой экран; ожидайте лавины звонков в службу поддержки.

Промежуточные строки

Другая частая вредная привычка, в которой кроется антипаттерн «Не вижу зла», — отладка кода приложения, в котором запрос SQL выводится в виде строки. Она усложняется тем, что в этом случае труднее наглядно представить результирующую строку SQL после применения к ней логики приложения, конкатенации и добавления данных из переменных приложения.

Попытки отлаживать программу таким образом напоминают попытки собрать пазл, не глядя на картинку на коробке.

Приведу пример вопроса, который мне часто приходится слышать от разработчиков. Следующий код строит запрос с проверкой условий, присоединяя секцию `WHERE`, если скрипт ищет конкретную ошибку, а не набор ошибок.

See-No-Evil/anti/white-space.py

```
import mysql.connector

bug_id = int(input() or '0')

cnx = mysql.connector.connect(user='scott', database='test')

cursor = cnx.cursor()

query = '''SELECT * FROM Bugs'''

parameters = tuple()

if bug_id > 0:
    query = query + '''WHERE bug_id = %s'''
    parameters = parameters + (bug_id,)

cursor.execute(query, parameters)

for row in cursor:
    print(row)
```

Почему запрос в этом примере выдает ошибку? Ответ становится очевидным, если взглянуть на полную строку запроса, полученную в результате конкатенации:

See-No-Evil/anti/white-space.py

```
SELECT * FROM BugsWHERE bug_id = 1234
```

Между `Bugs` и `WHERE` отсутствует пробел, из-за чего синтаксис запроса искажается и выглядит как чтение из таблицы с именем `BugsWHERE`, за которым следует выражение SQL в недопустимом контексте. В коде была выполнена конкатенация двух строк, не разделенных пробелом.

На отладку подобных проблем тратится очень много времени и сил, так как разработчики просматривают код, в котором строится SQL, вместо того чтобы просматривать собственно SQL.

Как распознать антипаттерн

Хотя может показаться, что обнаружить недостающий код довольно трудно, современные IDE помечают код, который игнорирует возвращаемое значение функции или не обрабатывает проверяемое исключение. Паттерн «Не вижу зла» также можно распознать по следующим фразам:

- «Когда я выдаю запрос к базе данных, программа завершается со сбоем».

  Часто сбой происходит из-за того, что запрос завершился неудачей, а вы попытались использовать результат недопустимым образом, например вызвать метод для того, что не является объектом, или разыменовать `NULL`-указатель.
- «А вы не поможете мне найти ошибку в SQL? Вот мой код...»

  Для начала изучите запрос SQL, а не код, в котором он строится.
- «Я не хочу загромождать код обработкой ошибок».

  По некоторым оценкам, до 50 % строк кода по-настоящему надежного приложения занимает код обработки ошибок. Кажется много, но только до тех пор, пока не перечислить все операции, которые можно отнести к обработке ошибок: обнаружение, классификацию, получение информации и исправление. Также для многих ошибок можно просмотреть результаты автоматизированных тестов и подсчитать, сколько модульных тестов потребуется для проверки обработки всех потенциальных ошибок. Все это важные этапы создания любого программного продукта.

Допустимые применения антипаттерна

Опустить проверку ошибок можно, если с ошибкой ничего нельзя поделать. Например, функция `close()`, вызванная для подключения к базе данных, воз-

вращает статус, но если приложение собирается завершить работу и вернуть управление, скорее всего, ресурсы этого подключения будут освобождены в любом случае.

В объектно-ориентированных языках можно инициировать исключение, не принимая на себя ответственность за его обработку. Ваш код полагается на то, что за обработку исключения отвечает код, из которого он был вызван. Тогда ваш код может разрешить исключению подняться наверх по стеку вызовов.

Решение: корректное восстановление после ошибок

Каждый любитель танцев знает, что не оступиться невозможно. Чтобы оставаться изящным, необходимо уметь незаметно выходить из положения. Не лишайте себя возможности выявить причину ошибки. Тогда вы сможете отреагировать быстро, вернувшись в ритм, пока никто не заметил вашу оплошность.

Следовать за ритмом

Проверка возвращаемых статусов и исключений из API баз данных — лучший способ убедиться в том, что вы не оступились. Код из следующего примера проверяет статус после каждого вызова, который может инициировать ошибку:

See-No-Evil/soln/check.py

```
import mysql.connector
from mysql.connector import errorcode

try:
    cnx = mysql.connector.connect(user='scott', database='test')

except mysql.connector.Error as err: # Проверка ошибок ❶
    if err.errno == errorcode.ER_ACCESS_DENIED_ERROR:
        print("Something is wrong with your user name or password")
    elif err.errno == errorcode.ER_BAD_DB_ERROR:
        print("Database does not exist")
    else:
        print(err)

cursor = cnx.cursor()

try:
    query = '''SELECT bug_id, summary, date_reported FROM Bugs
        WHERE assigned_to = %s AND status = %s'''

    parameters = (1, 'NEW')

    cursor.execute(query, parameters)
except mysql.connector.Error as err: ❷
    print(err)
```

Код в точке ❶ перехватывает исключение, выдаваемое при неудачном подключении к базе данных, и выводит сообщение об исключении. Еще лучше зарегистрировать исключение SQL в журнале, чтобы разработчик мог его просмотреть, и вывести для пользователя более понятное сообщение. Точно так же следует обрабатывать исключения при выполнении запросов, как показано в точке ❷.

Повторение шагов

Также важно использовать для отладки фактический запрос SQL вместо кода, который строит запрос SQL. Многие простые ошибки, например опечатки, непарные кавычки или скобки, в этом случае моментально становятся заметными, хотя обычно кажутся непонятными и приводят в замешательство.

- Стройте свой запрос SQL в переменной, а не в аргументах метода API. Так у вас появится возможность просмотреть содержимое переменной перед ее использованием.
- Выберите для вывода SQL место, отдельное от вывода приложения: файл журнала, консоль отладчика IDE или расширение браузера для отображения диагностического вывода.
- Не выводите запрос SQL в HTML-комментариях вывода веб-приложения. Любой пользователь может просмотреть исходный код страницы. Прочитав запрос SQL, злоумышленник получит массу информации о структуре вашей базы данных.

Использование фреймворка объектно-реляционного отображения (ORM), который строит прозрачные запросы SQL, может усложнить отладку, поскольку запрос SQL генерируется кодом ORM по ходу работы. Некоторые фреймворки ORM решают эту проблему, направляя сгенерированный SQL в журнал. Во многих языках веб-приложений программные ошибки выводятся в журнал ошибок HTTP-сервера. Выясните, какой журнал использует ваш язык в вашей среде, и просматривайте его в ходе отладки.

Наконец, многие РСУБД предоставляют собственный механизм ведения журналов на серверах баз данных вместо клиентского кода приложения. Даже если в приложении не ведется журнал SQL, запросы можно отслеживать при их выполнении сервером базы данных.

Исходите из того, что в любой строке кода может произойти сбой, а не из того, что он не произойдет. Прежде чем переходить к исправлению ошибки, соберите информацию о ее природе и причинах.

Мини-антипаттерн: чтение сообщений о синтаксических ошибках

В случае синтаксических ошибок SQL MySQL предоставляет полезную информацию: какая именно часть запроса SQL следовала за частью, в которой парсер синтаксиса зашел в тупик или ожидал чего-то другого.

В мини-антипаттерне «Зарезервированные слова» в главе 11, с. 154, было показано, как использование зарезервированного ключевого слова в неожиданном месте запроса может привести к синтаксической ошибке. Другие виды синтаксических ошибок могут возникать из-за отсутствующих ключевых слов, лишних ключевых слов или ошибок со знаками препинания.

В следующем примере должен использоваться синтаксис `ORDER BY`, но ключевое слово `BY` было пропущено.

```
SELECT * FROM Bugs ORDER date_reported;
                         ^ error starts here

ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near 'date_
reported' at line 1
```

(Ошибка в синтаксисе SQL; обратитесь к документации своей версии MySQL Server за описанием синтаксиса, который должен использоваться рядом с 'date_ reported' в строке 1)

Другой пример демонстрирует незнание того, как должна записываться секция `WHERE`; ключевое слово `WHERE` должно входить в инструкцию только один раз, а за ним должно следовать логическое выражение.

```
SELECT * FROM Bugs WHERE status = 'NEW' AND WHERE assigned_to = 123;
                                             ^ error starts here

ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near 'WHERE
assigned_to = 1' at line 1
```

(Ошибка в синтаксисе SQL; обратитесь к документации своей версии MySQL Server за описанием синтаксиса, который должен использоваться рядом с ‘WHERE assigned_to = 1’ в строке 1)

В следующем примере показан синтаксис, который вызвал трудности у парсера в самом конце запроса, потому что в нем не была закрыта скобка. Так как ошибка встречается в конце строки, далее следует пустая строка. В предыдущем примере фрагмент, следующий за ошибкой, выводится в одинарных кавычках. Так как фрагмент после конца строки является пустой строкой, в следующем

сообщении об ошибке выводятся две одинарные кавычки, между которыми ничего нет.

```
SELECT * FROM Bugs WHERE (status = 'NEW';
                                        ^ error starts here

ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that
corresponds to your MySQL server version for the right syntax to use near '' at
line 1
```

(Ошибка в синтаксисе SQL; обратитесь к документации своей версии MySQL Server за описанием синтаксиса, который должен использоваться рядом с '' в строке 1)

В каждом случае следует тщательно проверить запрос в точке, указанной в сообщении об ошибке. Так вы хотя бы примерно поймете, какую часть необходимо исправить.

Сообщения об ошибках часто трудно понять, некоторые труднее других (в частности, сообщения Oracle), — но нужно постараться выжать из них максимум полезной информации.

Если что-то не записано, то этого не было.

➤ *Известная поговорка*

ГЛАВА 24

Дипломатический иммунитет

На одном из первых мест работы я усвоил, насколько важно следовать лучшим практикам разработки, — после того, как в результате трагического происшествия я стал ответственным за важное приложение базы данных.

Я прошел собеседование на работу по контракту в Hewlett-Packard. Мне поручили разработку и сопровождение UNIX-приложения, написанного на C с HP ALLBASE/SQL. Руководитель и специалист, проводивший собеседование, с грустью сообщили, что предыдущий разработчик погиб в автокатастрофе. Больше никто в отделе не умел работать в UNIX и ничего не знал о приложении.

Взявшись за работу, я обнаружил, что разработчик не писал документацию или тесты для приложения, а также не пользовался системой контроля версий и даже не оставлял комментарии. В одном каталоге хранилось все: код работающей системы; код, находившийся в процессе разработки; и код, который уже не использовался.

В проекте было много *технического долга* — следствия желания сэкономить время вместо применения лучших практик. Технический долг повышает риск и ведет к лишней работе, пока вы от него не избавитесь, проведя рефакторинг, тестирование и составив документацию.

Я полгода приводил код в порядок и писал документацию для приложения, которое еще было довольно скромным, потому что мне приходилось тратить слишком много времени на поддержку пользователей и продолжение разработки.

Очевидно, мой предшественник никак не мог ввести меня в курс дела. Эта история наглядно показывает последствия, которыми чревато накопление огромного технического долга.

Цель: применение лучших практик

Профессионалы отрасли стараются применять в своих проектах хорошие практики разработки, в том числе:

- Хранение кода приложения в системе контроля версий, с применением таких средств, как Git или Subversion.
- Разработка и запуск автоматизированных модульных или функциональных тестов для приложения.
- Сопровождение кода документацией, спецификациями, комментариями и соблюдение единого стиля программирования для обеспечения требований, стратегий реализации, эксплуатации и сопровождения приложения.

Время, которое занимает внедрение лучших практик, не пропадает даром, поскольку оно избавляет от большого объема лишней или повторяющейся работы. Многие опытные разработчики знают, что отказ от этих практик ради ускорения — верный путь к катастрофе.

Антипаттерн: второсортный SQL

Даже разработчики, которые приветствуют лучшие практики, склонны считать, что на код базы данных эти практики не распространяются. Этот антипаттерн называется «Дипломатический иммунитет», потому что в его основе лежит убеждение, что правила разработки приложений неприменимы к разработке баз данных.

Разработчики считают так по двум причинам:

- В некоторых компаниях роли разработчика и администратора базы данных различаются. Администоратор обычно работает с несколькими командами разработчиков, поэтому создается впечатление, что он не входит ни в одну из них. К нему относятся как к гостю и не возлагают на него такие же обязанности, как на разработчиков.
- Язык SQL, используемый для реляционных баз данных, отличается от традиционных языков программирования. Даже способ выполнения инструкций SQL как специализированного языка из кода приложения предполагает своего рода гостевой статус.
- Для языков программирования приложений существуют современные инструменты IDE, с которыми редактирование, тестирование и контроль версий выполнять легко и удобно. С другой стороны, средства разработки баз данных не настолько проработаны или по крайней мере не получили столь широкого применения. Создавать приложения с применением лучших прак-

тик легко, но применять эти практики к SQL довольно неудобно. Разработчики обычно предпочитают заняться чем-нибудь другим.

- В сфере IT разбирается в базе данных и обслуживает ее один человек — администратор базы данных. Так как он единственный, у кого есть доступ к серверу базы данных, он принимает на себя роль ходячей базы знаний и системы контроля версий.

База данных составляет основу приложения, и ее качество играет важную роль. Вы знаете, как разрабатывать качественный код приложения, но можно строить приложение на основе базы данных, которая не соответствует потребностям проекта или в которой никто не разбирается. Вы рискуете разработать приложение, которое станет непригодным для использования сразу после своего создания.

Как распознать антипаттерн

Казалось бы, понять, что что-то *не* делается, довольно трудно, тем не менее это не так. Вот некоторые типичные признаки поиска коротких путей:

- «Мы переходим на новый инженерный процесс, то есть его облегченную версию».

  «Облегченный» в данном контексте означает, что команда решила пропустить часть задач, которых требует инженерный процесс. Этот пропуск может быть оправдан, но может и говорить о пренебрежении важными лучшими практиками.

- «Администраторы базы данных могут не ходить на тренинг по новой системе контроля версий, потому что они все равно ею не пользуются».

  Исключение некоторых участников группы из обучения гарантирует, что они не будут пользоваться этими инструментами (и возможно, не получат доступа к ним).

- «Как отследить, какие таблицы и столбцы содержат *данные, идентифицирующие личность* (PII, Personally Identifiable Information), или *конфиденциальную личную информацию* (SPI, Sensitive Personal Information)? Нужно подтвердить, что мы правильно работаем с конфиденциальными данными, чтобы не нарушать требований аудита и законодательства».

  Конфиденциальность данных — очень важная тема даже для небольших компаний. Если вы не ведете точную и актуальную документацию по вашей схеме базы данных, вам придется обрабатывать все данные как конфиденциальные. Это повышает затраты на проект из-за добавления работы и затрат на обеспечение совместимости и оплату носителей для хранения данных.

- «Нет ли программы, которая сравнивает две схемы баз данных, выводит сводку различий и создает скрипт для приведения одной схемы в соответствие с другой?»

  Если вы не соблюдаете процесс развертывания изменений в схемах баз данных, они могут рассинхронизироваться и соотнести их вновь будет сложно.
- «Мой код описывает себя сам».

  Этой фразой часто оправдывается отсутствие документации или комментариев в коде. Она уже превратилась в штамп, но крайне редко соответствует действительности.

Допустимые применения антипаттерна

Документацию, тесты и контроль версий можно отнести к хорошим практикам для любого кода, который планируется использовать больше одного раза. Тем не менее иногда нужно написать действительно одноразовый код, например, тест для функции API или код для подтверждения концепции или демонстрации какого-то приема коллеге.

Чтобы проверить, что код на самом деле временный, удалите его сразу после использования. Если вы не сможете заставить себя это сделать, то, скорее всего, код лучше оставить. А если его лучше оставить, его следует сохранить в репозитории контроля версий и хотя бы кратко пояснить, для чего нужен этот код и как им пользоваться.

В качестве компромисса некоторые разработчики хранят одноразовый код в специальном репозитории для кода или заметок, для которых не нашлось «официального» места.

Решение: формирование разносторонней культуры качества

Для большинства разработчиков качество равнозначно тестированию, но это всего лишь *контроль качества* — одна из составляющих процесса. Полный жизненный цикл разработки включает гарантию качества (QA, quality assurance), состоящую из трех частей:

1. Понятное изложение требований к проекту и его конечным продуктам в письменной форме.
2. Проектирование и разработка решения под изложенные требования.
3. Проверка и тестирование, подтверждающие соответствие решения требованиям.

Для надлежащей гарантии качества необходимы все три этапа, хотя в некоторых методологиях программирования не обязательно выполнять их именно в таком порядке.

QA в разработке баз данных обеспечивается применением лучших практик в *документации, контроле версий исходного кода* и *тестировании*.

Пример: документация

Самодокументируемого кода не существует. Хотя опытный разработчик способен расшифровать большую часть кода путем анализа и экспериментов, этот процесс довольно сложен (если бы код было легко читать, его бы не называли *кодом*). Кроме того, код ничего не скажет об отсутствующей функциональности или нерешенных проблемах.

Требования и реализация базы данных должны документироваться так же, как код приложения. Независимо от того, проектируете ли вы базу данных с нуля или унаследовали ее проект, соблюдайте следующие принципы документирования базы данных:

- *Диаграмма «объект — отношение»*: важнейшим видом документации базы данных является диаграмма «объект — отношение» (ER, Entity-relationship), представляющая таблицы и отношения между ними. В нескольких главах этой книги встречается упрощенная форма диаграмм «объект — отношение». На более сложных диаграммах ER используются обозначения для столбцов, ключей, индексов и других объектов баз данных.

  Некоторые пакеты для построения диаграмм включают элементы, используемые на диаграммах ER. Некоторые инструменты даже могут выполнить реверс-инжиниринг скрипта SQL или живой базы данных и построить диаграмму ER. Базы данных бывают довольно сложными и могут содержать столько таблиц, что ограничиться одной диаграммой оказывается нереально. В таком случае следует разбить ее на несколько диаграмм. Обычно удается естественным образом сгруппировать таблицы так, чтобы каждая диаграмма читалась легко, была полезной и не перегружала информацией читателя.
- *Таблицы, столбцы и представления*: вам также понадобится письменная документация для базы данных, потому что формат диаграммы ER не подходит для описания назначения и использования каждой таблицы, столбца и других объектов.

  Таблицам нужно описание типа сущности, которую моделирует таблица. Например, имена `Bugs`, `Products` и `Accounts` вполне понятны, но как насчет таблиц сопоставления, таких как `BugStatus`, таблиц пересечения, таких как `BugsProducts`, или зависимых таблиц, таких как `Comments`? Кроме того, сколько строк, по вашему мнению, будет содержать каждая таблица? Каких запросов к этой таблице можно ожидать?

У каждого столбца есть имя и тип данных, но они не сообщают читателю назначение столбца или то, какие значения имеют в нем смысл (скорее всего, не каждое значение будет допустимым для типа данных). Для столбцов, хранящих количественные значения, единица измерения понятна. Но допустим ли в столбце `NULL` и почему?

В представлениях хранятся часто используемые запросы к одной или нескольким таблицам. Чем оправдывается создание представления? Какое приложение или пользователь должны использовать представление? Предназначалось ли представление для абстрагирования сложных отношений между таблицами? Должно ли оно разрешать непривилегированным пользователям запускать предписанные запросы? Можно ли обновить представление?

- *Отношения*: ограничения ссылочной целостности реализуют зависимости между таблицами, но они не скажут вам все, что вы собираетесь моделировать при помощи ограничений. Например, столбец `Bugs.reported_by` не допускает `NULL`, а `Bugs.assigned_to` допускает. Означает ли это, что ошибку можно исправить до закрепления за ответственным исполнителем? Если нет, то когда назначать ошибку ответственному?

  В некоторых ситуациях могут существовать неявные отношения, для которых ограничения не определены. Без документации о них трудно догадаться.
- *Триггеры*: проверка данных, преобразования данных, регистрация изменений в базе данных — все это примеры задач, решаемых при помощи триггеров. Какие бизнес-правила вы реализуете в триггерах?
- *Хранимые процедуры*: документируйте хранимые процедуры как API. Какую задачу решает процедура? Вносит ли процедура какие-либо изменения в данные? Какие используются типы данных, каков смысл входных и выходных параметров? Призвана ли процедура заменить какой-то тип запроса, чтобы устранить узкое место производительности? Используется ли процедура для предоставления непривилегированным пользователям доступа к привилегированным таблицам?
- *Безопасность SQL*: какие типы пользователей базы данных определены в приложении? Какие привилегии доступа имеет каждый из этих типов? Какие роли SQL вы предоставляете, каким пользователям они доступны? Существуют ли типы пользователей для выполнения конкретных задач, таких как резервное копирование или формирование отчетов? Какие меры безопасности системного уровня применяются: например, должен ли клиент связываться с сервером РСУБД по SSL? Что делается для обнаружения и блокировки попыток незаконной аутентификации (например, подбора паролей методом брутфорса)? Проведена ли тщательная ревизия кода на предмет уязвимостей SQL-инъекций?
- *Инфраструктура базы данных*: эта информация используется в основном IT-специалистами и администраторами баз данных, но разработчикам тоже

полезно знать ее. Какая разновидность и версия РСУБД используется? Какое имя хоста используется сервером базы данных? Используются ли множественные серверы базы данных, репликация, кластеризация, прокси и т. д.? Как организована сеть, какой номер порта использует сервер базы данных? Какие параметры подключения должны использоваться клиентским приложением? Какие пароли используют пользователи базы данных? Какая политика резервного копирования выбрана для базы данных?

- *Объектно-реляционное отображение*: логика работы с базами данных из кода приложения может быть реализована в проекте на уровне классов, базирующихся на модели ORM. Какие бизнес-правила реализуются таким способом? Проверка данных, преобразование данных, ведение журнала, кэширование или профилирование?

Разработчики не любят вести инженерную документацию. Ее трудно писать и трудно поддерживать в актуальном состоянии, а когда ее почти никто не читает, это разочаровывает. Но самые закаленные, опытные разработчики знают, что базу данных нужно документировать, даже если по другим частям продукта это не делается.

Не документировать ничего, кроме базы данных

Джоэл Спольски (Joel Spolsky) — один из основателей популярного сайта помощи разработчикам Stack Overflow. В своем подкасте Stack Overflow № 80[1] он упомянул о том, что не видит особой ценности в документировании кода.

> *Джоэл:* У разработчиков складывается ощущение, что для кода, над которым им приходится работать, никогда не бывает достаточной документации, никогда. И вырабатывается стойкое нежелание ее писать вообще, так как ее почти никогда не пишут.

Он полагает, что от документации, которая не обновляется в соответствии с изменениями в коде, нет никакой пользы. А если документация бесполезна, то другие разработчики не считают нужным ее читать; получается, что ее вообще не нужно писать. Но через минуту в том же подкасте Спольски делает одно исключение. Он говорит, что важно вести документацию хотя бы по базе данных.

> *Джоэл:* Я на собственном опыте узнал, что если у вас есть база данных и вы не напишете подробную документацию по каждому ее столбцу, то через год или два ваш мир рискует рухнуть.

[1] https://stackoverflow.blog/2010/01/21/podcast-80/

Цепочка доказательств: контроль исходного кода

Как восстановить базу данных, если сервер базы данных полностью выйдет из строя? Как лучше всего отслеживать полное обновление в архитектуре базы данных? Как отменить изменение?

Мы привыкли пользоваться системой контроля версий для управления кодом, чтобы решать похожие проблемы при разработке. Проект, использующий систему контроля версий, должен включать *все* необходимое для пересборки и повторного развертывания. Система контроля исходного кода также хранит историю изменений и служит пошаговой резервной копией, чтобы любое изменение можно было отменить.

Контроль версий также следует применять с кодом базы данных, чтобы пользоваться всеми ее преимуществами для разработки. Сохраняйте в репозитории файлы, относящиеся к разработке базы данных, включая следующие:

- *Скрипты определения данных*: все РСУБД предоставляют средства для выполнения скриптов SQL, содержащих инструкции `CREATE TABLE` и другие инструкции, определяющие объекты базы данных.
- *Триггеры и процедуры*: многие проекты дополняют код приложения процедурами, хранящимися в базе данных. Скорее всего, приложение не будет работать без этих процедур, так что их можно считать частью кода проекта.
- *Начальные данные*: таблицы сопоставления могут содержать набор данных, представляющий начальное состояние базы данных до ввода данных пользователем. Следует хранить начальные данные, которые позволяют воссоздать базу данных из исходных файлов проекта, или *данные заполнения (seed data)*.
- *Диаграммы ER и документация*: строго говоря, эти файлы не являются кодом, но тесно связаны с ним. В них описываются требования к базе данных, ее реализация и интеграция с приложением. По мере эволюции проекта и изменения базы данных и приложения необходимо поддерживать актуальность этих файлов. Следите, чтобы документы описывали текущую структуру.
- *Скрипты администратора базы данных*: многие проекты содержат набор заданий обработки данных, которые выполняются за пределами приложения. К их числу относятся задачи импорта/экспорта, синхронизации, формирования отчетов, резервного копирования, проверки данных, тестирования и т. д.

Убедитесь, что файлы с кодом базы данных связаны с кодом приложения, использующим эту базу данных. Одно из преимуществ системы контроля версий заключается в том, что можно загрузить проект из репозитория по конкретному номеру версии, дате или контрольной точке, и файлы должны заработать в этом сочетании. Используйте один репозиторий как для кода приложения, так и для кода базы данных.

Средства эволюции схемы

Код управляется системой контроля версий, а база данных — нет. Фреймворк Ruby on Rails популяризировал так называемые *миграции* для управления обновлениями экземпляров баз данных, задействованных в контроле версий. Миграции автоматизируют значительную часть работы по синхронизации экземпляра базы данных со структурой, ожидаемой для заданной версии кода под контролем версии.

Чтобы разработать миграцию для изменения схемы базы данных, напишите скрипт, основанный на абстрактном классе Rails, который повышает версию миграции, поэтапно обновляя базу данных. Также напишите понижающую функцию, которая отменяет изменения, внесенные функцией обновления.

```
class AddHoursToBugs < ActiveRecord::Migration
  def self.up
    add_column :bugs, :hours, :decimal
  end

  def self.down
    remove_column :bugs, :hours
  end
end
```

При автоматическом проведении миграций инструмент Rails создает таблицу для сохранения версии (или версий), применяемых к текущему экземпляру базы данных. Например, если нужно заменить базу данных версией 5, следует передать нужную версию в аргументе инструмента миграции:

```
$ rake db:migrate VERSION=5
```

У вас накапливается набор скриптов миграции; каждый скрипт может обновить или понизить схему базы данных на один шаг.

О миграции Rails можно подробнее узнать из книги Agile Web Development with Rails 7 [Rub22] или Alembic migrations in Essential SQLAlchemy [MC15].

Другие фреймворки разработки (например, Liquibase или Flyway for Java, Doctrine for PHP, SqlAlchemy for Python или Microsoft ASP.NET) поддерживают функциональность, сходную с миграцией Rails.

Бремя доказательства: тестирование

Последняя составляющая гарантии качества — это контроль качества, или проверка, что приложение делает то, для чего оно создавалось. Многие профессиональные разработчики умеют писать автоматизированные тесты для про-

верки поведения кода приложения. Одним из важных принципов тестирования является *изоляция*, то есть тестирование только одной части системы за раз, чтобы в случае обнаружения дефекта сузить место поиска.

Изоляционное тестирование базы данных можно дополнить проверкой структуры базы данных и поведения независимо от кода. В следующем примере приведен скрипт модульного тестирования на Python:

Diplomatic_immunity/DatabaseTest.py

```
import unittest
import mysql.connector

class TestDatabase(unittest.TestCase):

    def setUp(self):
        self.cnx = mysql.connector.connect(user='scott', database='test')
        self.cursor = self.cnx.cursor()

    def test_table_bugs_exists(self):
        query = '''SELECT true FROM Bugs LIMIT 1'''
        self.cursor.execute(query)

    def test_table_bugs_column_bugid_exists(self):
        query = '''SELECT bug_id FROM Bugs LIMIT 1'''
        self.cursor.execute(query)

    # Столбец issue_id был удален, тест должен провалиться
    def test_table_bugs_column_issueid_not_exists(self):
        with self.assertRaises(mysql.connector.errors.ProgrammingError) as e:
            query = '''SELECT issue_id FROM Bugs LIMIT 1'''
            self.cursor.execute(query)

if __name__ == '__main__':
    unittest.main()
```

При написании тестов для базы данных можно руководствоваться следующим контрольным списком:

- *Таблицы, столбцы, представления*: убедитесь, что таблицы и представления, которые должны существовать в базе данных, действительно существуют. Каждый раз, добавляя в базу данных новую таблицу, представление или столбец, добавляйте тест, который подтверждает присутствие этого объекта. Также можно использовать *негативные тесты*, которые подтвердят, что удаленные из текущей версии проекта таблица или столбец действительно отсутствуют в базе.
- *Ограничения*: еще одна ситуация для негативного тестирования. Попробуйте выполнить инструкцию `INSERT`, `UPDATE` или `DELETE`, приводящую к ошибке из-за ограничения. Например, нарушьте ограничения недопустимости `NULL`,

уникальности или внешнего ключа. Если команда не возвращает ошибку, значит, ограничение не работает. Это тестирование позволит выявить многие ошибки на ранней стадии.

- *Триггеры*: триггеры также могут обеспечивать соблюдение ограничений. Они могут выполнять каскадные эффекты, преобразовывать значения, регистрировать изменения в журнале и т. д. Чтобы протестировать эти сценарии, выполните инструкцию, которая порождает триггер, а затем запрос, подтверждающий, что триггер выполнил необходимое действие.
- *Хранимые процедуры*: тестирование процедур в базе данных ближе к традиционному модульному тестированию кода приложения. Хранимая процедура получает входные параметры, что может привести к ошибкам при попытке передачи значений, лежащих за пределами допустимого диапазона. Логика в теле процедуры должна поддерживать несколько путей выполнения. Процедура может возвращать одно значение или результирующий набор запроса в зависимости от входных данных и состояния данных в базе. Кроме того, процедура может иметь *побочные эффекты* в виде обновления базы данных. Все эти характеристики процедур можно протестировать.
- *Начальные данные*: даже предположительно пустая база данных обычно содержит какие-то исходные данные, например, в таблицах сопоставления. Можно выполнить запросы, подтверждающие наличие исходных данных.
- *Запросы*: код приложения чередуется с запросами SQL. Запросы можно выполнить в тестовой среде, чтобы проверить правильность их синтаксиса и результатов. Убедитесь, что результирующий набор включает имена столбцов и типы данных, которые вы ожидаете, как и при тестировании таблиц и представлений.
- *Классы ORM*: как и триггеры, классы ORM содержат логику, включая проверку данных, преобразования или отслеживание состояния. Код абстрагирования базы данных на основе ORM следует тестировать так же, как и любой другой код приложения. Убедитесь, что эти классы выполняют ожидаемые действия со входными данными, и проверьте, что они отклоняют недействительный ввод.

Если какие-то тесты не проходят, возможно, приложение использует неверный экземпляр базы данных. Предположим, вы собирались подключиться к промежуточной базе данных, но случайно настроили тесты для подключения к рабочей базе данных, или к реплике базы данных, или к тестовой базе данных, в которой применены еще не все изменения схемы. Перепроверьте конфигурацию, исправьте ее при необходимости и попробуйте снова. Если вы уверены, что подключение настроено верно, но базу данных необходимо изменить, запустите *скрипт миграции* (см. врезку «Средства эволюции схемы», с. 311) для синхронизации этого экземпляра базы данных и приведения его к ожидаемому виду.

Работа в нескольких ветвях

Создавая приложение, вы можете работать с несколькими версиями кода. Иногда даже с разными версиями в один день. Например, можно внести срочное исправление в развертываемой ветви, а затем продолжить разработку в основной ветви.

База данных, используемая приложением, не участвует в системе контроля версий. Каждый раз создавать и уничтожать базу данных непрактично, даже если используемая РСУБД относительно динамична и проста.

Идеальный вариант — создавать отдельный экземпляр базы данных для каждой разрабатываемой, тестируемой или развертываемой версии приложения. Также каждому разработчику в команде проекта необходим отдельный экземпляр базы данных, чтобы его работа не мешала другим.

Параметры подключения к базе данных должны задаваться таким образом, чтобы при работе с любой версией приложения можно было выбрать используемую базу данных без переписывания кода.

Сегодня все РСУБД — как коммерческие, так и с открытым исходным кодом — предоставляют бесплатное решение для разработки и тестирования. Облачные сервисы и технологии контейнеризации (такие, как Docker) позволяют каждому разработчику запустить виртуальный сервер с минимальными затратами и использовать его для тестирования. Ничто не мешает разработчику проводить разработку и тестирование в полностью функциональной среде, которая соответствует рабочей.

Лучшие практики разработки, включая документирование, тестирование и контроль версий, должны применяться к базе данных точно так же, как к коду приложения.

Мини-антипаттерн: переименование

Задумав переименовать уже используемую таблицу, вы столкнетесь с классической проблемой «курица или яйцо».

Если начать с переименования таблицы, то приложение выдаст ошибку, потому что в запросе все еще используется старое имя. Вы понимаете, что код нужно обновить для использования нового имени, и развертываете приложение заново, но не можете сделать это за секунды. Развертывание обычно занимает несколько минут, а в это время приложение продолжает генерировать ошибки. И наоборот, если начать с обновления и развертывания кода, необходимо будет

изменить имя таблицы в базе данных — тоже с ограниченными временными рамками.

Если приложение развертывается на нескольких серверах, обновления кода будут развертываться на каждом сервере независимо от других, так что несколько версий приложения могут активно сосуществовать на разных серверах.

Если у вас есть возможность прервать обслуживание клиентов приложения до того момента, когда можно будет выполнить переименование таблицы и изменение кода, проблем нет. Однако современный бизнес требует, чтобы приложения работали без простоев.

Сталкиваясь с таким уровнем сложности, многие разработчики понимают, что переименование таблиц или столбцов создает еще больше проблем, чем добавление таблиц, столбцов или индексов в используемую таблицу, и решают, что переименование не стоит усилий и риска простоев.

Если переименование уже не ограничивается личными предпочтениями или особенностями стиля, расходы на него могут быть оправданными:

- Прежнее имя таблицы нарушает закон, и его использование влечет юридические последствия для организации.
- В прежнем имени таблицы упоминается название технологии, компании-партнера или корпоративного бренда, которые больше не используются. Возможно, после поглощения компании имя нужно будет изменить.
- Прежнее имя таблицы конфликтует с товарным знаком, и владельцы товарного знака требуют прекратить его несанкционированное использование.
- Прежнее имя слишком сильно напоминает другое слово, используемое в проекте или компании. Например, приложение с именем `Raffle` стоит переименовать, поскольку в той же компании существует другое приложение с именем `Rattle` и использование двух имен создаст путаницу.

При изменении имен столбцов необходимо исходить из тех же соображений. Для задачи переименования таблицы или столбца с минимальным временем недоступности приложения существуют возможные решения, причем каждое из них требует тщательного планирования и тестирования.

Первый вариант — переименование таблицы или столбца в новой таблице с последующим постепенным переводом кода приложения на новую таблицу. В этом случае развертывание кода может происходить автономно и без простоев.

1. Создайте таблицу с измененным именем таблицы или столбца.
2. Измените код так, чтобы все операции записи применялись как к старой, так и к новой таблице, но запросы обращались только к старой таблице —

единственной содержащей полностью актуальные данные. Проведите развертывание кода.

3. Постепенно скопируйте все данные из старой таблицы в новую.
4. Измените код, чтобы чтение выполнялось из новой таблицы, а запись по-прежнему выполнялась в обе таблицы. Проведите развертывание кода.
5. Измените код, прекратив запись в старую таблицу. Проведите развертывание кода.
6. Удалите старую таблицу.

Второй вариант — использование *представления*. Присвойте таблице новое имя на свое усмотрение, а затем практически одновременно создайте представление с прежним именем, которое служит своего рода «псевдонимом» для бывшего имени таблицы. Скорее всего, приложение сможет использовать представление для большинства типов запросов, даже `INSERT` или `UPDATE`. Необходимо провести тщательное тестирование и проверить, что запросы работают с представлениями так же, как они работали с базовой таблицей. Когда переключение имени будет успешно завершено, код приложения можно изменить в удобное время, чтобы обращаться к таблице по новому имени.

У людей аллергия на изменения. Они любят повторять: «Мы всегда так делали». Я пытаюсь с этим бороться. Вот почему часы у меня на стене идут против часовой стрелки.

➤ *Контр-адмирал Грейс Мюррей Хоппер (Grace Murray Hopper)*

ГЛАВА 25

Стандартные рабочие процедуры

Я консультировал одну начинающую компанию по вопросам производительности SQL. Компания постаралась привлечь лучших разработчиков PHP в своем городе. А архитектура использовала с десяток серверов приложений PHP с распределением нагрузки и один сервер, выделенный под размещение базы данных MySQL.

В компании возникла проблема, за решением которой она обратилась ко мне: сервер базы данных был хронически перегружен, тогда как серверы, на которых выполнялся код PHP, практически простаивали. Их приложение часто ожидало отклика базы данных. Владельцы компании хотели, чтобы я повысил производительность сервера базы данных.

Прежде всего я проанализировал журнал запросов базы данных, чтобы найти самые частые запросы и оптимизировать их. Из журнала я узнал, что в большинстве запросов хранимые процедуры выполнялись инструкциями вида `CALL ListCustomersProc();`. Эти вызовы занимали слишком много времени. Они должны были занимать не более 50 миллисекунд, но занимали в 20 раз больше, нередко более секунды.

Я спросил, могу ли я просмотреть код хранимых процедур. «Конечно, — сказал менеджер. — Я попрошу ведущего разработчика объяснить код одной из наших хранимых процедур, потому что он единственный, кто этот код понимает. Нам пришлось изрядно потрудиться над разработкой и оптимизацией кода хранимых процедур в MySQL».

В хранимых процедурах содержалось много сложного и неэффективного кода. В компании строили запросы из фрагментов и выполняли их как динамический SQL.

«Проблемы с производительностью возникли из-за кода процедур, — объяснил я. — Вам следует перенести часть логики построения запросов в PHP. Тогда

рабочая нагрузка будет распределена между серверами приложений, вместо того чтобы выполняться на сервере базы данных в хранимых процедурах. А поскольку ваша команда лучше разбирается в PHP, чем в языке хранимых процедур MySQL, она быстрее напишет код, подготовит модульные тесты и использует вспомогательные функции для генерирования похожих запросов».

«Но в начале проекта было решено, что весь код SQL должен быть реализован в хранимых процедурах», — сказал разработчик.

«Почему? — спросил я. — У вас в команде полно опытных разработчиков PHP, а ваш уровень PHP обслуживает множество отлично масштабируемых серверов. Зачем вам писать код на незнакомом языке и выполнять его только на одном сервере?»

«Нам сказали, что приложения баз данных должны разрабатываться именно так, — ответил разработчик. — Основатель компании сказал, что он делал так в других проектах с Oracle PL/SQL или Microsoft SQL Server T-SQL».

Цель: использование хранимых процедур

Хранимые процедуры поддерживаются в SQL, чтобы разработчики могли сохранять свой код в самой базе данных и вызывать эти процедуры из клиентских приложений.

В следующем примере представлена хранимая процедура MySQL, которая закрывает все незавершенные ошибки для заданного продукта. Логика реализована в процедуре так, чтобы ее можно было выполнять как обновление, но по особым правилам: это действие должно закрывать только ошибки со статусом `NEW` или `OPEN`. Если бы приложение использовало прямой доступ для обновления таблицы, оно могло бы изменить статус неподходящей ошибки.

Procedures/anti/close-bugs.sql

```
CREATE PROCEDURE CloseUnresolvedBugsForProduct(
  IN given_product_id BIGINT UNSIGNED)
BEGIN
  START TRANSACTION;
  UPDATE Bugs JOIN BugsProducts USING (bug_id)
  SET status = 'WONTFIX'
  WHERE product_id = given_product_id
  AND status IN ('NEW', 'OPEN');
  COMMIT;
END
```

Разработчику клиентского приложения достаточно вызвать процедуру и передать один идентификатор продукта в аргументе. Ему не нужно знать, как писать SQL для правильного обновления данных.

Procedures/anti/close-bugs.sql

```
CALL CloseUnresolvedBugsForProduct(1234);
```

На заре существования баз данных SQL разработка велась не так, как в наши дни. Из-за особенностей языков и парадигм программирования, использовавшихся в то время, всю бизнес-логику, относящуюся к базам данных, нужно было консолидировать в единый интерфейс для предотвращения дублирования кода. Процедуры, реализующие эту бизнес-логику, было логично хранить в базе данных, где они могли вызываться любым клиентским приложением. Таким образом обеспечивалась некоторая защита от того, что каждый клиент реализует эту логику по-своему.

Кроме того, администратор базы данных может быть единственным в команде, кто умеет грамотно писать запросы SQL, и именно ему поручают реализацию запросов в хранимых процедурах. Тогда все клиенты вызывают эти процедуры с правильными аргументами и полагаются на то, что работа с базой данных выполняется эффективно и последовательно.

Язык процедур SQL, который использовали администраторы баз данных в то время, назывался PL/SQL. Он больше напоминал такие старые процедурные языки, как Algol или Pascal, а не современные объектно-ориентированные, функциональные или скриптовые языки, популярные в наши дни.

Эта модель разработки не обязательно подходит для современных проектов. Часто в проектах нет специалиста-администратора баз данных, более опытного в написании эффективных запросов SQL, чем другие разработчики. Дублирование кода несет меньше риска, поскольку объектно-ориентированные фреймворки или фреймворки функционального программирования часто используются для реализации *уровня доступа к данным* (DAL, Data Access Layer).

Использование хранимых процедур не является целью само по себе; оно обусловлено выбором реализации. Однако кое-кто до сих пор полагает, что хранимые процедуры нужно использовать, потому что «мы всегда так делали».

Антипаттерн: делай как я

Использование хранимых процедур само по себе не является антипаттерном этой главы. Антипаттерном становится использование любой технологической возможности (хранимые процедуры — всего лишь хороший пример) только потому, что так принято. Из того, что хранимые процедуры хорошо подходили для какого-то проекта в прошлом, вовсе не следует, что они подойдут для нового проекта.

Хранимые процедуры (к которым также относятся функции, триггеры и пакеты) являются традиционной частью разработки приложений, если вы используете одну из основных коммерческих платформ баз данных SQL, например Oracle,

Microsoft SQL Server, IBM DB2, Informix или Sybase. Тем не менее использование хранимых процедур влечет некоторые скрытые расходы.

Язык процедур

Несмотря на тот факт, что ANSI/ISO SQL определяет стандартный язык хранимых процедур, каждая РСУБД реализует синтаксис по-своему, в результате чего процедуры теряют портируемость. Например, если вы разработали процедуру для Microsoft SQL Server, а затем переключили проект на MySQL (или наоборот), вам все равно придется переписывать каждую процедуру. Разработчикам нужно знать конкретный синтаксис, поддерживаемый обеими РСУБД, а также встроенные функции и другие идиомы кода процедур. Им придется скрупулезно анализировать предполагаемую логику каждой процедуры и думать над тем, как лучше реализовать эквивалентную логику на языке процедур другой РСУБД.

Многие диалекты SQL реализуют расширения стандартного языка хранимых процедур, чтобы обеспечить поддержку объектно-ориентированных средств, пакетов, библиотек встроенных функций или типов данных для массивов или коллекций. К сожалению, эти расширения реализуются нестандартным образом, так что каждая реализация отличается от других.

Разработка и развертывание

В наши дни в распоряжении разработчиков появилось множество современных редакторов кода и продуктов IDE. Некоторые из них распространяются с открытым кодом (как IntelliJ Community Edition, Eclipse, NetBeans или Microsoft Visual Studio). Другие относятся к категории коммерческих средств разработки (например, Apple Xcode или различные продукты JetBrains).

Разработчики, использующие отладчик для клиентских языков (таких, как Java или Python), привыкли, что у них есть возможность расставить точки прерывания и динамически просматривать содержимое переменных. Лишь немногие популярные прикладные IDE поддерживают хранимые процедуры SQL. Для разработки и отладки хранимых процедур приходится использовать более специализированные инструменты, такие как Oracle SQL Developer, Microsoft SQL Server Management Studio или Toad от Quest Software.

Развертывание хранимых процедур отличается от развертывания приложений. Развертывание процедуры выглядит как создание процедуры инструкцией `CREATE PROCEDURE`. Для внесения изменений в развернутый код в разных СУБД существуют разные инструкции: Oracle, IBM DB2, Informix и PostgreSQL используют `CREATE OR REPLACE PROCEDURE`; Microsoft SQL Server — `CREATE OR ALTER PROCEDURE`.

Развертывание проекта с жесткими требованиями высокого уровня доступности особенно проблематично. Клиентские приложения можно развернуть без простоев, потому что приложения работают на многих серверах приложений. По мере развертывания можно перезапускать приложения по одному экземпляру за раз, а остальные экземпляры будут продолжать обслуживать запросы. Для единственной реализации базы данных избыточных экземпляров не существует. Выполнение `CREATE OR REPLACE PROCEDURE` для изменения кода процедуры на занятом сервере базы данных должно ожидать, пока все клиенты, выполняющие эту процедуру, завершат ее, а затем блокирует все последующие вызовы процедуры. Если выполнение процедуры занимает достаточно много времени, клиентам придется ждать, и вся база данных будет казаться недоступной.

Как описано во врезке «Средства эволюции схемы» главы 24, с. 311, в некоторых приложениях изменения в базе данных можно провести с использованием средств миграции. Фреймворк приложений обычно поддерживает специальный формат конфигурации или API для определения таблиц, столбцов и индексов, чтобы разработчикам было проще создавать объекты базы данных без синтаксических ошибок. К сожалению, большинство средств миграции не поддерживают синтаксис хранимых процедур на таком уровне. В лучшем случае они делегируют хранимые процедуры общим средствам выполнения сценариев SQL, а разработчик должен закодировать инструкции `CREATE PROCEDURE` в скрипте SQL. Подобным же образом средства миграции предоставляют путь к обратным, или «понижающим», изменениям таблиц, столбцов или индексов, но не хранимых процедур.

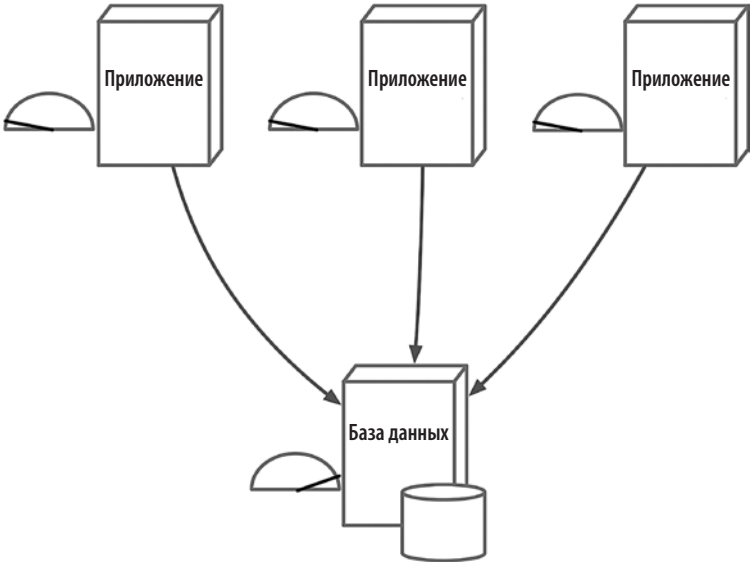
Производительность и масштабируемость

Выполняемый код использует ресурсы сервера, на котором он выполняется. Сервер базы данных должен парсить и выполнять запросы SQL, а это тоже требует ресурсов. Хранимые процедуры также выполняются на сервере базы данных, и чем сложнее код, тем больше вычислительных ресурсов он требует от сервера. Если вы реализовали в хранимых процедурах большой объем сложного кода и клиенты часто выполняют процедуры, процессор на сервере базы данных может быть перегружен, в результате чего все одновременные вызовы процедур и запросы будут выполняться слишком медленно.

В это время клиенты, вызвавшие эти процедуры, простаивают, ожидая, когда процедуры завершатся и вернут свои результаты. Пока клиенты ожидают, ресурсы процессора на серверах приложений остаются недозагруженными. У этих серверов достаточно вычислительной мощности, которая могла бы использоваться эффективно, но вместо этого они просто ждут. На диаграмме ниже изображены серверы приложений с низкой активностью (шкала со стрелкой у левого края), подключенные к одному серверу базы данных со слишком вы-

сокой нагрузкой. При такой высокой нагрузке сервер базы данных становится узким местом всей системы, потому что он не справится с дополнительным повышением нагрузки.

В случае с Oracle существует исключение: процедуры PL/SQL могут выполняться на любом сервере приложений как часть Oracle Forms. В остальных случаях процедуры PL/SQL выполняются на сервере базы данных.



Некоторые реализации SQL предоставляют компилятор для хранимых процедур, чтобы они могли выполняться более эффективно. Процедура может вызываться многократно разными клиентами до ее следующего изменения. После изменения кода процедуру необходимо перекомпилировать (в этом отношении она напоминает все остальные компилируемые языки, такие как Java или C++). Если индексы или данные изменяются так, что это повлияет на оптимизатор запросов, возможно, процедура потребует перекомпиляции. Из-за этого становится трудно понять, когда нужно перекомпилировать процедуру, а если перекомпиляция будет неверной, это может привести к снижению эффективности оптимизации запросов.

В зависимости от конкретной РСУБД сообщения о синтаксических ошибках могут выводиться сразу при определении процедуры. Другие продукты не проверяют синтаксис до первого выполнения процедуры. Все эти различия создают путаницу. Если вы забудете протестировать процедуры, вы узнаете об ошибках только на более позднем этапе — возможно, уже после развертывания процеду-

ры в рабочей среде. Обязательно изучите особенности разработки и тестирования хранимых процедур в используемой РСУБД.

С масштабируемостью тоже возникают проблемы. База данных может дорасти до такого размера, что она потребует разбиения, и сегменты данных будут распределены по нескольким серверам. Однако хранимая процедура выполняется на одном сервере базы данных, так что она может обращаться к данным только на том же сервере. PostgreSQL, Oracle и Microsoft SQL Server поддерживают функциональность, позволяющую процедуре или запросу, выполняемым на одном сервере, обращаться к данным, находящимся на удаленном сервере (это может называться по-разному — *обертка внешних данных* (foreign data wrapper), *Database Link* или *Linked Server*). Однако это требует настройки конфигурации связи с серверами: для запросов SQL и транзакций устанавливаются ограничения и повышается риск возникновения проблем с соединением серверов.

Как распознать антипаттерн

Несколько примеров фраз, которые могут произносить ваши коллеги в ходе разработки проекта с базой данных (или даже вы сами):

- «Откуда при создании хранимой процедуры взялась синтаксическая ошибка в `DECLARE`?»

  Различия в синтаксисе и использовании инструкций хранимых процедур между разными РСУБД могут создавать сложности. Например, разработчики, использующие MySQL после работы с другими РСУБД, часто испытывают проблемы, потому что `DECLARE` может использоваться только в начале главного блока `BEGIN...END`-процедуры, а локальные переменные, объявленные этим способом, не используют знак `@`. Тщательно изучите документацию обеих РСУБД, чтобы понять, как преобразовать код процедур при переходе.

- «Какой инструмент использовать для миграции более 500 хранимых процедур с T-SQL на MySQL?»

  Разработчик думает, что существует инструмент, способный переписать произвольную процедуру T-SQL в процедуру MySQL, и ошибается (это справедливо и для любой другой пары РСУБД с SQL).

- «Мы всегда используем процедуры, потому что они повышают производительность».

  Не стоит внедрять столь категоричные политики, потому что вряд ли существуют средства, обеспечивающие высокую производительность во всех ситуациях. Чтобы провести полноценную оптимизацию, разработчик должен анализировать архитектуру для каждого конкретного случая, а не руководствоваться универсальными правилами.

Допустимые применения антипаттерна

Как уже говорилось, сами по себе хранимые процедуры не являются антипаттерном. Это особенно справедливо для более зрелой, полнофункциональной и высокопроизводительной РСУБД, чем MySQL.

В некоторых граничных случаях хранимая процедура становится лучшим решением. Например, если сеть между клиентским приложением и сервером базы данных работает недостаточно быстро, а для решения задачи требуется выполнять запросы SQL поэтапно, получая промежуточные результаты, реализация кода в виде хранимой процедуры по крайней мере устраняет задержку, обусловленную круговой передачей данных по сети.

К задачам, которые запускаются редко или не требуют клиентского приложения, также можно применять хранимые процедуры. Процедуры иногда используются для таких задач администрирования баз данных, как аудит привилегий, очистка кэшей или журналов, измерение производительности или эффективности использования ресурсов или запуск запланированных задач.

Инкапсуляция запросов SQL, требующих повышенных привилегий, — еще один подходящий сценарий для применения хранимых процедур. Администратор базы данных может предоставить привилегии самой процедуре, а затем предоставить пользователям привилегии для запуска этих процедур. Это даст пользователям больше свободы действий, так как они смогут запускать процедуры, выполняющие чувствительные операции, заранее определенным образом. Работа с конфиденциальными данными (PII или SPI) в хранимой процедуре избавляет от риска атак с перехватом сетевого трафика.

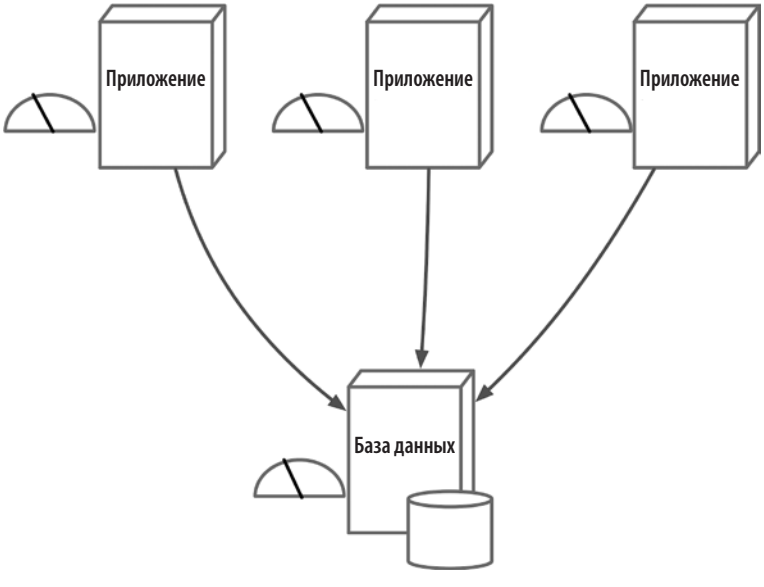
Решение: переход на современные архитектуры приложений

Клиентские языки программирования прошли долгий путь с момента появления хранимых процедур в SQL. Выбор такого языка, как Java, Python или Go, для выполнения запросов исключает упомянутые ранее недостатки. Безусловно, разработчики лучше знают свой любимый язык и работают на нем эффективнее, чем на любом языке хранимых процедур базы данных. Современные редакторы кода, отладчики и тестовые фреймворки позволяют каждому работать привычным образом. Например, применение заглушек ускоряет тесты и повышает их стабильность. Современная архитектура развертывания использует множественные серверы приложений, что позволяет выполнять круговые (round robin) или сине-зеленые (blue/green) развертывания, используя динамический балансировщик нагрузки, чтобы предотвратить передачу запросов каждому экземп-

ляру приложения на время перезапуска. Это позволяет развертывать изменения в приложениях без простоев.

Для более эффективного распределения нагрузки необходимо реализовать большую часть логики в коде приложения, используя вычислительные ресурсы на серверах приложений. Сами запросы SQL все равно должны выполняться централизованно на сервере баз данных, но весь код, выполняемый между запросами SQL (например, для форматирования запросов и обработки результатов), лучше выполнять на серверах приложений.

На следующей диаграмме изображены серверы приложений и сервер базы данных, которые мы уже видели в этой главе, с более равномерным распределением нагрузки по серверам. Все серверы работают эффективно, обеспечивают лучшую среднюю производительность и справляются со случайными выбросами нагрузки, не становясь узким местом системы.



На диаграмме показана архитектура «клиент — сервер», но можно использовать и более сложную многоуровневую архитектуру. Преимущества те же: проще масштабировать нагрузку по нескольким серверам, а не накапливать ее в хранимых процедурах на сервере базы данных.

Использование хранимых процедур — лишь один пример устоявшихся способов разработки. Вместо того чтобы вырабатывать привычки или традиции, проверяйте предположения, применяйте инженерный подход к принятию решений,

касающихся архитектуры, и выбирайте инструменты и технологию, подходящие для конкретного проекта.

Если у вас нет ничего, кроме молотка, то все задачи кажутся гвоздями. Расширяйте свой инструментарий и выбирайте для задачи подходящее средство.

Мини-антипаттерн: хранимые процедуры в MySQL

Реализация хранимых процедур в MySQL появилась в версии 5.0 в 2005 году. Особого спроса на нее не было, потому что многие разработчики, использовавшие MySQL в то время, предпочитали выполнять запросы SQL из кода приложения или классов ORM (вместо использования хранимых процедур). Из-за этого использование процедур в MySQL создавало и другие проблемы в дополнение к тем, что были описаны в предыдущем разделе. Даже разработчик, привыкший пользоваться хранимыми процедурами в других РСУБД на основе SQL, должен изучить этот раздел, прежде чем принимать решение о применении процедур в MySQL.

Использование пакетов

У хранимых процедур MySQL отсутствует поддержка пакетов, модулей или объектно-ориентированных средств. Все это усложняет организацию больших коллекций процедур или развертывание процедур из нескольких источников.

Отладка

В других РСУБД на основе SQL присутствуют специализированные инструменты разработчика, но для MySQL нет IDE или инструмента разработчика с поддержкой хранимых процедур. Разработчики не могут устанавливать точки прерывания или проверять состояние локальных переменных напрямую. Некоторые редакторы пытались имитировать отладку, вставляя строки кода в процедуру для отслеживания состояния переменных, но это требует изменения кода, что затрагивает всех клиентов, вызывающих процедуру.

Тестирование

Модульное тестирование кода подразумевает изоляцию кода от других программных компонентов и выполнение его в управляемой среде. Тестируемый код может вызывать другие функции; в таких случаях используются объекты-заглушки для моделирования вызываемых функций и тестирования вызовов. Хранимые про-

цедуры MySQL не могут выполняться в управляемых средах, они выполняются только на сервере базы данных. Они также не поддерживают вызовы функций через интерфейсы заглушек; если в них используются инструкции SQL, эти инструкции также могут обращаться к реальным таблицам или процедурам. Фактически не существует стандартной поддержки модульного тестирования хранимых процедур MySQL, а все инструменты тестирования придется разрабатывать самостоятельно. Можно выполнять системное тестирование, вызывая хранимые процедуры, но это придется делать на реальном экземпляре MySQL Server, а процедуры обращаются к реальным данным в ходе тестирования.

Компиляция

MySQL не сохраняет компилируемые версии процедур. Каждый сеанс компилирует процедуру при первом использовании, но эта компилируемая версия не используется другими сеансами. Когда сеанс клиента завершается, компилируемые версии вызываемых процедур уничтожаются. Такая реализация приводит к значительным расходам при вызове процедур, если сеансы имеют короткий срок жизни, как в большинстве веб-приложений.

Развертывание

В MySQL не поддерживается обновление кода процедуры без риска простоев. Чтобы развернуть изменение хранимой процедуры, разработчик должен сначала выполнить `DROP PROCEDURE`, а затем `CREATE PROCEDURE` с измененным кодом. Эти этапы не могут выполняться атомарно; в промежутке между этапами наступает момент, когда процедура удаляется, а база данных возвращает ошибку клиенту, пытающемуся вызвать ее в этот момент. Это означает, что при использовании одного сервера базы данных развертывание процедуры неизбежно вызовет хотя бы непродолжительный простой. Если процедура вызывается достаточно редко, между ее удалением и повторным созданием может пройти мало времени, и никто этого не заметит. Но если приложение вызывает процедуры при каждом запросе, количество неудачных вызовов процедур может исчисляться сотнями, даже если вы приложите все усилия, чтобы создать новую процедуру как можно быстрее.

Использование шардированной архитектуры

MySQL предоставляет ограниченную поддержку таблиц, доступ к которым осуществляется через Linked Server (для этого требуется специальное ядро хранения данных с именем `FEDERATED`), поэтому запускать хранимые процедуры в шардированных архитектурах неудобно. В одном из возможных обходных решений клиент заранее знает, на каком сервере хранится необходимый сегмент данных, и вызывает процедуру с использованием сеанса, подключенного к соответствующему серверу базы данных. В другом обходном решении клиент

должен вызывать ту же процедуру со всеми шардами просто на случай, если части данных существуют в нескольких шардах. Затем клиент загружает результаты всех вызовов, часть из которых могут быть пустыми.

Некоторые проекты реализуют распределенные запросы с использованием промежуточных программных средств. Эти средства предназначены для запросов к таблицам. Они могут не поддерживать вызовы хранимых процедур. Например, PlanetScale Vitess 14 — отличная технология для поддержки шардированных архитектур с MySQL, но она не поддерживает хранимые процедуры, выдающие запросы к шардированным пространствам ключей[1].

В общем случае при использовании MySQL вместо написания хранимых процедур лучше включать в клиентские приложения код для выполнения отдельных запросов. Исключение составляют случаи, когда необходимо сократить влияние сетевой задержки между приложением и базой данных, если запросов много и они выполняются один за другим. Тем не менее обычно лучше всего запускать приложение в одной быстрой сети с сервером базы данных.

[1] https://vitess.io/docs/14.0/reference/compatibility/mysql-compatibility/

ЧАСТЬ V

Дополнение: мини-антипаттерны внешних ключей

Мне часто приходилось отвечать на вопросы о внешних ключах. В результате удалось выявить в их отношении несколько дополнительных мини-антипаттернов, и мы решили объединить их в отдельную часть книги. Она разделена на две главы: мини-антипаттерны, относящиеся к использованию внешних ключей вообще, и мини-антипаттерны, относящиеся к использованию внешних ключей в MySQL.

Слишком много людей ломаются, даже не подозревая о том,
насколько близки к успеху они были в тот момент,
когда упали духом.

➤ *Томас Эдисон (Thomas Edison)*

ГЛАВА 26

Ошибки внешних ключей в стандартном SQL

Внешние ключи являются частью стандарта ANSI/ISO SQL, так что многие ошибки внешних ключей не зависят от конкретной используемой разновидности РСУБД с SQL. В этой главе представлена подборка мини-антипаттернов, относящихся к таким типам ограничений внешнего ключа, составленная на основе вопросов, часто задаваемых разработчиками на открытых интернет-форумах.

Каждый пример сопровождается описанием ошибки, возвращаемым MySQL 8.0. Сообщения об ошибках могут быть другими в других версиях MySQL или в других РСУБД. После описания ошибки и других ее последствий приводится или описывается правильный способ реализации внешнего ключа.

В следующих примерах строка таблицы `Parent` может содержать ноль, одну или несколько связанных строк из таблицы `Child`. Строка в таблице `Child` должна быть связана ровно с одной строкой таблицы `Parent`. Таким образом, между таблицами существует отношение «один ко многим» («многие» также могут быть нулем или единицей). Эти таблицы не моделируют реальные отношения в семьях, которые обычно более сложны.

Изменение направления ссылок

Может быть трудно понять, какая таблица должна иметь ограничение внешнего ключа. Если разработчик представляет отношения «один ко многим» как «у одного родителя (`Parent`) может быть несколько детей (`Child`)», он может предположить, что внешний ключ должен быть определен в таблице `Parent`.

Foreign-Key-Checklist/has-many-reversed.sql

```
CREATE TABLE Child (
  child_id INT PRIMARY KEY
);

CREATE TABLE Parent (
  parent_id INT PRIMARY KEY,
  child_id INT NOT NULL,
  FOREIGN KEY (child_id) REFERENCES Child(child_id)
);
```

Это ошибка. Одна строка `Parent` может содержать только одно значение в столбце `child_id`, так что определение ограничения внешнего ключа в таблице `Parent` означает, что заданная строка в `Parent` может иметь только одну связанную строку в `Child`, но ноль, одна или много строк `Parent` могут ссылаться на одну строку `Child`. Такое направление противоположно требуемому. Ошибка не выдается, но такую структуру нельзя использовать для хранения данных с нужным отношением.

Пожалуй, отношение «один ко многим» лучше представить как «`Child` принадлежит `Parent`» и определить внешний ключ в таблице `Child`. Это позволяет `Parent` иметь ссылки из нуля, одного или нескольких `Child`, тогда как каждая строка `Child` должна ссылаться ровно на одну строку `Parent`.

Foreign-Key-Checklist/has-many-correct.sql

```
CREATE TABLE Parent (
  parent_id INT PRIMARY KEY
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id INT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);
```

Помните: ограничение внешнего ключа должно определяться в таблице, находящейся на стороне «многих» в отношении «один ко многим».

Ссылки на еще не созданные таблицы

Если внешний ключ ссылается на еще не созданную таблицу, выдается сообщение об ошибке.

Foreign-Key-Checklist/table-order-error.sql

```
CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id INT NOT NULL,
```

```
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);

CREATE TABLE Parent (
  parent_id INT PRIMARY KEY
);
```

Следующая ошибка возвращается при неудачном выполнении первой инструкции `CREATE TABLE`, прежде чем будет выполнена вторая инструкция.

```
ERROR 1824 (HY000): Failed to open the referenced table 'Parent'.
```

(Не удалось открыть таблицу ‘Parent’, на которую указывает ссылка.)

Порядок, в котором создаются таблицы, важен. Таблица `Parent` должна быть создана до определения внешнего ключа, ссылающегося на нее.

Foreign-Key-Checklist/table-order-correct.sql

```
CREATE TABLE Parent (
  parent_id INT PRIMARY KEY
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id INT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);
```

Если обе таблицы имеют внешний ключ, ссылающийся на другую таблицу (например, таблица `Child` ссылается на `Parent`, а `Parent` ссылается на одну избранную строку `Child`), придется использовать три инструкции определения данных вместо двух. Создайте первую таблицу без ограничения внешнего ключа, затем создайте вторую таблицу, после чего добавьте ограничение внешнего ключа в первую таблицу инструкцией `ALTER TABLE`.

Foreign-Key-Checklist/table-order-mutual.sql

```
CREATE TABLE Parent (
  parent_id INT PRIMARY KEY,
  favorite_child_id INT
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id INT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);

ALTER TABLE Parent
  ADD FOREIGN KEY (favorite_child_id) REFERENCES Child(child_id);
```

Если таблица содержит автоссылочный внешний ключ, можно определить внешний ключ в инструкции CREATE TABLE.

С ростом количества таблиц становится труднее создавать таблицы в правильном порядке. Можно начать с таблиц, которые сами по себе не имеют ограничений внешнего ключа. Эти таблицы можно считать корневыми на диаграмме «объект — отношение». Затем добавляются таблицы, ссылающиеся на корневые таблицы, и т. д. Это невозможно при наличии в наборе таблиц циклических ссылок внешних ключей, поэтому в качестве альтернативы можно сначала создать все таблицы без ограничений внешнего ключа, а затем добавить все ограничения в уже существующих таблицах.

Отсутствие ссылок на ключ родительской таблицы

Если столбцы таблицы Parent не имеют ограничения PRIMARY KEY или UNIQUE KEY, выдается сообщение об ошибке.

Foreign-Key-Checklist/no-key-error.sql

```
CREATE TABLE Parent (
  parent_id INT NOT NULL – не является PRIMARY KEY или UNIQUE KEY
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id INT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);


ERROR 1822 (HY000): Failed to add the foreign key constraint. Missing index for
constraint 'child_ibfk_1' in the referenced table 'Parent'.

(Не удалось добавить ограничение внешнего ключа. Отсутствует индекс для
ограничения 'child_ibfk_1' в таблице 'Parent', на которую указывает ссылка.)
```

Обратите внимание на упоминание имени ограничения child_ibfk_1 в сообщении об ошибке. У ограничений SQL есть имена — как у таблиц, индексов и столбцов. В примерах кода этой главы имя ограничения не указано, поэтому MySQL генерирует уникальное имя автоматически.

Столбцы, на которые указывает ссылка внешнего ключа, должны иметь ограничение PRIMARY KEY или UNIQUE KEY в таблице Parent.

Foreign-Key-Checklist/no-key-correct.sql

```
CREATE TABLE Parent (
  parent_id INT PRIMARY KEY
);
```

```
CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id INT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);
```

Создание отдельных ограничений для всех столбцов составного ключа

Если первичный ключ таблицы `Parent` состоит из нескольких столбцов, но внешний ключ разбивается на отдельные ограничения для всех столбцов, выдается сообщение об ошибке.

Foreign-Key-Checklist/multi-column-error.sql

```
CREATE TABLE Parent (
  parent_id1 INT,
  parent_id2 INT,
  PRIMARY KEY (parent_id1, parent_id2)
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id1 INT NOT NULL,
  parent_id2 INT NOT NULL,
  FOREIGN KEY (parent_id1) REFERENCES Parent(parent_id1),
  FOREIGN KEY (parent_id2) REFERENCES Parent(parent_id2)
);

ERROR 1822 (HY000): Failed to add the foreign key constraint. Missing index for
constraint 'child_ibfk_2' in the referenced table 'Parent'.

(Не удалось добавить ограничение внешнего ключа. Отсутствует индекс для
ограничения 'child_ibfk_2' в таблице 'Parent', на которую указывает ссылка.)
```

Если первичный ключ в таблице `Parent` состоит из нескольких столбцов, необходимо создать один внешний ключ, ссылающийся на оба столбца.

Foreign-Key-Checklist/multi-column-correct.sql

```
CREATE TABLE Parent (
  parent_id1 INT,
  parent_id2 INT,
  PRIMARY KEY (parent_id1, parent_id2)
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id1 INT NOT NULL,
  parent_id2 INT NOT NULL,
  FOREIGN KEY (parent_id1, parent_id2)
    REFERENCES Parent(parent_id1, parent_id2)
);
```

Неверный порядок столбцов

Если первичный ключ таблицы `Parent` состоит из нескольких столбцов, но во внешнем ключе они перечислены в неправильном порядке, сообщение об ошибке не выдается (при условии, что типы данных столбцов совместимы), но добавить строки данных будет невозможно, поскольку столбцы не ссылаются на нужные столбцы таблицы `Parent`.

Foreign-Key-Checklist/multi-column-order-error.sql

```
CREATE TABLE Parent (
  parent_id1 INT,
  parent_id2 INT,
  PRIMARY KEY (parent_id1, parent_id2)
);

INSERT INTO Parent (parent_id1, parent_id2) VALUES (1234, 5678);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id1 INT NOT NULL,
  parent_id2 INT NOT NULL,
  FOREIGN KEY (parent_id2, parent_id1)
    REFERENCES Parent(parent_id1, parent_id2)
);

INSERT INTO Child (child_id, parent_id1, parent_id2) VALUES (1, 1234, 5678);

ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint
fails (`test`.`child`,CONSTRAINT `child_ibfk_1` FOREIGN KEY (`parent_
id2`,`parent_id1`) REFERENCES `parent` (`parent_id1`, `parent_id2`))

(Не удается добавить или обновить дочернюю строку: нарушается ограничение
внешнего ключа (…))
```

Столбцы в ограничении внешнего ключа должны следовать в том же порядке, в каком они определяются в ограничении `PRIMARY KEY` или `UNIQUE KEY` в таблице `Parent`.

Foreign-Key-Checklist/multi-column-order-correct.sql

```
CREATE TABLE Parent (
  parent_id1 INT,
  parent_id2 INT,
  PRIMARY KEY (parent_id1, parent_id2)
);

INSERT INTO Parent (parent_id1, parent_id2) VALUES (1234, 5678);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id1 INT NOT NULL,
```

```
  parent_id2 INT NOT NULL,
  FOREIGN KEY (parent_id1, parent_id2)
  REFERENCES Parent(parent_id1, parent_id2)
);

INSERT INTO Child (child_id, parent_id1, parent_id2) VALUES (1, 1234, 5678);
```

Несоответствие типов данных

Столбцы внешнего ключа должны определяться в таблице `Child` с такими же типами данных, как у соответствующих столбцов таблицы `Parent`, на которые они ссылаются. Если типы данных не совпадают, выдается сообщение об ошибке.

Foreign-Key-Checklist/data-type-error.sql

```
CREATE TABLE Parent (
  parent_id INT PRIMARY KEY
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id VARCHAR(10) NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);

ERROR 3780 (HY000): Referencing column 'parent_id' and referenced column
'parent_id' in foreign key constraint 'child_ibfk_1' are incompatible.

(Ссылающийся столбец 'parent_id' и столбец 'parent_id' в ограничении внешнего
ключа 'child_ibfk_1', на который указывает ссылка, несовместимы.)
```

Чтобы столбцы оказались несовместимыми, достаточно различий в знаке целого числа.

Foreign-Key-Checklist/data-type-int-error.sql

```
CREATE TABLE Parent (
  parent_id INT PRIMARY KEY
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id INT UNSIGNED NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);

ERROR 3780 (HY000): Referencing column 'parent_id' and referenced column
'parent_id' in foreign key constraint 'child_ibfk_1' are incompatible
```

Лучшее решение — следить, чтобы типы данных были одинаковыми.

Foreign-Key-Checklist/data-type-correct.sql

```
CREATE TABLE Parent (
  parent_id VARCHAR(10) PRIMARY KEY
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id VARCHAR(10) NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);
```

Но как и у большинства правил, у этого правила существует исключение. Строковые столбцы с переменной длиной могут иметь разную максимальную длину, но тем не менее останутся совместимыми в отношении ссылок внешнего ключа.

Foreign-Key-Checklist/data-type-length-correct.sql

```
CREATE TABLE Parent (
  parent_id VARCHAR(10) PRIMARY KEY
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id VARCHAR(20) NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);
```

Если столбец внешнего ключа в таблице `Child` имеет меньшую максимальную длину, чем столбец таблицы `Parent`, на который он ссылается, это не ошибка, но в таком случае строки `Child` могут ссылаться только на строки `Parent` со строковыми значениями, не превышающими по длине строку `Child`. Таким образом, в таблице `Parent` можно сохранить более длинную строку, которая не может соответствовать ни одной строке таблицы `Child`. Присутствие в `Parent` строки, на которую не ссылается ни одна строка таблицы `Child`, ошибкой не является.

Точно так же столбец таблицы `Parent` может иметь меньшую максимальную длину, чем ссылающийся столбец из таблицы `Child`. Это тоже не ошибка. Строки таблицы `Child` должны ссылаться на строки таблицы `Parent`, так что в таблицу `Child` можно вставлять только короткие строки.

Несоответствие порядка сопоставления символов

Этот случай связан с предыдущим правилом о типах данных. В таблицах могут присутствовать строковые столбцы, которые предположительно имеют одинаковые типы, кроме порядка сопоставления. Если столбец, на который указывает ссылка, использует другой порядок сопоставления, выдается сообщение об ошибке.

Foreign-Key-Checklist/collation-error.sql

```
CREATE TABLE Parent (
  parent_id VARCHAR(10) PRIMARY KEY
) CHARSET utf8mb4 COLLATE utf8mb4_unicode_ci;

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id VARCHAR(10) NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
) CHARSET utf8mb4 COLLATE utf8mb4_general_ci;

ERROR 3780 (HY000): Referencing column 'parent_id' and referenced column
'parent_id' in foreign key constraint 'child_ibfk_1' are incompatible
```

Чтобы понять суть проблемы, необходимо знать, что собой представляют символьные кодировки и порядки сопоставления. Символьной кодировкой называется способ кодирования символов в байтах. Порядок сопоставления (collation) определяет, как символы некоторой кодировки должны сравниваться друг с другом (то есть для каждой пары символов генерировать результат «равно», «меньше» или «больше»). Правила сопоставления символов в столбце внешнего ключа и ключевом столбце, на который он ссылается, должны быть одинаковыми.

Следите, чтобы строковые столбцы имели совместимые кодировки символов и порядки сопоставления (на практике это означает, что порядки сопоставления должны быть одинаковыми).

Foreign-Key-Checklist/collation-correct.sql

```
CREATE TABLE Parent (
  parent_id VARCHAR(10) PRIMARY KEY
) CHARSET utf8mb4 COLLATE utf8mb4_unicode_ci;

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id VARCHAR(10) NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
) CHARSET utf8mb4 COLLATE utf8mb4_unicode_ci;
```

Создание осиротевших строк

Добавляя внешний ключ в таблицу `Child`, которая уже содержит данные, необходимо убедиться, что каждой строке таблицы `Child` соответствует строка в таблице `Parent`.

Foreign-Key-Checklist/orphan-error.sql

```
CREATE TABLE Parent (
  parent_id INT PRIMARY KEY
);
```

```
INSERT INTO Parent (parent_id)
VALUES (1234);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id INT NOT NULL
);

INSERT INTO Child (child_id, parent_id)
VALUES (1, 1234), (2, 5678);
```

В предыдущем примере таблица `Child` содержит вторую строку, у которой нет соответствующей строки в таблице `Parent`. Если в таблице `Child` присутствуют такие осиротевшие строки, то попытка добавления внешнего ключа завершится неудачей и будет выдано сообщение об ошибке.

Foreign-Key-Checklist/orphan-error.sql

```
ALTER TABLE Child
  ADD FOREIGN KEY (parent_id) REFERENCES Parent(parent_id);

ERROR 1452 (23000): Cannot add or update a child row: a foreign key constraint
fails (`test`.`child`, CONSTRAINT `child_ibfk_1` FOREIGN KEY (`parent_id`)
REFERENCES `parent` (`parent_id`))
```

(Не удается добавить или обновить дочернюю строку: нарушается ограничение внешнего ключа (...))

Каждое значение в столбце (столбцах) внешнего ключа должно соответствовать значению в столбцах, на которые указывает ссылка. Для проверки наличия осиротевших строк можно воспользоваться запросом следующего вида:

Foreign-Key-Checklist/orphan-check.sql

```
SELECT CASE COUNT(*)
  WHEN 0 THEN 'Ready to add foreign key'
  ELSE 'Do not add foreign key, because orphan rows exist'
  END AS `check`
FROM Child
LEFT OUTER JOIN Parent ON Child.parent_id = Parent.parent_id
WHERE Parent.parent_id IS NULL;
```

Это обобщенный пример запроса; подставьте в него нужные имена таблиц и столбцов.

Если в таблице `Child` присутствуют осиротевшие строки, добавить в таблицу внешний ключ не получится. Сначала необходимо исправить данные, используя один или несколько способов из следующего списка:

- Вставка (`INSERT`) новых строк в таблицу `Parent`, пока в таблице `Child` не останется ни одной строки с осиротевшими значениями.

- Обновление (UPDATE) строк таблицы Child и замена осиротевших значений на NULL или значение, соответствующее существующему значению в столбце (столбцах) таблицы Parent, на который указывает ссылка.
- Удаление (DELETE) строк из таблицы Child, пока не останется ни одного осиротевшего значения.

Применение SET NULL к столбцам, не допускающим NULL

К ограничению внешнего ключа можно добавить действия, которые будут выполняться при изменении столбца(-ов) таблицы Parent, на который(-е) указывает ссылка, или при удалении строки таблицы Parent, на которую указывает ссылка. Одно из таких дополнительных действий — SET NULL — заменяет значения таблицы Child на NULL, чтобы они не стали осиротевшими.

Если столбцы внешнего ключа определяются с NOT NULL и вы пытаетесь определить ограничение внешнего ключа с ON UPDATE SET NULL или ON DELETE SET NULL, выдается сообщение об ошибке.

Foreign-Key-Checklist/set-null-error.sql

```
CREATE TABLE Parent (
  parent_id INT PRIMARY KEY
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id INT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
    ON DELETE SET NULL
);
ERROR 1830 (HY000): Column 'parent_id' cannot be NOT NULL: needed in a foreign
key constraint 'child_ibfk_1' SET NULL

(Столбец 'parent_id' не может определяться с NOT NULL: условие ограничения
внешнего ключа 'child_ibfk_1' SET NULL)
```

Столбцы внешнего ключа должны допускать NULL, если требуется возможность присваивать им NULL по ссылочным действиям.

Foreign-Key-Checklist/set-null-correct.sql

```
CREATE TABLE Parent (
  parent_id INT PRIMARY KEY
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id INT NULL,
```

```
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
    ON DELETE SET NULL
);
```

Создание повторяющихся идентификаторов ограничений

Ограничения внешнего ключа могут иметь необязательные идентификаторы, чтобы их можно было использовать позже, при необходимости удалить ограничение. Идентификаторы ограничений должны быть уникальными в границах схемы. Иначе говоря, если двум и более ограничениям в одной схеме назначен один идентификатор, будет выдано сообщение об ошибке.

Foreign-Key-Checklist/identifier-error.sql

```
CREATE TABLE Parent (
  parent_id INT PRIMARY KEY
);

CREATE TABLE Child1 (
  child_id INT PRIMARY KEY,
  parent_id INT NOT NULL,
  CONSTRAINT c1 FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);

CREATE TABLE Child2 (
  child_id INT PRIMARY KEY,
  parent_id INT NOT NULL,
  CONSTRAINT c1 FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);
ERROR 1826 (HY000): Duplicate foreign key constraint name 'c1'

(Повторяющееся имя ограничения внешнего ключа 'c1')
```

Назначая идентификаторы ограничений, следите за тем, чтобы они были уникальными.

Foreign-Key-Checklist/identifier-correct.sql

```
CREATE TABLE Parent (
  parent_id INT PRIMARY KEY
);

CREATE TABLE Child1 (
  child_id INT PRIMARY KEY,
  parent_id INT NOT NULL,
  CONSTRAINT c1 FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);

CREATE TABLE Child2 (
  child_id INT PRIMARY KEY,
  parent_id INT NOT NULL,
  CONSTRAINT c2 FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);
```

Если вы решите присвоить имена ограничениям, выберите схему, позволяющую формировать уникальные имена. Если не задавать имена ограничений, то уникальные имена обычно генерируются автоматически.

Несовместимые типы таблиц

В стандартном SQL таблица `Child` и таблица `Parent` должны относиться к одному типу таблиц. Иначе говоря, обе должны быть либо таблицами долгосрочных баз, либо глобальными временными таблицами, либо локальными временными таблицами. См. также «Несовместимые типы таблиц в MySQL» в главе 27, с. 349.

Используйте эту главу как руководство по устранению возможных ошибок при создании внешних ключей в любой РСУБД на основе SQL.

Быть успешным не означает никогда не ошибаться —
это означает никогда не повторять своих ошибок.

➤ *Джош Биллингс (Josh Billings)*

ГЛАВА 27

Ошибки внешних ключей в MySQL

У всех реализаций SQL есть свои достоинства и недостатки, и существует ряд ошибок внешних ключей, свойственных конкретно MySQL. В этой главе представлена подборка мини-антипаттернов, относящихся к таким типам ограничений внешнего ключа, составленная на основе вопросов, часто задаваемых разработчиками на открытых интернет-форумах.

За каждым примером следует описание ошибки, возвращаемое MySQL 8.0, и приводится правильный способ реализации внешнего ключа.

Несовместимые ядра хранения данных

MySQL поддерживает разные механизмы хранения данных . Если две таблицы связаны ограничением внешнего ключа, они должны использовать один и тот же механизм и он должен поддерживать внешние ключи.

Используемый по умолчанию в MySQL механизм хранения данных InnoDB поддерживает внешние ключи. Многие другие механизмы хранения данных не поддерживают внешние ключи. Если вы попытаетесь определить внешний ключ в таблице InnoDB, но указанная таблица не является таблицей InnoDB, вы получите сообщение об ошибке.

Foreign-Key-Checklist/storage-engine-error.sql

```
CREATE TABLE Parent (
  parent_id INT PRIMARY KEY
) ENGINE=MyISAM;

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id INT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
) ENGINE=InnoDB;
ERROR 1824 (HY000): Failed to open the referenced table 'Parent'
```

У этой ошибки есть дополнительная разновидность: если механизм хранения данных таблицы `Child` не поддерживает ограничения внешнего ключа, то ограничение внешнего ключа просто игнорируется. Ошибки или предупреждения в этом случае не выдаются, но в полученной таблице отсутствует ограничение внешнего ключа.

Foreign-Key-Checklist/storage-engine-myisam.sql

```
CREATE TABLE Parent (
  parent_id INT PRIMARY KEY
) ENGINE=InnoDB;

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id INT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
) ENGINE=MyISAM;
```

У такого поведения в MySQL имеется объяснение: на первых порах разработчики MySQL хотели разрешить импортирование файлов определений SQL из других РСУБД, даже при том, что в MySQL еще не были реализованы все возможности SQL.

Убедитесь, что обе таблицы — `Parent` и `Child` — используют механизм хранения InnoDB. Если вы создаете таблицу без указания механизма, по умолчанию будет использоваться InnoDB.

Механизм хранения данных таблицы можно узнать при помощи команды `SHOW CREATE TABLE` *ИмяТаблицы* или выполнив запрос к таблицам метаданных следующего вида:

Foreign-Key-Checklist/storage-engine-check.sql

```
SELECT ENGINE FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_SCHEMA = ? AND TABLE_NAME = ?;
```

В приведенном примере запроса подставьте нужное имя схемы и имя таблицы вместо заполнителей `?`.

Механизм хранения данных NDB в MySQL Cluster также поддерживает внешние ключи. При этом действует аналогичное ограничение: если одна таблица в отношении использует механизм хранения данных NDB, то другая таблица также должна использовать NDB.

Использование больших типов данных

В стандартном SQL невозможно определить `PRIMARY KEY`, `UNIQUE KEY` или ограничение внешнего ключа для столбца `BLOB`, `CLOB`, `TEXT`, `JSON` или `ARRAY`.

В MySQL определение ключей или индексов для таких больших столбцов переменного размера также не поддерживается, потому что индексированный тип данных не должен превышать 3072 байта (или 767 байт в старых версиях). Вы можете создать *префиксный индекс*, чтобы сгенерировать ключ или индекс по начальным байтам индексируемого столбца. И хотя это позволяет создать ограничение `PRIMARY KEY` или `UNIQUE KEY` по части длинного столбца, MySQL не может создать внешний ключ, ссылающийся на префиксный индекс, и вы получите сообщение об ошибке.

Foreign-Key-Checklist/text-error.sql

```
CREATE TABLE Parent (
  parent_id TEXT NOT NULL,
  UNIQUE KEY (parent_id(40))
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id TEXT NOT NULL,
  KEY (parent_id(40)),
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);
ERROR 1170 (42000): BLOB/TEXT column 'parent_id' used in key specification
without a key length.

(Столбец BLOB/TEXT 'parent_id' используется в спецификации ключа без длины
ключа)
```

Такой длинный столбец не может быть внешним ключом сам по себе. Обходное решение в MySQL 5.7 и более поздних версиях основано на создании *генерируемого столбца*, содержащего хеш-код длинного столбца, и определения внешнего ключа для этого столбца. Генерируемый столбец должен использовать режим `STORED`, чтобы на него можно было ссылаться из внешнего ключа. Следующий пример показывает, как определяется генерируемый столбец со `STORED` для реализации этого обходного решения.

Foreign-Key-Checklist/text-workaround.sql

```
CREATE TABLE Parent (
  parent_id TEXT NOT NULL,
  parent_id_crc INT UNSIGNED AS (CRC32(parent_id)) STORED,
  UNIQUE KEY (parent_id_crc)
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id_crc INT UNSIGNED,
  FOREIGN KEY (parent_id_crc) REFERENCES Parent(parent_id_crc)
);
```

Существует небольшой риск того, что для двух разных текстов будут сгенерированы одинаковые хеш-коды, и это приведет к ошибке дублирования в уни-

кальном индексе. Можно использовать хеш-функцию с большим пространством значений (например, `MD5()` или `SHA1()`), но риск коллизий никогда не будет исключен. В качестве уникального ключа таблицы лучше использовать псевдоключ вместо столбца с таким длинным содержимым.

Внешние ключи MySQL с неуникальными индексами

В стандартном SQL столбцы, на которые ссылается внешний ключ, должны быть объявлены с `PRIMARY KEY` или `UNIQUE KEY` в таблице `Parent`. InnoDB поддерживает нестандартную возможность: внешний ключ не обязан включать весь набор столбцов ключа, на который указывает ссылка. Внешний ключ также может ссылаться на неуникальный индекс родительской таблицы (вместо `PRIMARY KEY` или `UNIQUE KEY`). Действует только одно правило: столбцы, на которые ссылается внешний ключ, должны быть крайними левыми столбцами ключа или индекса. В противном случае вы получите сообщение об ошибке, в котором говорится, что в таблице `Parent` не найден индекс с этими столбцами в крайних левых позициях.

Foreign-Key-Checklist/non-unique-error.sql

```
CREATE TABLE Parent (
  parent_id1 INT,
  parent_id2 INT,
  parent_id3 INT,
  PRIMARY KEY (parent_id1, parent_id2, parent_id3)
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id2 INT NOT NULL,
  parent_id3 INT NOT NULL,
  FOREIGN KEY (parent_id2, parent_id3)
    REFERENCES Parent(parent_id2, parent_id3)
);

ERROR 1822 (HY000): Failed to add the foreign key constraint. Missing index for
constraint 'child_ibfk_1' in the referenced table 'Parent'
```

Столбцы внешнего ключа должны ссылаться на крайнее левое подмножество столбцов ключа или индекса.

Foreign-Key-Checklist/non-unique-left-subset.sql

```
CREATE TABLE Parent (
  parent_id1 INT,
  parent_id2 INT,
  parent_id3 INT,
  PRIMARY KEY (parent_id1, parent_id2, parent_id3)
);
```

```
CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id1 INT NOT NULL,
  parent_id2 INT NOT NULL,
  FOREIGN KEY (parent_id1, parent_id2)
    REFERENCES Parent(parent_id1, parent_id2)
);
```

И хотя ссылки на неуникальный индекс или подмножество ключевых столбцов разрешены, так поступать не рекомендуется.

Внешний ключ, ссылающийся на неуникальный индекс или подмножество ключевых столбцов, позволяет заданной строке таблицы `Child` ссылаться на значения, которые могут встречаться в нескольких строках таблицы `Parent`, на которую указывает ссылка. Это приводит к возникновению нетипичных и неоднозначных логических отношений наподобие следующих:

- Считать ли строку `Child` осиротевшей, если удаляется одна, но не все соответствующие строки из `Parent`?
- Если внешний ключ определяется в режиме `ON DELETE RESTRICT`, что будет ошибкой — удаление любой строки из `Parent` или только последней?
- Если внешний ключ определяется в режиме `ON UPDATE CASCADE` и значение, на которое указывает ссылка, обновляется в одной строке таблицы `Parent`, произойдет ли каскадное распространение изменения в строку `Child`?

Ответы на эти вопросы зависят от ситуации. Они могут быть утвердительными в одном случае и отрицательными в другом. Лучше избегать таких ситуаций, определяя внешние ключи только стандартным образом, а именно: столбцы внешнего ключа должны ссылаться на полный набор столбцов `PRIMARY KEY` или `UNIQUE KEY`, так что строка `Child` должна ссылаться только на одну строку таблицы `Parent`.

Foreign-Key-Checklist/non-unique-correct.sql

```
CREATE TABLE Parent (
  parent_id1 INT,
  parent_id2 INT,
  parent_id3 INT,
  PRIMARY KEY (parent_id1, parent_id2, parent_id3)
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id1 INT NOT NULL,
  parent_id2 INT NOT NULL,
  parent_id3 INT NOT NULL,
  FOREIGN KEY (parent_id1, parent_id2, parent_id3)
    REFERENCES Parent(parent_id1, parent_id2, parent_id3)
);
```

Синтаксис встроенных ссылок

Стандартный SQL и большинство реализаций поддерживают синтаксис определения внешнего ключа для одного столбца в той же строке, в которой определяется столбец. Однако MySQL не поддерживает синтаксис встроенных внешних ключей. Если вы попытаетесь определить внешний ключ подобным способом, ошибки не будет, но внешний ключ не добавится в таблицу[1].

Foreign-Key-Checklist/inline-ignored.sql

```
CREATE TABLE Parent (
  parent_id VARCHAR(10) PRIMARY KEY
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id VARCHAR(10) NOT NULL REFERENCES Parent(parent_id)
);
```

Если вы позже просмотрите определение таблицы инструкцией `SHOW CREATE TABLE Child`, то увидите, что ограничение внешнего ключа отсутствует, как если бы он вообще не определялся.

Foreign-Key-Checklist/inline-ignored.sql

```
CREATE TABLE `Child` (
  `child_id` int NOT NULL,
  `parent_id` varchar(10) NOT NULL,
  PRIMARY KEY (`child_id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci
```

Внешний ключ должен определяться только с синтаксисом ограничений уровня таблицы.

Foreign-Key-Checklist/inline-correct.sql

```
CREATE TABLE Parent (
  parent_id VARCHAR(10) PRIMARY KEY
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id VARCHAR(10) NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);
```

[1] https://bugs.mysql.com/bug.php?id=4919

Синтаксис ссылок по умолчанию

Стандарный SQL и многие реализации позволяют в секции `REFERENCES` во внешних ключах опускать имена столбцов, на которые указывает ссылка; в этом случае по умолчанию используется столбцы первичного ключа таблицы. Однако в MySQL синтаксис неявных ссылок на столбцы не поддерживается[1].

Foreign-Key-Checklist/implicit-columns-error.sql

```
CREATE TABLE Parent (
  parent_id VARCHAR(10) PRIMARY KEY
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id VARCHAR(10) NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent
);
ERROR 1239 (42000): Incorrect foreign key definition for 'foreign key without
name': Key reference and table reference don't match.

(Неверное определение внешнего ключа для 'foreign key without name': несоответ-
ствие между ссылкой на ключ и ссылкой на таблицу)
```

Внешний ключ может определяться только с явным указанием имен в ссылках.

Foreign-Key-Checklist/implicit-columns-correct.sql

```
CREATE TABLE Parent (
  parent_id VARCHAR(10) PRIMARY KEY
);

CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id VARCHAR(10) NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);
```

Несовместимые типы таблиц в MySQL

В MySQL ни таблица `Parent`, ни таблица `Child` не может быть таблицей `TEMPORARY` или таблицей `PARTITIONED`.

Foreign-Key-Checklist/partition-error.sql

```
CREATE TABLE Parent (
  parent_id INT PRIMARY KEY
);
```

[1] https://bugs.mysql.com/bug.php?id=35522

```
CREATE TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id INT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
) PARTITION BY HASH(child_id) PARTITIONS 11;
ERROR 1506 (HY000): Foreign keys are not yet supported in conjunction with
partitioning.

(Внешние ключи еще не поддерживаются в сочетании с партицированием)
```

Foreign-Key-Checklist/temp-error.sql

```
CREATE TABLE Parent (
  parent_id INT PRIMARY KEY
);

CREATE TEMPORARY TABLE Child (
  child_id INT PRIMARY KEY,
  parent_id INT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Parent(parent_id)
);
ERROR 1215 (HY000): Cannot add foreign key constraint.

(Не удается добавить ограничение внешнего ключа)
```

В обоих случаях должны использоваться не временные и не партицированные таблицы.

Используйте эту главу как руководство по устранению возможных ошибок при создании внешних ключей в MySQL.

Юноша, в математике не надо ничего понимать.
Нужно просто привыкнуть.

➤ *Джон фон Нейман (John von Neumann)*

ПРИЛОЖЕНИЕ

Правила нормализации

В архитектуре реляционных баз данных нет ничего загадочного или непредсказуемого. Используя набор четко определенных правил, можно спроектировать стратегию хранения данных, которая предотвращает избыточность и помогает повысить устойчивость приложений к ошибкам, как концепция *poka-yoke*, о которой речь шла выше в книге. Вероятно, вы слышали и другие метафоры для выражения той же идеи, например *защитное проектирование* (defensive design) или *быстрый отказ* (fail early).

Правила нормализации не очень сложны, но в них много нюансов. Разработчики часто не понимают, как они работают. Возможно, они подсознательно ожидают, что эти правила сложнее, чем это есть на самом деле.

Также, возможно, их обескураживает необходимость следовать правилам. Все разработчики, которые ценят новизну, творчество и инновации, терпеть не могут правила. В каком-то смысле правила являются противоположностью свободы.

Разработчикам постоянно приходится искать компромисс между простотой и гибкостью. Можно выполнять большую часть работы самостоятельно, заново изобретая велосипед и разрабатывая специализированные схемы управления данными для каждого приложения. А можно воспользоваться существующими знаниями и технологиями, если придерживаться реляционной структуры при использовании реляционной базы данных.

Антипаттерны в этой книге описываются с точки зрения их собственных достоинств (или недостатков), чтобы не делать изложение слишком отвлеченным или теоретическим. В этом приложении вы увидите, что теория также может принести практическую пользу.

Что значит «реляционный»?

Термин «*реляционный*» не связан с отношениями между таблицами. Он относится к самой таблице, а вернее, к отношениям между столбцами в таблице.

Математики определяют *отношение* как комбинацию двух множеств значений из разных областей с применением условия, которое определяет подмножество всех возможных комбинаций.

Например, одно множество составляют названия бейсбольных команд, а другое — названия городов. Комбинации каждой команды с каждым городом образуют длинный список пар. Отношение выделяет особое подмножество этого списка: команды, входящие в пару со своим домашним городом. Допустимые пары: Чикаго/White Sox, Чикаго/Cubs или Бостон/Red Sox, но не Майами/Red Sox.

Слово «*отношение*» используется двумя способами: как правило («этот город является домашним для этой команды») и как подмножество пар, которые соответствуют этому правилу. В SQL этот результат может храниться в таблице с двумя столбцами и с одной строкой данных на пару.

Конечно, отношения не ограничиваются двумя столбцами. В отношение можно объединить любое количество областей, по одной на столбец. Кроме того, можно использовать такие области значений, как множество 32-разрядных целых чисел или множество текстовых строк конкретной длины.

Прежде чем переходить к нормализации таблиц, необходимо убедиться, что они находятся в правильных отношениях. Для этого они должны удовлетворять ряду критериев.

Для строк не определен порядок «сверху вниз»

В SQL запрос возвращает результаты в непредсказуемом порядке, если только вы не воспользуетесь секцией `ORDER BY` для определения порядка. Независимо от порядка множества строк считаются одинаковыми.

Для столбцов не определен порядок «слева направо»

Независимо от того, просим ли мы разработчика проверить продукт Open RoundFile на наличие ошибки 1234 или нам нужно знать, может ли этот разработчик проверить ошибку 1234 в продукте Open RoundFile, результаты должны быть одинаковыми.

Ситуация отчасти напоминает антипаттерн из главы 19 «Неявные столбцы», где для ссылок на столбцы используется позиция, а не имена.

Запрет дубликатов

Если вам известен какой-то факт, его повторение не сделает его более правдивым. Если название бейсбольной команды известно, то данные определяют город. В этом случае город *зависит* от названия команды.

Чтобы предотвратить появление дубликатов, необходимо иметь возможность отличить одну строку от другой и обращаться к отдельным строкам. Чтобы реализовать такую возможность в SQL, объявите ограничение первичного ключа для столбца или набора столбцов (в зависимости от того, как вы хотите однозначно идентифицировать строки). Также можно объявить ограничение уникального ключа, если столбец, указанный в ограничении, объявлен с `NOT NULL`.

Дубликаты в неключевых столбцах допустимы (например, могут быть две бейсбольные команды из Бостона), но строка в целом остается уникальной, потому что имена команд различаются.

Каждый столбец имеет только один тип и только одно значение на строку

Отношение имеет *заголовок*, определяющий имена и типы данных столбцов. Каждая строка должна содержать те же столбцы, что и заголовок, и смысл отдельного столбца должен оставаться одинаковым для всех строк.

Примеры нарушения этого правила антипаттерном были представлены в главе 6 «Сущность — атрибут — значение». Во-первых, таблица EAV моделирует сущность, которая может содержать специализированный набор атрибутов для каждого экземпляра, так что сущность не привязывается ни к какому заголовку, определяющему ее атрибуты.

Во-вторых, столбец EAV `attr_value` содержит все атрибуты сущности (например, дату регистрации ошибки, статус ошибки, учетную запись ответственного пользователя и т. д.). Конкретное значение в этом столбце (например, `1234`) может быть допустимым для двух разных атрибутов, но иметь совершенно разный смысл.

Антипаттерн в главе 7 «Полиморфная связь» также нарушает это правило, потому что заданное значение (например, `1234`) ссылается на первичный ключ нескольких родительских таблиц. Нельзя утверждать, что `1234` в одной строке означает то же, что и `1234` в другой строке.

Строки не имеют скрытых компонентов

Столбцы содержат значения данных, а не физические признаки хранения (скажем, идентификаторы строк или объектов). В главе 22 «Чистка псевдоключа» мы выяснили, что первичные ключи уникальны, но они не являются номерами строк.

Некоторые базы данных нарушают это правило, предоставляя доступ к внутренним подробностям хранения в расширениях SQL (например, псевдостолбец `ROWNUM` в Oracle или `OID` в PostgreSQL). Впрочем, эти значения не являются частью отношения.

Мифы о нормализации

Трудно найти другую тему, которая бы порождала столько недопонимания, несмотря на четкое определение. Вы наверняка встретите разработчиков, которые с полной уверенностью говорят:

- «Нормализация замедляет работу базы данных. Денормализация ускоряет работу базы данных».

  Неправда. Действительно, после применения нормализации может понадобиться применить соединение для получения атрибутов из отдельных таблиц. Денормализация данных позволит избежать некоторых соединений.

  Например, в списке, разделенном запятыми, из главы 2 «Кривая дорожка» перечисляются продукты для данной ошибки. А если нужно задать список ошибок для данного продукта? Денормализация обычно повышает удобство или производительность одного типа запросов, но увеличивает расходы на другие типы запросов.

  Хотя применение денормализации в некоторых случаях обоснованно, стоит начать с моделирования базы данных в нормальных формах, прежде чем принимать решение о денормализации. Рекомендации MENTOR по индексированию из главы 13 «Индексный дробовик» также применимы к нормализации: в целях эффективности обязательно оцените производительность до и после внесения изменений.

- «Нормализация требует вынести данные в дочерние таблицы и ссылаться на них через псевдоключ».

  Неправда. Псевдоключи можно использовать в целях повышения удобства, производительности или эффективности хранения, и эти причины вполне обоснованны. Тем не менее они не имеют никакого отношения к нормализации.

- «Нормализация требует, чтобы атрибуты были по возможности разделены, как в архитектуре «сущность — атрибут — значение».

  Неправда. Разработчики часто используют термин «*нормализация*» неточно, подразумевая, что нормализованные данные менее понятны человеку или менее удобны для запросов. На самом деле всё наоборот.

- «Нормализовать дальше третьей нормальной формы никому не потребуется. Остальные нормальные формы настолько редкие, что никогда не встретятся».

  Неправда. Одно исследование показало, что более 20 % баз данных бизнеса используют структуры, удовлетворяющие первым трем нормальным формам, но нарушают требования четвертой нормальной формы. Это меньшинство, но его нельзя назвать несущественным. Если вы узнаете об ошибке, которая теоретически может привести к потере данных и встречается в 20 % ваших приложений, разве вы не захотите ее исправить?»

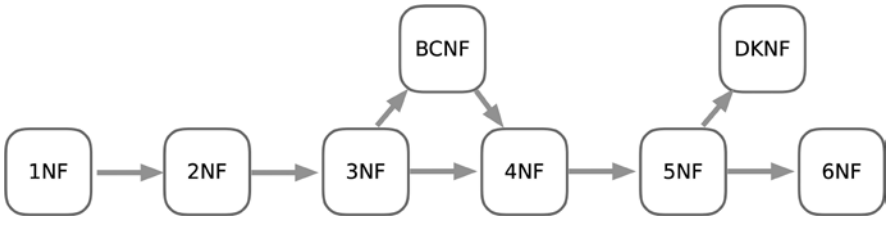
Что такое нормализация?

Ниже перечислены основные цели нормализации:

- Представление фактов о реальном мире в понятной форме.
- Сокращение избыточного хранения информации и предотвращение появления аномальных или противоречивых данных.
- Поддержка ограничений целостности.

Обратите внимание: повышения производительности базы данных в списке нет. Нормализация поможет *правильно* хранить данные и избежать проблем при использовании SQL для хранения этих данных. Ненормализованная база данных практически неизбежно превращается в хаотичное месиво. Скорее всего, вам придется написать намного больше кода для чистки противоречивых или дублирующихся данных. Дефектные данные приведут к задержкам и затратам для бизнеса. Если учесть такие сценарии, преимущества нормализации базы данных для производительности станут более очевидными.

Если таблица соответствует правилам нормализации, это значит, что она имеет *нормальную форму*. Существуют пять традиционных нормальных форм, описывающих последовательные уровни нормализации. Каждая нормальная форма исключает определенный тип избыточности или аномалий при проектировании отношений. В общем случае если таблица соответствует одной нормальной форме, то она также соответствует всем предшествующим нормальным формам. Существуют три дополнительные нормальные формы. Иерархия нормальных форм выглядит так:



Первая нормальная форма

Фундаментальное требование первой нормальной формы заключается в том, что таблица должна быть отношением. Если таблица не соответствует критериям отношения, описанным в первом разделе, то она не может иметь первую нормальную форму или любую из последующих нормальных форм.

Следующее требование — в таблице не должно быть *повторяющихся групп*. Вспомните, что каждая строка отношения является комбинацией нескольких множеств, с выбором одного значения из каждого множества. Повторяющаяся

группа означает, что одна строка может содержать несколько значений из заданного множества.

Проблемы также могут возникнуть при проектировании таблицы с набором похожих столбцов, когда эти столбцы должны быть одним столбцом с разными строками. Такую структуру иногда неформально относят к примерам повторяющихся групп. Строго говоря, она не эквивалентна повторяющимся группам, так как каждый столбец содержит одно значение, но проблемы с ней те же.

Два антипаттерна этой книги нарушают требования первой нормальной формы:

- множественные значения из одной области в нескольких столбцах — глава 8 «Многостолбцовые атрибуты»;
- множественные значения в одном столбце — глава 2 «Кривая дорожка».

Эти антипаттерны приводят к появлению разреженных столбцов или строк, разделенных запятыми, как показано на следующих схемах:

BugsTags

bug_id	tag_1	tag_2	tag_3
1234	crash		
3456	printing	crash	
5678	report	crash	data

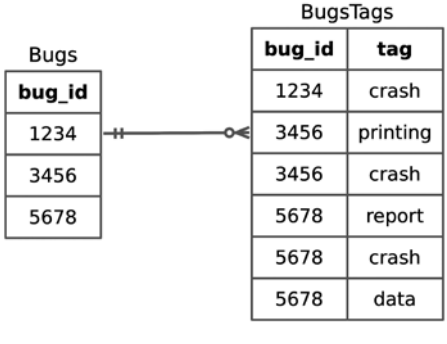
Многостолбцовые атрибуты

BugsTags

bug_id	tags
1234	crash
3456	printing,crash
5678	report,crash,data

Кривая дорожка

Правильная структура, удовлетворяющая требованиям первой нормальной формы, подразумевает создание отдельной таблицы, в которой каждое значение находится в отдельной строке и в одном столбце.



В предыдущем примере для поддержки множественных тегов можно вставить столько строк данных, сколько нужно.

Вторая нормальная форма

Вторая нормальная форма идентична первой, если таблица не содержит составного первичного ключа. В примере с тегами сохраняется информация о том, какой пользователь решил назначить каждый отдельный тег ошибке. В таблице также присутствует атрибут для идентификации пользователя, который первым определил заданный тег.

Normalization/2NF-anti.sql

```
CREATE TABLE BugsTags (
  bug_id BIGINT UNSIGNED NOT NULL,
  tag VARCHAR(20) NOT NULL,
  tagger BIGINT UNSIGNED NOT NULL,
  coiner BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (tagger) REFERENCES Accounts(account_id),
  FOREIGN KEY (coiner) REFERENCES Accounts(account_id)
);
```

Мы видим, что данные создателя тега (`coiner`) хранятся избыточно (на следующей диаграмме вместо идентификаторов используются имена):

BugsTags

bug_id	tag	tagger	coiner
1234	crash	Larry	Shemp
3456	printing	Larry	Shemp
3456	crash	Moe	Shemp
5678	report	Moe	Shemp
5678	crash	Larry	Shemp
5678	data	Moe	Shemp

Избыточность

BugsTags

bug_id	tag	tagger	coiner
1234	crash	Larry	Shemp
3456	printing	Larry	Shemp
3456	crash	Moe	Shemp
5678	report	Moe	Shemp
5678	crash	Larry	**Curly**
5678	data	Moe	Shemp

Аномалия

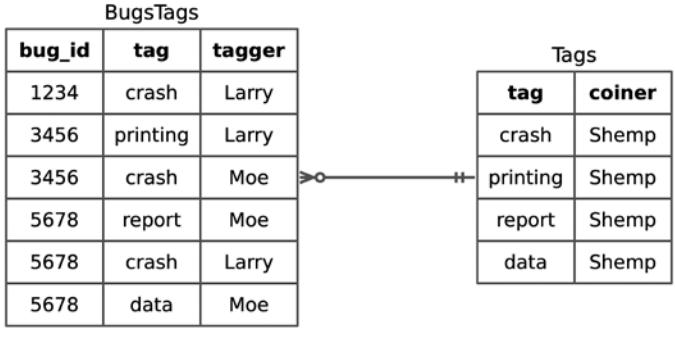
А это означает, что кто-то может создать *аномалию*, изменив личность создателя в одной строке для заданного тега (`crash`) без изменения всех строк для того же тега.

Чтобы соответствовать второй нормальной форме, следует хранить данные создателя для заданного тега в одном экземпляре. Это означает, что необходимо создать другую таблицу `Tags`, где тег является первичным ключом, чтобы для каждого отдельного тега существовала только одна запись. Затем можно сохранить создателя тега в новой таблице вместо `BugsTags` и тем самым предотвратить возможные аномалии.

Normalization/2NF-normal.sql

```
CREATE TABLE Tags (
  tag VARCHAR(20) PRIMARY KEY,
  coiner BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (coiner) REFERENCES Accounts(account_id)
);

CREATE TABLE BugsTags (
  bug_id BIGINT UNSIGNED NOT NULL,
  tag VARCHAR(20) NOT NULL,
  tagger BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (tag) REFERENCES Tags(tag),
  FOREIGN KEY (tagger) REFERENCES Accounts(account_id)
);
```

Следующая схема показывает, что у каждого тега есть только один пользователь, обозначенный как его создатель, что исключает появление аномалий.



Третья нормальная форма

Предположим, вам потребовалось сохранить в таблице `Bugs` адрес электронной почты специалиста, которому поручено работать над ошибкой.

Normalization/3NF-anti.sql

```
CREATE TABLE Bugs (
  bug_id SERIAL PRIMARY KEY,
  -- . . .
  assigned_to BIGINT UNSIGNED,
  assigned_email VARCHAR(100),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id)
);
```

Однако адрес электронной почты является атрибутом учетной записи назначенного инженера; строго говоря, он не является атрибутом ошибки. Хранить адрес электронной почты подобным образом будет избыточно, и в таблице, нарушающей требования второй нормальной формы, возникает риск аномалий.

В примере второй нормальной формы столбец-нарушитель связан по крайней мере с *частью* составного первичного ключа. В следующем примере, нарушающем требования третьей нормальной формы, столбец-нарушитель вообще не связан с первичным ключом.

Bugs

bug_id	assigned_to	assigned_email
1234	Larry	larry@example.com
3456	Moe	moe@example.com
5678	Moe	moe@example.com

Избыточность

Bugs

bug_id	assigned_to	assigned_email
1234	Larry	larry@example.com
3456	Moe	moe@example.com
5678	Moe	**curly@example.com**

Аномалия

Чтобы исправить положение, поместите адрес электронной почты в таблицу `Accounts`. Столбец можно отделить от таблицы `Bugs`:

Bugs

bug_id	assigned_to
1234	Larry
3456	Moe
3456	Moe

Accounts

account_id	email
Larry	larry@example.com
Moe	moe@example.com

И это верное решение, поскольку теперь адрес электронной почты напрямую соответствует первичному ключу таблицы без избыточности.

Нормальная форма Бойса — Кодда

Несколько усиленная версия третьей нормальной формы называется *нормальной формой Бойса — Кодда*. Различия между этими двумя нормальными формами заключаются в том, что в третьей нормальной форме все неключевые атрибуты должны зависеть от ключа таблицы. В нормальной форме Бойса — Кодда это правило распространяется как на ключевые, так и на неключевые столбцы. Ситуация встречается только в том случае, если таблица содержит несколько наборов столбцов, которые *могут* служить ключом таблицы. Это так называемые *ключи-кандидаты* (или *потенциальные ключи*). Некоторые таблицы содержат несколько ключей-кандидатов, но только один из них становится первичным ключом этой таблицы.

Предположим, существуют три разновидности тегов: теги, описывающие последствия ошибки; теги подсистемы, на которую влияет ошибка; и теги, описывающие исправление ошибки. Каждая ошибка должна иметь не более одного тега каждого типа. Ключом-кандидатом может быть пара `bug_id` и `tag`, а может быть и другая пара `bug_id` и `tag_type`. Любая из этих пар столбцов будет достаточно конкретной, чтобы адресовать каждую отдельную строку, так что оба варианта являются ключами-кандидатами для таблицы.

На следующей схеме приведен пример таблицы, которая соответствует требованиям третьей нормальной формы, но не нормальной формы Бойса — Кодда.

BugsTags

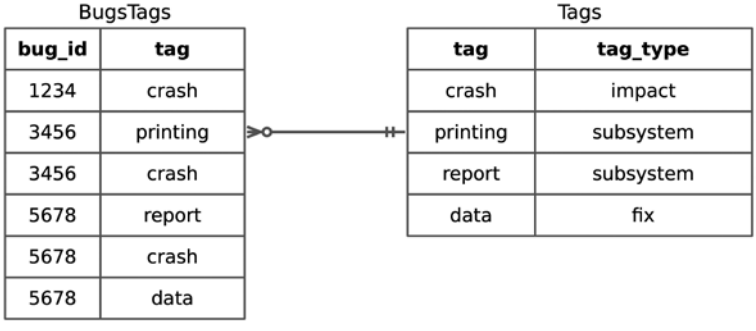
bug_id	tag	tag_type
1234	crash	impact
3456	printing	subsystem
5678	crash	impact
5678	report	subsystem
5678	crash	impact
5678	data	fix

Несколько ключей-кандидатов

BugsTags

bug_id	tag	tag_type
1234	crash	impact
3456	printing	subsystem
5678	crash	impact
5678	report	subsystem
5678	crash	subsystem
5678	data	fix

Чтобы исправить эту таблицу, проведите рефакторинг атрибута `tag_type` в таблице `Tags`, как на следующей схеме:



Если тип тега станет атрибутом таблицы `Tags`, это предотвратит возможные аномалии. Теперь таблица удовлетворяет требованиям нормальной формы Бойса — Кодда.

Четвертая нормальная форма

Теперь изменим базу данных, чтобы о каждой ошибке могли сообщать сразу несколько пользователей, ее можно было закреплять за несколькими разработчиками и ее могли проверять несколько QA-инженеров. Отношение «многие ко многим» требует отдельной таблицы:

Normalization/4NF-anti.sql

```
CREATE TABLE BugsAccounts (
  bug_id BIGINT UNSIGNED NOT NULL,
  reported_by BIGINT UNSIGNED,
  assigned_to BIGINT UNSIGNED,
  verified_by BIGINT UNSIGNED,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (verified_by) REFERENCES Accounts(account_id)
);
```

Столбец `bug_id` не может использоваться в одиночку в качестве первичного ключа. Чтобы иметь возможность хранить несколько учетных записей в каждом столбце, понадобится несколько строк на каждый баг. Также невозможно объявить первичный ключ по первым двум или трем столбцам, поскольку при этом все равно не будут поддерживаться множественные значения в последнем столбце. Таким образом, первичный ключ должен охватывать все четыре столбца. Однако столбцы `assigned_to` и `verified_by` должны допускать `NULL`, потому что сообщения об ошибках могут поступить до закрепления их за разработчи-

ками или проверки. Все столбцы первичных ключей стандартно содержат ограничение `NOT NULL`.

Другая проблема заключается в том, что таблица может содержать избыточные значения, если какой-то столбец содержит меньше учетных записей, чем другой. Избыточные значения показаны на следующей иллюстрации.

BugsAccounts

bug_id	reported_by	assigned_to	verified_by
1234	Zeppo	NULL	NULL
3456	Chico	Groucho	Harpo
3456	Chico	Spalding	Harpo
5678	Chico	Groucho	NULL
5678	Zeppo	Groucho	NULL
5678	Gummo	Groucho	NULL

Избыточность, NULL, нет первичного ключа

Все эти проблемы возникают из-за создания таблицы пересечений, которая пытается решать сразу две задачи — и даже три в данном случае. Использование одной таблицы пересечений для представления нескольких отношений «многие ко многим» нарушает требования четвертой нормальной формы.

Проблему можно решить разбиением таблицы и выделением одной таблицы пересечений для каждой разновидности отношений «многие ко многим». Тем самым устраняется избыточность и несовпадение числа значений в каждом столбце.

Normalization/4NF-normal.sql

```
CREATE TABLE BugsReported (
  bug_id BIGINT NOT NULL,
  reported_by BIGINT NOT NULL,
  PRIMARY KEY (bug_id, reported_by),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
);

CREATE TABLE BugsAssigned (
  bug_id BIGINT NOT NULL,
  assigned_to BIGINT NOT NULL,
  PRIMARY KEY (bug_id, assigned_to),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id)
);

CREATE TABLE BugsVerified (
  bug_id BIGINT NOT NULL,
  verified_by BIGINT NOT NULL,
  PRIMARY KEY (bug_id, verified_by),
```

```
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (verified_by) REFERENCES Accounts(account_id)
);
```

Выделение каждого отношения «многие ко многим» в отдельную таблицу будет выглядеть так:

BugsReported

bug_id	reported_by
1234	Zeppo
3456	Chico
5678	Zeppo
5678	Gummo

BugsAssigned

bug_id	assigned_to
3456	Groucho
3456	Spalding
5678	Groucho

BugsVerified

bug_id	verified_by
3456	Harpo

Четвертая нормальная форма уже не кажется такой редкостью, и понятно, почему она важна.

Пятая нормальная форма

Любая таблица, которая удовлетворяет критериям нормальной формы Бойса — Кодда и не имеет составного первичного ключа, уже находится в пятой нормальной форме. Чтобы понять смысл пятой нормальной формы, рассмотрим следующую ситуацию.

Некоторые разработчики работают только над определенными продуктами. Требуется спроектировать базу данных так, чтобы можно было узнать, кто над какими продуктами и какими ошибками работает, с минимумом избыточности. В первой версии можно добавить в таблицу `BugsAssigned` столбец, показывающий, что данный инженер работает над продуктом :

Normalization/5NF-anti.sql

```
CREATE TABLE BugsAssigned (
  bug_id BIGINT UNSIGNED NOT NULL,
  assigned_to BIGINT UNSIGNED NOT NULL,
  product_id BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (bug_id, assigned_to),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

Это изменение не сообщает, какие продукты могут быть закреплены за разработчиком для исправления ошибок; оно только перечисляет, какие продукты

в настоящее время закреплены за инженером. Тот факт, что инженер работает над заданным продуктом, сохраняется избыточно. Это происходит из-за хранения множественных фактов о независимых отношениях «многие ко многим» в одной таблице, по аналогии с проблемой четвертой нормальной формы. Избыточность показана на следующей иллюстрации (на ней вместо идентификаторов используются имена):

BugsAssigned

bug_id	assigned_to	product_id
3456	Groucho	Open Roundfile
3456	Spalding	Open Roundfile
5678	Groucho	Open Roundfile

Избыточность, множественные факты

Проблема решается изоляцией каждого отношения в отдельной таблице:

Normalization/5NF-normal.sql

```
CREATE TABLE BugsAssigned (
  bug_id BIGINT UNSIGNED NOT NULL,
  assigned_to BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (bug_id, assigned_to),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id)
);

CREATE TABLE EngineerProducts (
  account_id BIGINT UNSIGNED NOT NULL,
  product_id BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (account_id, product_id),
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

Изменения представлены на следующем рисунке:

BugsAssigned

bug_id	assigned_to
3456	Groucho
3456	Spalding
5678	Groucho

EngineerProducts

account_id	product_id
Groucho	Open Roundfile
Groucho	ReConsider
Spalding	Open Roundfile
Spalding	Visual Turbo Builder

Теперь в базе данных можно сохранить тот факт, что разработчик доступен для работы над данным продуктом, даже если сейчас он не занят исправлением ошибки этого продукта.

Другие нормальные формы

Доменно-ключевая нормальная форма (DKNF) требует, чтобы каждое ограничение таблицы являлось логическим следствием ограничений домена (области значений) и ограничений ключей таблицы. DKNF охватывает третью, четвертую и пятую нормальную форму, а также нормальную форму Бойса — Кодда.

Например, можно решить, что ошибка со статусом `NEW` или `DUPLICATE` не требует обработки, так что для нее не нужно сохранять затраты рабочего времени и назначать инженера по качеству в столбце `verified_by`. Эти ограничения можно реализовать при помощи триггера или ограничения `CHECK`. Они существуют между неключевыми столбцами таблицы и поэтому не удовлетворяют критериям DKNF.

Шестая нормальная форма направлена на устранение всех зависимостей соединений. Обычно она используется для поддержки истории изменений в атрибутах. Например, `Bugs.status` изменяется со временем; возможно, вы захотите сохранить эту историю в дочерней таблице, как и аннотации к изменениям (когда было внесено изменение, кто его внес и, возможно, другие подробности).

Нетрудно представить, что для полной поддержки шестой нормальной формы почти каждый столбец может иметь отдельную таблицу с историей изменений. Это приводит к чрезмерному увеличению количества таблиц и усложняет написание запросов SQL, поскольку для получения простейшего результирующего набора придется соединять множество таблиц.

Шестая нормальная форма не требуется для большинства приложений, но некоторые технологии организации хранилищ данных (например, Anchor Modeling[1]) используют эту форму для реализации *временных баз данных*. Эта форма также может использоваться для запроса любых данных, существовавших в конкретный момент, или для анализа их изменений во времени.

Здравый смысл

Правила нормализации не так запутанны или сложны, как может казаться. Это всего лишь метод сокращения избыточности и улучшения целостности данных, обусловленный здравым смыслом.

Использование этого краткого обзора отношений и нормальных форм при проектировании баз данных поможет повысить их качество в будущих проектах.

---

[1] https://en.wikipedia.org/wiki/Anchor_modeling

Библиография

[BMMM98] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. AntiPatterns. John Wiley & Sons, New York, NY, 1998.

[Bra1 1] Ronald Bradford. Effective MySQL Optimizing SQL Statements. McGraw-Hill, Emeryville, CA, 2011.

[BT21] Silvia Botros and Jeremy Tinley. High Performance MySQL. O'Reilly &Associates, Inc., Sebastopol, CA, 4th edition, 2021.[1]

[Cel04] Joe Celko. Joe Celko's Trees and Hierarchies in SQL for Smarties. Morgan Kaufmann Publishers, San Francisco, CA, 2004.

[Cel05] Joe Celko. Joe Celko's SQL Programming Style. Morgan Kaufmann Publishers, San Francisco, CA, 2005.[2]

[Cod70] Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. Communications of the ACM. 13[6]:377–387, 1970, June.

[Fow03] Martin Fowler. Patterns of Enterprise Application Architecture. Addison-Wesley Longman, Boston, MA, 2003[3].

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Boston, MA, 1995.[4]

[Gol91] David Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. ACM Comput. Surv.. 5–48, 1991, March.

[GP03] Peter Gulutzan and Trudy Pelzer. SQL Performance Tuning. Addison-Wesley, Boston, MA, 2003.

[HLV09] Michael Howard, David LeBlanc, and John Viega. 24 Deadly Sins of Software Security. McGraw-Hill, Emeryville, CA, 2009.

---

[1] Ботрос С., Тинли Д. «MySQL по максимуму. Проверенные стратегии. 4-е изд.». — Санкт-Петербург, издательство «Питер».

[2] Селко Д. «Стиль программирования Джо Селко на SQL».

[3] Фаулер М. «Шаблоны корпоративных приложений».

[4] Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. «Приемы объектно-ориентированного проектирования. Паттерны проектирования». — Санкт-Петербург, издательство «Питер».

[Mar08] Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, Englewood Cliffs, NJ, 2008.[1]

[MC15] Jason Myers and Rick Copeland. Essential SQLAlchemy. O'Reilly & Associates, Inc., Sebastopol, CA, Second edition, 2015.

[Nic21] Daniel Nichter. Efficient MySQL Performance. O'Reilly & Associates, Inc., Sebastopol, CA, 2021[2].

[Rub22] Sam Ruby. Agile Web Development with Rails 7. The Pragmatic Bookshelf, Raleigh, NC, 2022.

[Tro06] Vadim Tropashko. SQL Design Patterns. Rampant Techpress, Kittrell, NC, 2006.

[1] Мартин Р. «Чистый код. Создание, анализ и рефакторинг». — Санкт-Петербург, издательство «Питер».

[2] Нихтер Д. «Настройка производительности MySQL».

КРОК

СОЗДАЕМ НАСТОЯЩЕЕ,
ИНТЕГРИРУЕМ БУДУЩЕЕ

croc.ru

КРОК — технологический партнер с комплексной экспертизой в области построения и развития инфраструктуры, внедрения информационных систем, разработки программных решений и сервисной поддержки.

Центры компетенций КРОК фокусируются на ключевых отраслевых кластерах — промышленность, финансовый сектор, розничные продажи, муниципальное управление, спорт и культура.

Ежегодно сотни проектов КРОК становятся системообразующими для экономики и социально-культурной сферы.
