Билл Карвин

Программирование баз данных SQL

Типичные ошибки и их устранение

ЧИТАЙ!

Учимся на ошибках профессиональных программистов!

Программирование баз данных SQL

Типичные ошибки и их устранение

Билл Карвин

SQL (Structured Query Language) — универсальный компьютерный язык, применяемый для создания, модификации и управления данными в реляционных базах данных. Сильной стороной этого языка можно считать то, что большинство SQL-операторов сравнительно легко переносятся из одной СУБД в другую.

SQL задумывался как средство работы конечного пользователя, но, в конце концов, стал настолько сложным, что превратился в инструмент программиста.

В этой книге описываются проблемы, которые часто возникают при использовании SQL. Автору удалось выявить в самую суть наиболее частых ошибок, которые допускают подавляющее большинство профессиональных программистов. Книга буквально насыщена хорошими практическими советами. Профессионалы, вооруженные описанными здесь методами, получили возможность дальнейшего совершенствования в области SQL.

ЧИТАЙ!

ISBN 978-5-4252-0510-0
9 785425 205100

Программирование баз данных SQL Типичные ошибки и их устранение
ЧИТАЙ!


ПРОФЕССИОНАЛЬНЫЕ
КОМПЬЮТЕРНЫЕ КНИГИ

Bill Karwin

SQL Antipatterns

Avoiding the Pitfalls of Database Programming

The Pragmatic Bookshelf

Raleigh, North Carolina Dallas, Texas

УДК 004.4
ББК 32.973.26-018.1
К 21

Перевод с английского — *М. Райтман*

Карвин Б.

К 21 Программирование баз данных SQL. Типичные ошибки и их устранение / Б. Карвин. — М.: Рид Групп, 2012. —336 с. — (Профессиональные компьютерные книги)

ISBN 978-5-4252-0510-0

В мире существует огромное число книг и интернет-публикаций по языку SQL. Но как отличить хорошие примеры от плохих?

«Программирование баз данных SQL» – продукт многолетней практической работы. Каждая тема здесь раскрывается подробно, а внимание к деталям превосходит ожидания. Хотя книга предназначена не для новичков, любой опытный SQL-программист найдет в ней что-нибудь новое.

Предложенные здесь решения охватывают множество случаев: от традиционных «Не могу поверить, что это опять сделал я» до хитрых сценариев, где оптимальный вариант противоречит догмам, на которых выросли все профессионалы.

УДК 004.4
ББК 32.973.26-018.1

ISBN 978-5-4252-0510-0

Программирование баз данных SQL

Типичные ошибки и их устранение

Билл Карвин

МОСКВА
ЧИТАЙ!
Рид Групп
2012

ОГЛАВЛЕНИЕ

Pragmatic Bookshelf

Наши продукты, а также практические курсы и семинары призваны помочь вам и вашей команде создавать более качественное программное обеспечение и получать от программирования больше положительных эмоций. Дополнительные сведения, а также информацию о последних изданиях см. в Интернете по адресу **www.pragprog.com**.

Эксперт — это человек, совершивший все ошибки, которые можно было совершить в очень узкой сфере деятельности.

Нильс Бор

ГЛАВА 1. ВВЕДЕНИЕ

Я отказался от моей первой работы, связанной с SQL.

Вскоре после того, как я окончил Калифорнийский университет по специальности «компьютеры и информатика», ко мне обратился администратор, трудившийся в том же университете и знавший меня по работам в студенческом городке. Он был владельцем молодой компании, занимавшейся разработкой системы управления базами данных, портируемой между разными UNIX-платформами с использованием скриптов оболочки и соответствующих программных средств, таких как `awk`. На тот момент современные динамические языки, такие как Ruby, Python, PHP и даже Perl, еще не были популярными. Словом, он обратился ко мне, поскольку ему требовался программист для написания программного кода, распознающего ограниченную версию языка SQL и выполняющего определенные функции.

Он сказал: «Мне не требуется поддерживать язык в полном объеме — это потребовало бы слишком большой работы. Мне необходим только один оператор SQL — `SELECT`».

Я не изучал SQL в школе. Базы данных были не так широко распространены, как сегодня, а MySQL и PostgreSQL с открытым исходным кодом еще не появились. Правда, к этому времени я уже разработал несколько приложений и знал кое-что о парсерах, занимаясь во время учебы компиляторами и компьютерной лингвистикой. В общем, я призадумался, принять ли его предложение. Насколько может быть трудным для меня синтаксический анализ одного оператора специализированного языка, такого как SQL?

Я нашел справочник по SQL и сразу обратил внимание, что этот язык отличается от тех, что поддерживают операторы, подобные `if()` и `while()`, присвоение значений переменным, выражения и, возможно, функции. Назвать `SELECT` только одним оператором в этом языке — все равно, что назвать мотор только одной частью автомобиля. Оба высказывания, несомненно, истинны, однако они искажают действительность, потому что не учитывают сложность и глубину упоминаемых объектов. Как я понял, для поддержки выполнения этого единственного SQL-оператора потребовалось бы разработать весь программный код функциональной системы управления реляционной базой данных, а также механизм запросов.

Я отказался от этого предложения — возможности разработать код парсера SQL и механизм СУБД. Администратор существенно сузил рамки данного проекта, возможно, просто не понимая, что должна уметь СУБД.

Как мне кажется, мой прежний опыт работы с SQL характерен для разработчиков программного обеспечения, даже для тех, кто окончил университет по специализации «вычислительная техника». Многие осваивают SQL самостоятельно, изучая эту технологию из соображений самозащиты, когда приходится работать над проектом, где эта технология востребована. Другие языки программирования изучают систематически, как положено. А вот SQL почему-то в большинстве случаев осваивают самостоятельно как любители, так и профессиональные программисты, и даже опытные исследователи с ученой степенью доктора философии.

После того как я изучил SQL, я был удивлен, насколько этот язык отличается от процедурных языков программирования, таких как C, Pascal, shell, и от объектноориентированных языков, подобных C++, Java, Ruby и Python. SQL относится к *описательным языкам программирования*, таким как LISP, Haskell или XSLT. В SQL в качестве основной структуры данных используются *множества*, в то время как в объектноориентированных языках работают с объектами. У разработчиков программного обеспечения, прошедших традиционное обучение, именно на этом основании отсутствует интерес к SQL. Назовем этот феномен *«рассогласованием импедансов»*. В результате многие программисты склоняются к использованию объектно-ориентированных библиотек, чтобы не изучать методы эффективного применения SQL.

Начиная с 1992 года мне пришлось немало поработать с SQL. Я использовал этот язык при разработке приложений, предоставлял техническую поддержку, составлял программы обучения и документацию для СУБД-продукта InterBase. Кроме того, я создал библиотеки для SQL-программирования на языках Perl и PHP. Я ответил на тысячи вопросов, используя почтовые веб-рассылки и блоки новостей. У меня скопилось множество повторяющихся, часто задаваемых вопросов, которые с очевидностью демонстрируют, что разработчики программного обеспечения снова и снова совершают одни и те же ошибки.

1.1. ДЛЯ КОГО ПРЕДНАЗНАЧЕНА ЭТА КНИГА

Моя книга адресована разработчикам программного обеспечения на основе SQL. Ее цель — повысить эффективность их работы при использовании этого языка. Мне не важно, кто вы — начинающий программист или опытный профессионал, я обращаюсь к людям всех уровней квалификации, которые извлекут пользу из данной книги.

Возможно, вы изучили справочник по синтаксису SQL. Теперь вы знаете все выражения оператора `SELECT` и можете выполнить часть работы. По-

степенно вы повысите свое мастерство в области SQL путем изучения других приложений и чтения ряда публикаций. Но как отличить хорошие примеры от плохих? Как можно быть уверенным, что вы изучаете передовой опыт, а не еще один тупиковый подход?

Возможно, некоторые темы, изложенные в книге, хорошо вам известны. Тем не менее я предлагаю по-новому взглянуть на ряд проблем, даже если вам известны решения. Полезно убедиться в том, что вы владеете передовыми методами, еще полезнее закрепить навыки путем знакомства с широко распространенными заблуждениями программистов. Другие темы могут оказаться новыми для вас. Надеюсь, что, прочитав соответствующие разделы, вы сможете улучшить навыки программирования на SQL.

Если вы — опытный администратор баз данных, вам, возможно, уже известны оптимальные способы обхода ловушек SQL, которые описаны в данной книге. Эта книга познакомит вас с точкой зрения разработчиков программного обеспечения относительно данной предметной области. Во взаимоотношениях разработчиков и администраторов баз данных нередко возникают споры, однако взаимное уважение и коллективная работа помогают действовать с большей эффективностью. Прочитав эту книгу, вы сможете познакомить разработчиков программного обеспечения, с которыми приходится иметь дело, с передовыми решениями. Теперь, если вы соберетесь проигнорировать предложенные ими варианты, вы будете способны объяснить, почему.

1.2. ЧТО СОДЕРЖИТСЯ В ЭТОЙ КНИГЕ

Что подразумевается в нашем случае под «типичными ошибками»? Способы решения одной проблемы, которые приводит к другим проблемам. Для краткости предлагаю подобные фрагменты кода называть «антипаттернами».

Антипаттерны широко применяются на практике в самых различных случаях, но для них характерны общие черты. Идея о пригодности антипаттерна может возникнуть под воздействием мнений коллег, книги или статьи, или независимо от них. Кстати, многие антипаттерны задокументированы, как в репозитории моделей Портленда (Portland Pattern Repository), так и в книге «AntiPatterns» [1], написанной Вильямом Дж. Брауном (William J. Brown).

В этой книге описываются ошибки, которые часто совершаются при использовании SQL. О них я узнавал в процессе предоставления технической поддержки, на курсах обучения, разрабатывая с коллегами программное обеспечение и отвечая на многочисленные вопросы на веб-форумах. Многие из этих грубых ошибок я совершил сам. Нет лучшего учителя, чем собственный опыт, приобретенный в результате многочасовых ночных мучений при устранении собственных ошибок.

РАЗДЕЛЫ КНИГИ

Книга содержит четыре раздела, посвященных следующим категориям антипаттернов.

Антипаттерны логической структуры базы данных

Прежде чем приступать к программированию, следует решить, какая информация должна содержаться в базе данных, а также определить оптимальный способ организации и взаимосвязи данных. Эта задача включает планирование таблиц, столбцов и взаимосвязей базы данных.

Антипаттерны физической структуры базы данных

После того как определено, какие данные должны сохраняться, реализуется максимально эффективное управление данными с использованием возможностей технологии СУБД. Эта задача включает определение таблиц и индексов, а также выбор типов данных. Используется язык определения данных SQL, операторы типа `CREATE TABLE`.

Антипаттерны запросов

Необходимо добавить данные в базу, а затем извлечь их. SQL-запросы создаются с помощью *языка манипулирования данными,* операторов `SELECT`, `UPDATE` и `DELETE`.

Антипаттерны разработки приложений

Предполагается, что SQL используется в контексте приложений, написанных на другом языке, таком как C++, Java, Python или Ruby. Существуют правильные и неправильные способы использования SQL в приложении. В этой части книги описываются некоторые распространенные грубые ошибки.

Многие из глав имеют смешные или ассоциативные заголовки, такие как «Магические "бобы"», «Дипломатическая неприкосновенность» или «Боязнь неизвестного». Традиционно присваиваются имена как удачно разработанным моделям, так и антипаттернам. Эти имена служат в качестве метафоры или мнемокода.

В приложении имеется практическое описание некоторой теории реляционных баз данных. Многие из антипаттернов, описываемых в данной книге, являются результатом неправильного понимания теории баз данных.

Анатомия антипаттерна

В каждой главе, описывающей антипаттерны, содержатся следующие подразделы.

Цель

Это задача, которую, возможно, требуется решить. Антипаттерны используются с намерением предоставить решение, но в конечном счете приводят к увеличению количества проблем.

Антипаттерн

В этом разделе описывается природа общего решения и показываются непредвиденные последствия, которые превращают это решение в антипаттерн.

Способы распознавания антипаттерна

Могут существовать некоторые признаки, помогающие определить, когда в разрабатываемом проекте используется антипаттерн. На присутствие антипаттерна могут указывать некоторые типы встречаемых препятствий, разговоры коллег и цитаты из их переписки.

Допустимые способы использования антипаттерна

Правила обычно сопровождаются исключениями. Могут сложиться такие обстоятельства, в которых подход, обычно считающийся антипаттерном, тем не менее является уместным или, по крайней мере, представляется наименьшим злом.

Решение

В этом разделе описываются предпочтительные решения, которые позволяют достичь исходной цели без проблем, возникающих при использовании антипаттерна.

1.3. ЧЕГО НЕТ В ЭТОЙ КНИГЕ

Я не собираюсь давать уроки по синтаксису и терминологии SQL. Существует множество книг и ресурсов в Интернете, содержащих базовые сведения. Я предполагаю, что читатели уже изучили синтаксис SQL в достаточной мере, чтобы пользоваться языком и выполнять определенную работу по программированию. Для многих из тех, кто занимается разработкой приложений на основе баз данных, особенно в Интернете, важны производительность, масштабируемость и оптимизация. Существуют книги, посвященные конкретно вопросам производительности в отношении программирования баз данных. В этой области я рекомендую «SQL Performance Tuning» [9] и второе издание «High Performance MySQL» [15]. И если некоторые из разделов моей книги касаются вопросов производительности, то основное внимание здесь уделяется не этому.

Я пытаюсь представить здесь случаи, которые характерны для любых баз данных, а также решения, которые должны работать с базами данных любых производителей. Язык SQL определяется как стандарт Американского национального института стандартов (ANSI) и Международной организации по стандартизации (ISO). Эти стандарты поддерживаются всеми известными базами данных, так что во всех случаях я описываю использование SQL без привязки к конкретной разработке и пытаюсь явно указывать на расширения SQL конкретного разработчика.

Инфраструктуры доступа к данным и библиотеки объектно-реляционных соответствий служат полезными инструментами, но они не являются основным предметом рассмотрения в данной книге. Большинство примеров программного кода написано мной на PHP и, по возможности, без излишних усложнений. Примеры достаточно просты, чтобы быть реализованными в равной степени в большинстве языков программирования.

Администрирование баз данных и рабочие задачи, такие как масштабирование, установка и конфигурирование сервера, мониторинг, создание резервных копий, анализ журналов и безопасность, являются важными и заслуживают отдельной книги. Однако целевой аудиторией этой книги, на мой взгляд, являются не администраторы баз данных, а в большей степени, разработчики, пользующиеся языком SQL.

В этой книге описываются SQL и реляционные базы данных, а не альтернативные технологии, такие как объектноориентированные базы данных, хранилища пар «ключ-значение», базы данных, ориентированные на обработку столбцов, базы данных, ориентированные на обработку документов, иерархические базы данных, сетевые базы данных, оболочки систем предварительной обработки данных и хранилища семантических данных. Сравнение достоинств и недостатков, а также описание надлежащих применений этих альтернативных решений для управления данными представляло бы интересную задачу, однако это тема для совсем другой книги.

1.4. УСЛОВНЫЕ ОБОЗНАЧЕНИЯ

В следующих разделах описываются некоторые условные обозначения, применяемые в данной книге.

Оформление

Названия антипаттернов выделены **полужирным шрифтом**.

Зарезервированные слова SQL набираются заглавными буквами моноширинным шрифтом, чтобы выделить их из текста, как в случае оператора `SELECT`.

В именах таблиц SQL, также набираемых моноширинным шрифтом, каждое слово начинается с заглавной буквы, например `Accounts` и `BugsProducts`. Имена столбцов SQL, также набираемые моноширинным

шрифтом, состоят из строчных букв, а слова разделяются знаками подчеркивания, например `account_name`.

Строки литералов отпечатаны курсивом, например *bill@example.com*.

Терминология

SQL правильно произносится «эс-кю-эл», а не «си-квел». У меня нет возражений относительно последнего, разговорного, варианта произношения, но я стараюсь использовать первый вариант.

В контексте баз данных слово *«индекс»* относится к упорядоченному набору информации. Предпочтительное множественное число от этого слова — *«индексы»*. В других контекстах «индекс» может означать *«указатель»*, и обычно в качестве множественного числа используется слово *«указатели»*.

В SQL термины *«запрос»* (query) и *«оператор»* (statement) отчасти взаимозаменяемы и представляют собой любую полную выполнимую команду SQL. Для большей ясности, термин *«запрос»* используется для обозначения оператора `SELECT`, а термин *«оператор»* — для всех остальных, включая `INSERT`, `UPDATE` и `DELETE`, а также операторы определения данных.

Диаграммы «объект-отношение»

Диаграммы «объект-отношение» — наиболее распространенный способ схематического представления реляционных баз данных. Таблицы показаны как прямоугольники, а взаимосвязи отображаются в виде линий, соединяющих прямоугольники, с символами на одном из концов отрезка, описывающими кардинальное число взаимосвязи. Примеры см. на рис. 1.1.

1.5. ПРИМЕР БАЗЫ ДАННЫХ

Большинство тем в этой книге рассматриваются на примере базы данных для гипотетического приложения, отслеживающего ошибки. Диаграмма отношений объектов для этой базы данных показана на рис. 1.2. Обратите внимание на три соединения между таблицей `Bugs` и таблицей `Accounts`, представляющие три отдельных внешних ключа.

Следующий язык определения данных показывает, как мною определяются таблицы. В некоторых случаях варианты выбора делаются для примеров, приведенных ниже, так что они, возможно, не всегда являются вариантами выбора, которые выполняются в реальных приложениях. Я пытаюсь использовать только стандартную версию SQL, поэтому пример применим для любой базы данных. Тем не менее здесь появляются некоторые типы данных MySQL, такие как `SERIAL` и `BIGINT`.

Файл примера: *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)
);
```

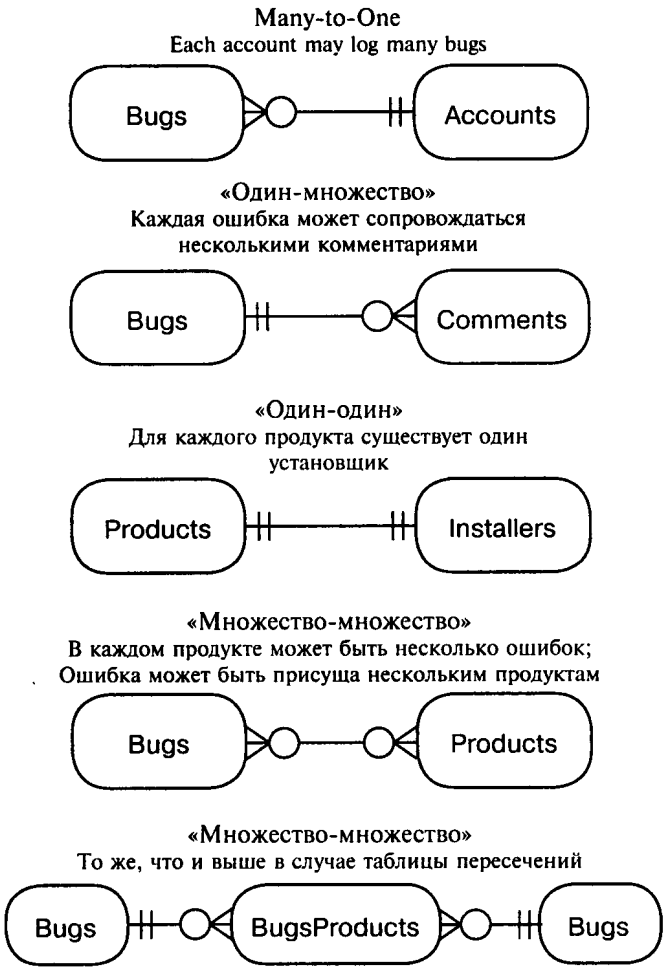


Рис. 1.1. Примеры диаграмм отношений объектов

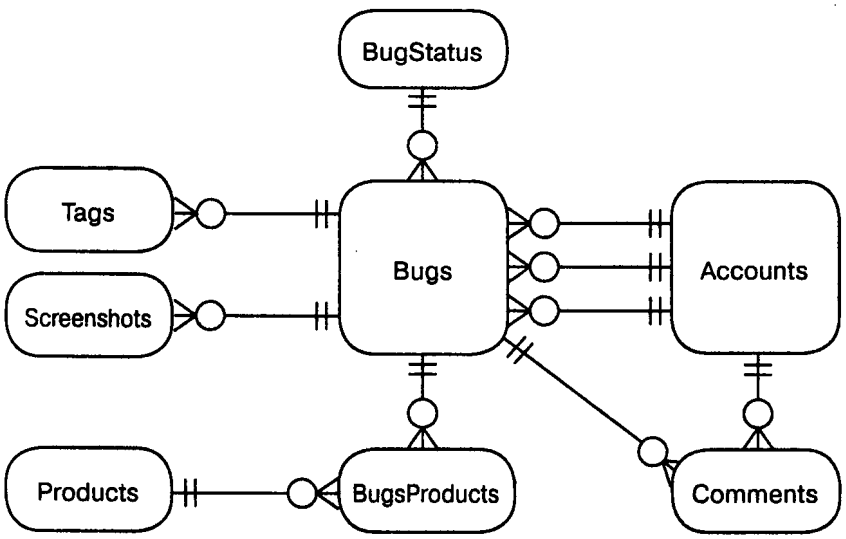


Рис. 1.2. Диаграмма базы данных ошибок

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

CREATE TABLE Bugs (
   bug_id          SERIAL PRIMARY KEY,
   date_reported   DATE NOT NULL,
   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,
   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)
);
```

В некоторых главах, особенно это касается раздела «Антипаттерны логической структуры базы данных», я показываю разные определения базы данных либо для демонстрации антипаттерна, либо для того, чтобы предложить альтернативное решение, которое исключает антипаттерн.

1.6. БЛАГОДАРНОСТИ

Прежде всего, я выражаю признательность моей жене Джен. Я бы не написал эту книгу, если бы не ее любовь и поддержка.

Я хочу также поблагодарить рецензентов, уделивших мне немало времени. Их предложения позволили значительно улучшить книгу. Это Маркус Адамс, Джефф Бин, Фредерик Дауд, Дерби Фелтон, Аджен Ленц, Энди Лестер, Крис Левеск, Майк Набережный, Лиз Нили, Дейв Роер, Марко Романини, Мейк Шмидт, Гейл Стрейни и Дани Торп.

Спасибо моему редактору Жаклин Картер и издателям компании Pragmatic Bookshelf, поверившим в успех данной книги.

ЧАСТЬ I. АНТИПАТТЕРНЫ ЛОГИЧЕСКОЙ СТРУКТУРЫ БАЗЫ ДАННЫХ

Инженер Netscape, чье имя мы не будем называть, однажды передал указатель в JavaScript, сохранил его как строку, а позже передал его обратно в Си, аннулировав 30.

Блейк Росс (Blake Ross)

ГЛАВА 2. БЛУЖДАНИЯ БЕЗ ОРИЕНТИРОВ

Вы разрабатываете функцию в приложении отслеживания ошибок, назначающую пользователя в качестве основного лица, ответственного за продукт. В первоначальном проекте в качестве доверенного лица по каждому продукту допускается только один пользователь. Однако нет ничего удивительного, если вас попросят обеспечить поддержку назначения нескольких пользователей в качестве ответственных лиц по указанному продукту.

Что может быть проще, чем изменение базы данных для хранения списка идентификаторов учетных записей пользователей, разделенных запятыми, вместо одного-единственного идентификатора.

Вскоре начальник обращается к вам с проблемой. «В техническом отделе добавляют персонал для работы над проектами. Они говорят, что могут добавить только пять человек. При попытке добавить больше выводится ошибка. Что происходит?»

Вы киваете: «Да, вы можете перечислять в проекте только определенное число людей», — причем так, как будто это очевидно.

Чувствуя, что начальнику требуется более внятное объяснение, вы добавляете: «Да, от пяти до десяти, возможно, несколько больше. Все зависит от того, насколько старой является каждая учетная запись». У начальника глаза лезут на лоб от удивления. Вы продолжаете: «Я храню идентификаторы учетных записей для проекта в виде списка с разделителями-запятыми. А список идентификаторов должен вмещаться в строку максимальной длины. Если идентификаторы учетных записей короткие, в список можно вместить большее число идентификаторов. Поэтому люди, создавшие более ранние учетные записи, имеют идентификатор, равный 99 или меньшему числу, и такие идентификаторы короче».

Ваш начальник хмурит брови. У вас возникает чувство, что вам придется задержаться на работе.

Программисты обычно используют списки с запятыми-разделителями, чтобы избежать создания таблицы пересечений для отношения «множество-множество». Я называю этот антипаттерн **Блуждания без ориентиров**, так как бесцельные блуждания также представляют собой действие, выполняемое во избежание столкновений.

2.1. ЦЕЛЬ: ХРАНЕНИЕ МНОГОЗНАЧНЫХ АТРИБУТОВ

Когда для столбца в таблице существует одно значение, структура считается простой: можно выбрать тип данных SQL для представления одиночного значения, например `integer`, `date` или `string`. Но как сохранить набор связанных значений в столбце?

В примере базы данных, отслеживающей ошибки, можно было бы связать продукт с контактным лицом, используя столбец `integer` в таблице `Products`. У каждой учетной записи может быть несколько продуктов, и каждый продукт ссылается на одно контактное лицо. Таким образом, между продуктами и учетными записями очевидно наличие отношения *«множество-один»*.

Файл примера: *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);
```

Когда проект продолжает развиваться, у продукта возможно появление нескольких контактных лиц. Помимо отношения «множество-один» между продуктами и учетными записями требуется поддержка отношения «один-множество». В одной строке в таблице `Products` должна быть возможность размещения более одного контакта.

2.2. АНТИПАТТЕРН: ФОРМАТИРОВАНИЕ СПИСКОВ С ЗАПЯТЫМИ-РАЗДЕЛИТЕЛЯМИ

Чтобы минимизировать изменения структуры базы данных, вы решаете переопределить столбец `account_id` как переменную `VARCHAR`, чтобы в этом столбце можно было перечислять несколько идентификаторов учетных записей, разделенных запятыми.

Файл примера: *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');
```

Это решение выглядит правильным, поскольку здесь не создаются дополнительные таблицы или столбцы. Изменен тип данных только одного столбца. Однако давайте рассмотрим проблемы, связанные с производительностью, и проблемы целостности данных, которые характерны для данной структуры таблицы.

Запрос продуктов для конкретной учетной записи

Если объединить все внешние ключи в одно поле, запросы становятся сложными. Невозможно больше использовать равенство, вместо этого требуется проверка по некоторому шаблону. Например, чтобы найти все продукты для учетной записи 12, в базе данных MySQL можно ввести что-то типа этого:

Файл примера: *Jaywalking/anti/regexp.sql*

```
SELECT * FROM Products WHERE account_id REGEXP '[[:<:]]12[[:>:]]';
```

Выражения сравнения с шаблоном могут возвращать ложные совпадения и не позволяют воспользоваться преимуществами индексов. Поскольку синтаксис сравнения с шаблоном зависит от модели базы данных, код SQL не может быть одинаковым для баз данных разных производителей.

Запрос учетных записей для заданного продукта

Подобным образом неудобно и затратно объединять список, разделенный запятыми, с сопоставляемыми строками в справочной таблице.

Файл примера: *Jaywalking/anti/regexp.sql*

```
SELECT * FROM Products AS p JOIN Accounts AS a
   ON p.account_id REGEXP '[[:<:]]' || a.account_id || '[[:>:]]'
WHERE p.product_id = 123;
```

Если объединить две таблицы с помощью выражения, подобного этому, отсутствуют какие-либо шансы использования индексов. Запросом должны выполняться сканирование обеих таблиц, генерирование векторного произведения и оценка регулярного выражения для каждой комбинации строк.

Выполнение агрегированных запросов

В агрегированных запросах используются функции, подобные `COUNT()`, `SUM()` и `AVG()`. Однако эти функции рассчитаны на группы строк, а не на списки с запятыми-разделителями. Приходится обращаться к трюкам, подобным следующему:

Файл примера: *Jaywalking/anti/count.sql*

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

Такие трюки могут быть искусными, но никогда не отличаются ясностью. Подобные решения требуют много времени на разработку и сложны в отладке. Некоторые агрегированные запросы вообще не могут быть выполнены с помощью указанных приемов.

Обновление учетных записей для конкретного продукта

Новый идентификатор можно добавить в конец списка с помощью конкатенации, но эта операция может не оставить список в отсортированном порядке.

Файл примера: *Jaywalking/anti/update.sql*

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

Чтобы удалить элемент из списка, необходимо выполнить два SQL-запроса: один для выборки старого списка и второй для сохранения обновленного списка.

Файл примера: *Jaywalking/anti/remove.php*

```
<?php

$stmt = $pdo->query(
   "SELECT account_id FROM Products WHERE product_id = 123");
$row = $stmt->fetch();
$contact_list = $row['account_id'];

// изменение list на языке PHP
$value_to_remove = «34»;
$contact_list = split(«,», $contact_list);
$key_to_remove = array_search($value_to_remove, $contact_
list);
unset($contact_list[$key_to_remove]);
$contact_list = join(",", $contact_list);

$stmt = $pdo->prepare(
   "UPDATE Products SET account_id = ?
   WHERE product_id = 123");
$stmt->execute(array($contact_list));
```

Как мы видим, для удаления записи из списка требуется пространный фрагмент программного кода.

Проверка достоверности идентификаторов продуктов

Как защититься от ввода пользователем недопустимых записей, таких как *banana*?

Файл примера: *Jaywalking/anti/banana.sql*

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

Пользователи найдут способ ввести любые изменения, и база данных превратится в настоящее месиво. Это не обязательно будут ошибки базы данных, однако данные будут лишены смысла.

Выбор символа в качестве разделителя

Если вместо целочисленных значений сохраняются строковые значения, некоторые записи списка могут содержать символ разделителя. Использование между записями запятой в качестве разделителя может оказаться двусмысленным. Допускается выбор другого символа в качестве разделителя, но можно ли быть уверенным, что этот новый разделитель никогда не появится в записи?

Ограничения на длину списка

Сколько записей списка можно сохранить в столбце VARCHAR(30)? Количество зависит от длины каждой записи. Если длина каждой записи равна двум символам, тогда можно сохранить десять (включая запятые). Однако если каждая запись состоит из шести символов, тогда можно будет сохранить только четыре записи.

Файл примера: *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 можно узнать, что VARCHAR(30) поддерживает список той длины, которая понадобится в будущем? Какая длина будет достаточной? Попробуйте объяснить необходимость в такой длине своему начальнику или клиентам.

2.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Если от членов команды, работающей в проекте, слышны фразы, подобные приведенным ниже, рассматривайте их как признак использования антипаттерна **Блуждания без ориентиров**.

- «Какое наибольшее число записей должно поддерживаться этим списком?»

Этот вопрос возникает, когда пытаются указать максимальную длину столбца VARCHAR.

- «Как сопоставить границу слова в SQL?»

Если для выбора частей строки используются регулярные выражения, это может служить признаком того, что данные части надо хранить отдельно.

- «Какой символ никогда не встретится в записях списка?»

Желательно использовать однозначный символ разделителя, однако следует ожидать, что любой символ может когда-нибудь появиться в значении в списке.

2.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Производительность некоторых видов запросов можно повысить путем применения *денормализации* в отношении структуры базы данных. Примером денормализации может служить хранение списков в виде строки, разделенной запятыми.

Приложению могут требоваться данные в формате с запятыми-разделителями, и при этом отсутствует необходимость в доступе к отдельным элементам в списке. Подобным образом, если приложением принимается формат с запятыми-разделителями из другого источника и требуется просто сохранить полный список в базе данных и позже извлечь его точно в том же формате, нет необходимости в разделении значений.

Если решите использовать денормализацию, придерживайтесь традиций. Начните с использования структуры нормализованной базы данных, поскольку она обеспечивает большую гибкость программного кода приложения и помогает сохранить целостность данных.

2.5. РЕШЕНИЕ: СОЗДАНИЕ ТАБЛИЦЫ ПЕРЕСЕЧЕНИЙ

Вместо хранения `account_id` в таблице `Products` используйте для этих значений отдельную таблицу, чтобы каждое значение этого атрибута занимало свою строку. В этой новой таблице `Contacts` реализуется отношение *«множество-множество»* между продуктами (`Products`) и учетными записями (`Accounts`):

Файл примера: *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);
```

Когда в таблице содержатся внешние ключи, ссылающиеся на две таблицы, она называется *таблицей пересечений*[1]. Она реализует отношение *«множество-множество»* между двумя упоминаемыми таблицами. То есть каждый продукт может быть связан посредством таблицы пересечений с несколькими учетными записями. Аналогично, каждая учетная запись может быть связана с несколькими продуктами. См. диаграмму отношения объектов (рис. 2.1).

Рис. 2.1. Диаграмма отношения объектов таблицы пересечений

Давайте посмотрим, как использование таблицы пересечений позволяет решать все проблемы, представленные в разделе «Антипаттерн».

Запрос продуктов по учетной записи, а также другими способами

Наиболее простой способ запроса атрибутов всех продуктов для заданной учетной записи состоит в объединении таблицы `Products` с таблицей `Contacts`:

Файл примера: *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 возражают против запросов, содержащих операцию объединения, думая, что такие запросы плохо функционируют. Однако этим запросом индексы используются гораздо лучше, чем решением, показанным ранее в разделе «Антипаттерн».

Подобным образом запрос сведений об учетных записях легко прочитать и легко оптимизировать. В отличие от скрытого применения регулярных выражений, запросом эффективно используются индексы для объединения.

---

[1] Для идентификации данной таблицы используются и другие термины, например: объединенная таблица, таблица «множество-множество», таблица сопоставлений. Название не имеет особого значения, поскольку идея остается той же.

Файл примера: *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;
```

Выполнение агрегированных запросов

В следующем примере возвращается количество учетных записей, приходящихся на продукт.

Файл примера: *Jaywalking/soln/group.sql*

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

Число продуктов, приходящихся на каждую учетную запись, можно выразить так же просто:

Файл примера: *Jaywalking/soln/group.sql*

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

Возможны и другие, более сложные отчеты, например продукт с наибольшим количеством учетных записей:

Файл примера: *Jaywalking/soln/group.sql*

```
SELECT c.product_id, c.accounts_per_product
FROM (
   SELECT product_id, COUNT(*) AS accounts_per_product
   FROM Contacts
   GROUP BY product_id
) AS c
HAVING c.accounts_per_product = MAX(c.accounts_per_product)
```

Обновление контактных лиц для конкретного продукта

В списке могут выполняться операции добавления и удаления записей путем вставки и удаления строк в таблице пересечений. Каждая ссылка на продукт хранится в отдельной строке в таблице `Contacts`, так что можно добавить или удалить их по одной за раз.

Файл примера: *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 достоверности идентификаторов продуктов

Для проверки достоверности записей по набору допустимых значений в другой таблице может использоваться внешний ключ. Вы декларируете, что `Contacts.account_id` ссылается на `Accounts.account_id`, и тем самым полагаетесь на базу данных с точки зрения целостности данных на уровне ссылок. Теперь можно быть уверенным, что таблица пересечений содержит только существующие идентификаторы учетных записей.

Для ограничения записей можно также использовать типы данных SQL. Например, если записи в списке должны быть допустимыми значениями `INTEGER` или `DATE` и объявлен столбец, использующий эти типы данных, можно быть уверенным, что все записи являются действующими значениями этого типа (а не бессмысленными записями, такими как *банан*).

Выбор символа разделителя

Поскольку каждая запись хранится в отдельной строке, символ разделителя не используется. Если записи содержат запятые или другие символы, которые могут применяться в качестве разделителя, двусмысленности не возникает.

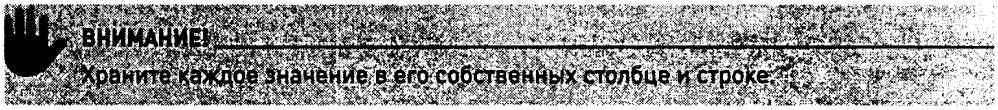
Ограничения на длину списка

Так как каждая запись находится в таблице пересечений в отдельной строке, список ограничен только числом строк, которые могут физически существовать в одной таблице. Если требуется ограничить количество записей, в приложении следует ввести в действие политику, использующую счетчик записей, а не суммарную длину списка.

Другие достоинства таблицы пересечений

При индексировании `Contacts.account_id` производительность выше, чем при сравнении подстрок в списке с запятыми-разделителями. Когда объявляется внешний ключ для столбца, во многих базах данных разных типов создается в неявном виде индекс по этому столбцу (проверьте данное утверждение по документации).

Путем добавления столбцов в таблицу пересечений можно также создать дополнительные атрибуты для каждой записи. Например, можно было бы записать дату, когда контактное лицо было добавлено для заданного продукта, или атрибут, отмечающий, кто является основным контактным лицом в отличие от второстепенных контактных лиц. Это невозможно сделать в списке, разделяемом запятыми.



Дерево и есть дерево — сколько их еще вам надо, чтобы рассмотреть его?

Рональд Рейган

ГЛАВА 3. ПРОСТЫЕ ДЕРЕВЬЯ

Предположим, что вы разрабатываете программное обеспечение для известного веб-сайта, где размещают новости науки и техники.

Это современный веб-сайт, так что пользователи могут добавлять комментарии и даже отвечать друг другу, создавая треды обсуждения, которые ветвятся и образуют сложную иерархическую структуру. Вы выбираете простое решение: отследить эти цепочки ответов. Каждый комментарий ссылается на комментарий, на который он отвечает.

Файл примера: *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 с помощью одного SQL-запроса трудно извлечь длинную цепочку ответов. Можно получить только прямые дочерние объекты или, возможно, объединить комментарий с потомками на фиксированную глубину. Тем не менее треды дискуссий могут обладать *неограниченной* глубиной. Для получения комментариев в заданном треде потребовалось бы выполнить много SQL-запросов.

Другая идея состоит в извлечении *всех* комментариев и сборе их в древовидные структуры данных в памяти приложения, с использованием традиционных алгоритмов деревьев, которые проходят в школе. Но владельцы вебсайта сообщили вам, что они ежедневно публикуют десятки статей и каждая статья может обрастать сотнями комментариев. Непрактично сортировать миллионы комментариев каждый раз, когда кто-нибудь просматривает вебсайт.

Должен существовать оптимальный способ хранения тредов комментариев, позволяющий просто и эффективно извлекать всю цепочку обсуждений.

3.1. ЦЕЛЬ: ХРАНЕНИЕ И ЗАПРОС ИЕРАРХИЙ

Для данных характерно наличие рекурсивных связей. Данные могут быть упорядочены в древовидные или иерархические структуры. В древовидной структуре данных каждая запись называется *узлом*. У узла может быть не-

сколько дочерних объектов и один родительский объект. Верхний узел, у которого нет родителя, называется *корнем*. Нижние узлы, не имеющие дочерних объектов, называются *листьями*. Узлы в середине — это попросту *узлы, не являющиеся листьями*.

Возможно, в предыдущих иерархических данных потребуется запросить отдельные элементы, связанные подмножества коллекции либо всю коллекцию. К примерам древовидных структур данных относятся следующие упорядоченные множества.

Организационная схема. Связь служащих с руководителями — хрестоматийный пример древовидной структуры данных. Она приводится в бесчисленных книгах и статьях по SQL. В организационной схеме у каждого служащего есть руководитель, который представляет *родительский объект* для служащего в структуре дерева. Руководитель в свою очередь также является служащим.

Ветвящееся обсуждение. Как показано во введении, структура дерева может использоваться для цепочки комментариев в ответ на другие комментарии. В дереве родительскими объектами узла комментария являются ответы.

В данной главе пример ветвящегося обсуждения используется, чтобы показать антипаттерн и его решения.

3.2. АНТИПАТТЕРН: ПОСТОЯННАЯ ЗАВИСИМОСТЬ ОТ ОДНОГО РОДИТЕЛЯ

Простое решение, обычно публикунмое в книгах и статьях, заключается в добавлении столбца `parent_id`. Этот столбец ссылается на другой комментарий в той же таблице, и можно создать ограничение внешнего ключа для ввода в действие данной взаимосвязи. SQL-код для определения такой таблицы показан ниже, а схема взаимосвязей объектов приводится на рис. 3.1.

Файл примера: *Trees/anti/adjacency-list.sql*

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

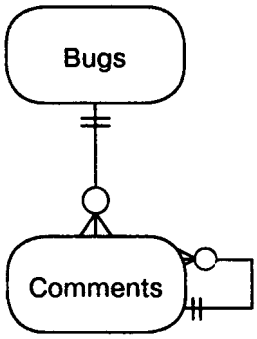


Рис. 3.1. Диаграмма взаимосвязей объектов Списка соседства

Данная структура называется *Списком соседства*. Вероятно, это наиболее распространенная структура, которая используется разработчиками программного обеспечения для хранения иерархических данных. Ниже для примера приводятся некоторые данные, демонстрирующие иерархию комментариев и иллюстрацию дерева, показанного на следующей странице, на рис. 3.2.

comment_id	parent_id	Author	Comment
1	NULL	Фран	В чем причина этой ошибки?
2	1	Олли	Полагаю, это указатель `null`.
3	2	Фран	Нет, я проверил это.
4	1	Кукла	Требуется проверить правильность входных данных.
5	4	Олли	Да, в этом ошибка.
6	4	Фран	Да, добавьте, пожалуйста, эту проверку.
7	6	Кукла	Проверка устраняет ошибку.

Запрос дерева со Списком соседства

Список соседства может быть антипаттерном, когда он является стандартным выбором такого большого числа разработчиков и тем не менее ему не удается стать решением для одной из самых распространенных задач, которую требуется выполнять с деревом: запрос всех потомков.

Комментарий и его прямые дочерние объекты можно извлечь с помощью относительно простого запроса:

Файл примера: *Trees/anti/parent.sql*

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

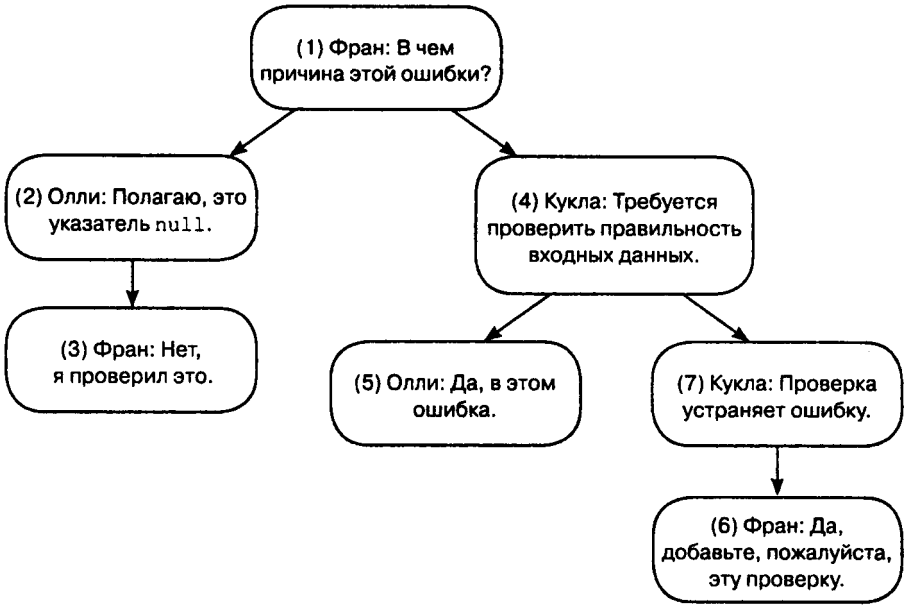


Рис. 3.2. Иллюстрация ветвящихся комментариев

Однако этим кодом запрашиваются только два уровня дерева. В соответствии с одним из свойств дерева оно может простираться на любую глубину, так что требуется возможность организации запроса потомков вне зависимости от количества уровней. Например, может возникнуть необходимость вычислить параметр `COUNT()` для комментариев в треде или параметр `SUM()` стоимости деталей в сборочном узле механического устройства.

Данный тип запроса неудобен, когда используется Список соседства, поскольку каждый уровень дерева соответствует еще одному объединению, а число объединений должно быть фиксированным в SQL-запросе. С помощью следующего запроса извлекается дерево глубиной до четырех уровней, но за пределами этой глубины уровни не могут быть извлечены:

Файл примера: *Trees/anti/ancestors.sql*

```
SEIECT 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
```

Этот запрос тоже неудобный, так как он включает потомков из постепенно углубляющихся уровней путем добавления столбцов. Это затрудняет вычисление составного значения, такого как `COUNT()`.

Другой способ запроса структуры дерева из Списка соседства заключается в извлечении всех строк в коллекции и восстановлении иерархии в приложении до того, как ее можно будет использовать подобно дереву.

Файл примера: *Trees/anti/all-comments.sql*

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

Копирование большого объема данных из базы данных в приложение, прежде чем их можно будет проанализировать, крайне неэффективно. В отдельных случаях может понадобиться лишь поддерево, а не все дерево начиная с корня. Иногда требуется лишь итоговая информация о данных, например параметр `COUNT()` для комментариев.

Обслуживание дерева с помощью Списка соседства

Общеизвестно, что некоторые операции оказываются простыми для выполнения с помощью Списка соседства, например добавление нового узла-листа:

Файл примера: *Trees/anti/insert.sql*

```
INSERT INTO Comments (bug_id, parent_id, author, comment)
  VALUES (1234, 7, 'Кукла', 'Спасибо!');
```

Реализовать перемещение одного узла или поддерева также просто:

Файл примера: *Trees/anti/update.sql*

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

Однако удаление узла из дерева представляет собой более сложную задачу. Если требуется удалить целое поддерево, необходимо сделать несколько запросов, чтобы найти всех потомков. Затем удалить потомков с самого нижнего уровня вверх, чтобы соблюсти требование целостности внешнего ключа.

Файл примера: *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;
```

Можно использовать внешний ключ с модификатором `ON DELETE CASCADE`, чтобы автоматизировать эту процедуру, если известно, что требуется постоянно удалять потомков вместо их продвижения или перемещения.

Если вместо этого необходимо удалить узел, не являющийся листом, и продвинуть его дочерние узлы или переместить их в другое место в дереве, сначала потребуется изменить параметр `parent_id` дочерних узлов, а затем удалить требуемый узел.

Файл примера: *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;
```

Это примеры операций, выполнение которых осуществляется за несколько шагов при использовании конструкции Списка соседства. Для задач, выполнение которых база данных должна сделать простым и эффективным, приходится писать большие фрагменты кода.

3.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Признаком попытки использования антипаттерна **Простые деревья** может быть возникновение одной из следующих ситуаций:

- «Сколько уровней необходимо поддерживать в деревьях?»

Вы стремитесь получить всех потомков или предков узла без использования рекурсивного запроса. Вы идете на компромисс, поддерживая только дере-

вья ограниченной глубины, но напрашивается следующий вопрос: какая глубина деревьев будет достаточна?

- «Постоянно испытываю благоговейный страх, когда надо работать с программным кодом, управляющим структурами данных дерева».

Вами принято одно из наиболее сложных решений по управлению иерархиями, но вы исповедуете неправильный подход. Каждый метод облегчает выполнение некоторых задач, но обычно за счет того, что другие задачи становятся труднее в реализации. Возможно, вы выберете решение, которое не станет лучшим вариантом использования иерархий в вашем приложении.

- «Мне требуется периодически запускать скрипт, чтобы убирать осиротевшие строки в деревьях».

При удалении узлов, не являющихся листьями, приложение создает в дереве разъединенные узлы. Когда в базе сохраняются сложные структуры данных, необходимо поддерживать структуру в согласованном, достоверном состоянии после любого изменения. Можно воспользоваться одним из решений, представляемых ниже, вместе с триггерами и каскадными ограничениями внешнего ключа, чтобы сохранять структуры данных, отличающиеся эластичностью, а не хрупкостью.

3.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Конструкция списка соседства, возможно, прекрасно обеспечит ту работу, которую требуется выполнить в приложении. Достоинство конструкции списка соседства состоит в извлечении прямого родительского или дочернего объекта заданного узла. В этом случае не представляет сложности и вставка строки. Если данные операции — это все, что необходимо делать с иерархическими данными, тогда Список соседства способен хорошо справляться с ними.

НЕ УСЛОЖНЯЙТЕ

Я написал для компьютерного центра обработки данных приложение по отслеживанию производственных ресурсов. Внутри компьютеров скомпоновано некоторое оборудование; например, дисковый контроллер с кэшем установлен в сервере, смонтированном в стойке, а дополнительные модули памяти установлены в контроллере диска.

Мне требовалось SQL-решение, позволяющее легко отслеживать использование иерархических коллекций. Но мне также надо было отслеживать каждую отдельную часть оборудования для создания бухгалтерских отчетов по амортизации, использованию оборудования и рентабельности инвестиций.

Менеджер сообщил, что совокупности могут иметь подмножества и, таким образом, дерево теоретически может распространяться на любую глубину. Потребовалось несколько недель, чтобы улучшить программный код функций обработки деревьев

в хранилище базы данных, пользовательского интерфейса, администрирования и отчетности.

Однако на практике приложению инвентаризации никогда не надо выполнять группирование оборудования с деревом глубже одиночной взаимосвязи «родитель-потомок». Если бы мой клиент подтвердил, что этого будет достаточно для моделирования его требований к инвентаризации, мы могли бы сэкономить усилия, не выполняя достаточно большого объема работы.

РСУБД (реляционные СУБД) некоторых производителей поддерживают расширения SQL с целью поддержки иерархий, хранящихся в формате списка соседства. Стандарт SQL-99 определяет синтаксис рекурсивного запроса, использующего ключевое слово WITH, за которым следует *обыкновенное табличное выражение.*

Файл примера: *Trees/legit/cte.sql*

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

В Microsoft SQL Server 2005, Oracle 11g, IBM DB2 и PostgreSQL 8.4 поддерживаются рекурсивные запросы с использованием обычных табличных выражений, подобных показанным выше.

MySQL, SQLite и Informix пока еще не поддерживают этот синтаксис. То же самое относится к СУБД Oracle 10g, которая по-прежнему широко применяется на практике. Можно было бы предположить, что в будущем синтаксис рекурсивных запросов станет доступным во всех популярных базах данных, и тогда использование списка соседства не будет связано с соответствующими ограничениями.

Базы данных Oracle 9i и 10g поддерживают оператор `WITH`, но не для рекурсивных запросов. Вместо этого существует фирменный синтаксис: `START WITH` и `CONNECT BY PRIOR`. Данный синтаксис подходит для подобных рекурсивных запросов:

Файл примера: *Trees/legit/connect-by.sql*

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

3.5. РЕШЕНИЕ: ИСПОЛЬЗОВАНИЕ АЛЬТЕРНАТИВНЫХ МОДЕЛЕЙ ДЕРЕВА

У списка соседства существует несколько альтернатив, включая *Перечисление путей*, *Вложенные множества* и *Таблицу замыканий*. В следующих трех разделах показываются примеры, где эти конструкции применяются для решения сценария из раздела «Антипаттерн», хранения и запроса древовидной совокупности комментариев.

К этим решениям необходимо привыкнуть. Вначале они могут показаться более сложными, чем список соседства, тем не менее они упрощают некоторые операции с деревьями, которые оказались очень сложными или неэффективными при использовании конструкции списка соседства. Если вашему приложению требуется выполнять такие операции, тогда эти конструкции будут более предпочтительным выбором, чем простой список соседства.

Перечисление путей

Один минус Списка соседства заключается в том, что он неэкономно решает задачу извлечения предков заданного узла в дереве. В методе Перечисления путей эта задача решается путем хранения строки предков как атрибута каждого узла.

Форму Перечисления путей можно наблюдать в иерархии каталогов. UNIX-путь, такой как */usr/local/lib/*, является Перечислением путей файловой системы, где *usr* — родитель папки *local*, которая в свою очередь является родительским объектом для папки *lib*.

В таблице `Comments` вместо столбца `parent_id` определите столбец, именуемый как `path`, в виде длинной переменной `VARCHAR`. Строка, хранящаяся в этом столбце, является последовательностью предков текущей строки в порядке сверху вниз, как UNIX-путь. Можно даже выбрать слеш (/) в качестве символа разделителя.

Файл примера: *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,
  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/	Фран	В чем причина этой ошибки?
2	1/2/	Олли	Полагаю, это указатель null.
3	1/2/3/	Фран	Нет, я проверил это.
4	1/4/	Кукла	Требуется проверить правильность входных данных.
5	1/4/5/	Олли	Да, в этом ошибка.
6	1/4/6/	Фран	Да, добавьте, пожалуйста, эту проверку.
7	1/4/6/7/	Кукла	Проверка устраняет ошибку.

Можно запросить предков путем сравнения пути текущей строки с шаблоном, образованным из пути другой строки. Например, чтобы найти предков комментария № 7, чей путь — 1/4/6/7/, выполните следующий программный код:

Файл примера: *Trees/soln/path-enum/ancestors.sql*

```
SELECT *
FROM Comments AS c
WHERE '1/4/6/7/' LIKE c.path || '%';
```

Этим кодом сравниваются шаблоны, образованные из путей предков `1/4/6/%`, `1/4/%` и `1/%`.

Можно запросить потомков, изменив на обратные аргументы предиката `LIKE`. Чтобы найти потомков комментария № 4, чей путь — `1/4/`, используйте следующий программный код:

Файл примера: *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/`.

Если есть возможность легко выбрать подмножество дерева или цепочки предков вплоть до корня, легко можно выполнить много других запросов, таких как вычисление `SUM()` стоимостей узлов в поддереве или просто вычислить количество узлов. Например, чтобы посчитать комментарии по авторам в поддереве, начиная с комментария № 4, выполните следующий программный код:

Файл примера: *Trees/soln/path-enum/count.sql*

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

Вставка узла похожа на вставку в модели Списка соседства. Узел, не являющийся листом, можно вставить без необходимости изменять какую-либо другую строку. Скопируйте `path` из родительского объекта нового узла и добавьте в эту строку идентификатор нового узла. Если первичный ключ генерирует свое значение автоматически во время вставки, возможно, потребуется вставить строку, а затем обновить `path` после того, как станет известно значение идентификатора (ID) для новой строки. Например, если используется MySQL, встроенная функция `LAST_INSERT_ID()` возвращает самое последнее значение идентификатора (ID), сгенерированное для вставленной строки в текущем сеансе. Получите остальную часть пути из родительского объекта нового узла.

Файл примера: *Trees/soln/path-enum/insert.sql*

```
INSERT INTO Comments (author, comment) VALUES ('Олли', 'Отличная
работа!');

UPDATE Comments
   SET path = (SELECT path FROM Comments WHERE comment_id = 7)
   || LAST_INSERT_ID() || '/'
WHERE comment_id = LAST_INSERT_ID();
```

Метод Перечисления путей обладает некоторыми недостатками, сходными с теми, которые описывались в главе 2. Базой данных не может быть принудительно установлено, что путь сформирован правильно или что значения пути соответствуют существующим узлам. Обслуживание строки пути зависит от кода приложения, и ее проверка является затратным делом. Не имеет значения, как долго создается столбец VARCHAR. Для него по-прежнему существует ограничение по длине, так что, строго говоря, им не поддерживаются деревья неограниченной глубины.

Метод Перечисления путей позволяет легко сортировать набор строк по их иерархии до тех пор, пока элементы между разделителями обладают непротиворечивой длиной[1].

Вложенные множества

Решение Вложенные множества позволяет хранить информацию с каждым узлом, принадлежащим множеству его потомков, а не непосредственному родителю узла. Данная информация может быть представлена путем кодирования каждого узла в дереве с помощью двух номеров, которые можно назвать `nsleft` и `nsright`.

Файл примера: *Trees/soln/nested-sets/create-table.sql*

```
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,
   comment       TEXT NOT NULL,
   FOREIGN KEY (bug_id) REFERENCES Bugs (bug_id),
   FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

Каждому узлу задаются номера `nsleft` и `nsright` следующим способом: номер `nsleft` меньше, чем номера всех дочерних объектов узла, тогда как номер `nsright` больше, чем номера всех дочерних объектов данного узла. Эти номера не связаны со значениями параметра `comment_id`.

Простой способ присвоения этих значений состоит в выполнении первого глубокого прохода по дереву, присвоении номеров `nsleft` с приращением

[1] С другой стороны, это во многом напоминает антипаттерн **Блуждания без ориентиров**.

по мере спуска по ветви дерева и присвоении номеров `nsright` по мере обратного подъема по ветви.

Возможно, модель легче наглядно представить по рис. 3.3, чем из приведенного выше описания.

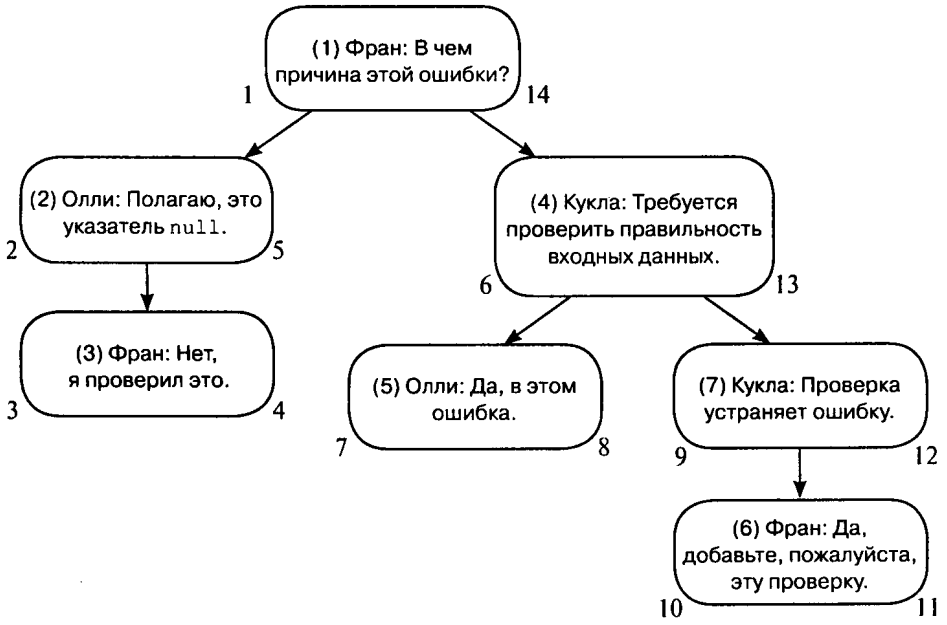


Рис. 3.3. Иллюстрация метода Вложенных множеств

comment_id	Nsleft	nsright	author	comment
1	1	14	Фран	В чем причина этой ошибки?
2	2	5	Олли	Полагаю, это указатель null.
3	3	4	Фран	Нет, я проверил это.
4	6	13	Кукла	Требуется проверить правильность входных данных.
5	7	8	Олли	Да, в этом ошибка.
6	9	12	Фран	Да, добавьте, пожалуйста, эту проверку.
7	10	11	Кукла	Проверка устраняет ошибку.

После того как каждому узлу присвоены эти номера, их можно использовать для поиска предков и потомков любого заданного узла. Например, можно извлечь комментарий № 4 и его потомков путем поиска узлов, номера которых находятся между `nsleft` и `nsright` текущего узла.

Файл примера: *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;
```

Можно извлечь комментарий № 6 и его предков путем поиска узлов, номера которых охватывают номера текущего узла. Например:

Файл примера: *Trees/soln/nested-sets/ancestors.sql*

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

Главное достоинство конструкции Вложенных множеств заключается в том, что когда удаляют узел, не являющийся листом, его потомки автоматически считаются прямыми дочерними узлами родительских объектов удаляемого узла. Хотя правый и левый номера каждого узла, показанные на иллюстрации, имеют значения, образующие непрерывную последовательность, причем разница всегда равна единице, если сравнивать с соседними элементами одного уровня и родителями, это не является обязательным условием поддержания иерархии для конструкции Вложенных множеств. Так что когда в результате удаления узла в значениях возникают пропуски, в структуре дерева не будет разрывов.

Например, можно вычислить глубину заданного узла и удалить его родительский объект; если затем еще раз вычислить глубину узла, окажется, что она уменьшилась на один уровень.

Файл примера: *Trees/soln/nested-sets/depth.sql*

```
-- Выводится значение глубины = 3
SELECT c1.comment_id, COUNT(c2.comment_id) AS depth
```

```
FROM Comment AS c1
   JOIN Comment AS c2
      ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 7
GROUP BY c1.comment_id;

DELETE FROM Comment WHERE comment_id = 6;

-- Выводится значение глубины = 2
SELECT c1.comment_id, COUNT(c2.comment_id) AS depth
FROM Comment AS c1
   JOIN Comment AS c2
      ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 7
GROUP BY c1.comment_id;
```

Однако некоторые запросы, которые просто реализуются в конструкции Списка соседства, например извлечение прямого дочернего узла или прямого родительского узла, оказываются более сложными в конструкции Вложенных множеств. Прямой родитель заданного узла `c1` является предком этого узла, и никакой другой узел не может существовать между ними. Только если такого узла не найдено (т. е. результатом внешнего соединения является значение `null`), предок действительно будет прямым родителем узла `c1`.

Например, чтобы найти прямого родителя комментария № 6, выполните следующий программный код:

Файл примера: *Trees/soln/nested-sets/parent.sql*

```
SELECT parent.*
FROM etwComment AS c
   JOIN Comment AS parent
      ON c.nsleft BETWEEN parent.nsleft AND parent.nsright
   LEFT OUTER JOIN Comment AS in_between
      ON c.nsleft BETWEEN in_between.nsleft AND in_between.
      nsright
```

```
      AND in_between.nsleft BETWEEN parent.nsleft AND parent.
      nsright
WHERE c.comment_id = 6
   AND in_been.comment_id IS NULL;
```

Манипуляции с деревом, вставка и перемещение узлов обычно более сложны в конструкции Вложенных множеств, чем в других моделях. Когда вставляют новый узел, требуется пересчитать все левые и правые значения, которые больше, чем левое значение нового узла.

К ним относятся правые равноправные элементы и родительские объекты нового узла, а также правые равноправные элементы его предков. Сюда также входят потомки, если новый узел вставлен в качестве узла, не являющегося листом. В предположении, что новый узел является листом, необходимо следующее обновление:

Файл примера: *Trees/soln/nested-sets/insert.sql*

```
-- создание места для значений 8 и 9 вложенных множеств
UPDATE Comment
   SET nsleft = CASE WHEN nsleft >= 8 THEN nsleft+2 ELSE nsleft
   END,
      nsright = nsright+2
WHERE nsright >= 7;

-- создание нового дочернего элемента для комментария № 5, за-
нимающего значения 8 и 9 вложенных множеств
INSERT INTO Comment (nsleft, nsright, author, comment)
   VALUES (8, 9, 'Фран', 'Я тоже!');
```

Модель Вложенных множеств оптимальна, когда важно быстро и легко выполнять запросы для поддеревьев, а не операции над отдельными узлами. Вставка и перемещение узлов являются сложными операциями из-за необходимости перенумерации левых и правых значений. Если использование дерева связано с частыми вставками, модель Вложенных множеств не будет наилучшим выбором.

Таблица замыканий

Решение Таблица замыканий — простой и элегантный способ хранения иерархий. Он реализует хранение всех путей по дереву, а не только путей с прямыми связями «родитель-потомок».

Помимо простой таблицы `Comments` создайте еще одну таблицу `TreePaths` с двумя столбцами, каждый из которых является внешним ключом для таблицы `Comments`.

Файл примера: *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,
   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 структуре дерева используйте таблицу `TreePaths`. Храните здесь одну строку для каждой пары узлов, связанных отношением «предок/потомок», даже если они разделяются в дереве несколькими уровнями. Также добавьте строку для каждого узла, чтобы ссылаться на него. Иллюстрация связывания узлов в пары показана на рис. 3.4.

предок	потомок	предок	потомок	предок	потомок
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

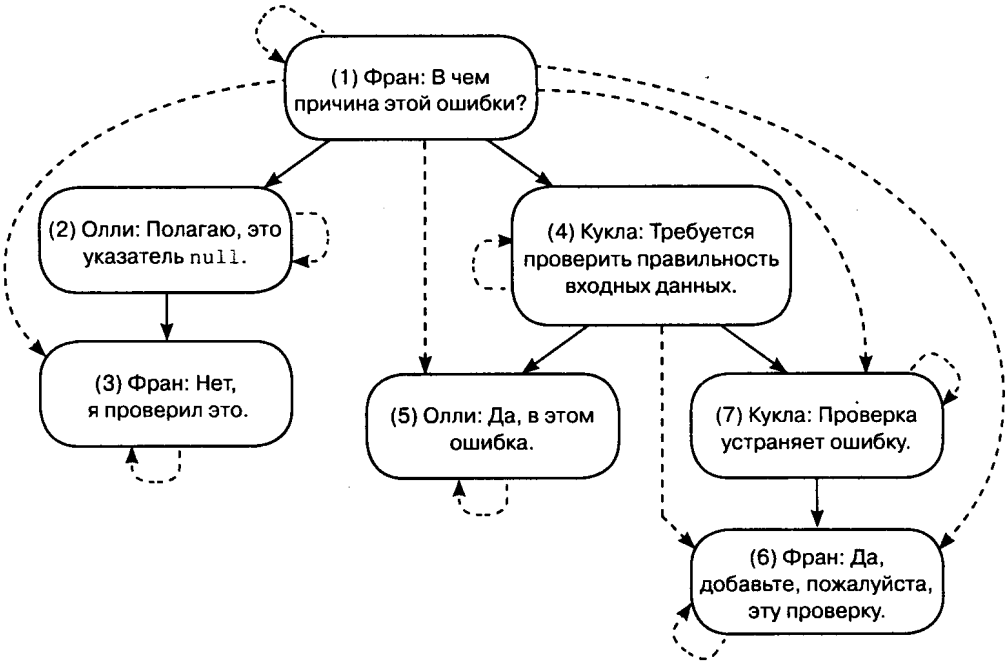


Рис. 3.4. Иллюстрация метода Таблица замыканий

Запросы для извлечения предков и потомков из данной таблицы становятся более простыми, чем запросы в решении Вложенные множества. Чтобы извлечь потомков комментария № 4, сравните строки в `TreePaths`, где `ancestor` — 4:

Файл примера: *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, подберите строки в `TreePaths`, где потомок — 6:

Файл примера: *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 лист, например дочерний элемент для комментария № 5, сначала вставьте строку, ссылающуюся на саму себя. Затем добавьте копию набора строк в TreePaths, которые ссылаются на комментарий № 5 как на descendant (включая строку, где комментарий № 5 ссылается на самого себя), заменив descendant на номер нового комментария:

Файл примера: *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 концевую вершину, например комментарий № 7, удалите все строки в TreePaths, которые ссылаются на комментарий № 7 как на потомка (descendant):

Файл примера: *Trees/soln/closure-table/delete-leaf.sql*

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

Чтобы удалить поддерево полностью, например комментарий № 4 и его потомков, удалите все строки в TreePaths, которые ссылаются на комментарий № 4 как на потомка (descendant), а также все строки, которые ссылаются на любых потомков комментария № 4 как на потомков:

Файл примера: *Trees/soln/closure-table/delete-subtree.sql*

```
DELETE FROM TreePaths
WHERE descendant IN (SELECT descendant
   FROM TreePaths
   WHERE ancestor = 4);
```

Обратите внимание на то, что если удалить строки в TreePaths, то сами комментарии при этом не удаляются. Это кажется странным для данного примера Comments, но такая политика не лишена смысла при работе с другими типами деревьев, например категориями в каталоге изделий или служащими в структурной схеме организации. Не обязательно требуется удалять узел, когда изменяют его взаимосвязи с другими узлами. Хранение путей в отдельной таблице обеспечивает более гибкое выполнение этой задачи.

Чтобы переместить поддерево из одного местоположения на дереве в другое, сначала отсоедините поддерево от его предков, удалив строки, которые ссылаются на предков верхнего узла в этом поддереве и потомков этого узла. Например, чтобы переместить комментарий № 6 из его позиции в качестве дочернего элемента комментария № 4 в дочерний элемент комментария № 3, начните со следующего удаления. Убедитесь, что не удаляется ссылка комментария № 6 на самого себя.

Файл примера: *Trees/soln/closure-table/move-subtree.sql*

```
DELETE FROM TreePaths
WHERE descendant IN (SELECT descendant
   FROM TreePaths
   WHERE ancestor = 6)
AND ancestor IN (SELECT ancestor
   FROM TreePaths
   WHERE descendant = 6
   AND ancestor != descendant);
```

При выборе предков комментария № 6, но не самого комментария № 6 и потомков комментария № 6, включая комментарий № 6, удаляются корректно все пути от предков комментария № 6 до комментария № 6 и его потомки. Другими словами, с помощью этого кода удаляются пути (1, 6), (1, 7), (4, 6) и (4, 7). При этом пути (6,6) и (6, 7) не удаляются.

Затем добавьте осиротевшее поддерево путем вставки строк, соответствующих предкам нового местоположения и потомкам поддерева. Можно использовать синтаксис `CROSS JOIN`, чтобы создать декартово произведение, генерирующее строки, необходимые для сопоставления предков нового местоположения со всеми узлами в поддереве, которое требуется переместить.

Файл примера: *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;
```

Данный программный фрагмент создает новые пути, используя предков комментария № 3, включая сам комментарий № 3, и потомков комментария № 6, включая комментарий № 6. Таким образом, новыми путями явля-

ются (1, 6), (2, 6), (3, 6), (1, 7), (2, 7), (3, 7). В результате выполненных действий поддерево начиная с комментария № 6 становится дочерним элементом комментария № 3. Оператор CROSS JOIN создает все необходимые пути, даже если поддерево перемещается в дереве на более высокий или более низкий уровень.

Конструкция Таблица замыканий более проста, чем Вложенные множества. В обеих конструкциях существуют быстрые и несложные способы запроса предков и потомков, но Таблица замыканий в большей степени упрощает обслуживание иерархий. В обеих конструкциях более удобно запрашивать прямые дочерние или родительские узлы, чем в конструкциях Список соседства и Перечисление путей.

Тем не менее конструкцию Таблица замыканий можно улучшить, чтобы упростить выполнение запросов прямых родительских или дочерних узлов. Добавьте в конструкцию Таблица замыканий атрибут TreePaths.path_lengtn. Значение path_length ссылки узла на самого себя равно нулю, значение path_length прямого дочернего объекта этого узла равно 1, значение path_length внучатого объекта узла равно 2 и так далее. Найти дочерние объекты комментария № 4 теперь просто:

Файл примера: *Trees/soln/closure-table/child.sql*

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

Какой конструкцией следует пользоваться?

Каждая из конструкций обладает своими сильными и слабыми сторонами. Выбирайте нужную в зависимости от операций, которые по вашему усмотрению должны быть самыми эффективными. На рис. 3.5 некоторые операции помечены как простые или трудные для каждой соответствующей конструкции дерева. Проанализируйте достоинства и недостатки каждой конструкции.

- *Список соседства* является самой обыкновенной конструкцией, и многие разработчики программного обеспечения признают ее достоинства.
- *Рекурсивные запросы,* использующие операторы WITH или CONNECT BY PRIOR, делают более эффективным применение конструкции Список соседства при условии, что используется одна из баз данных, поддерживающих этот синтаксис.
- *Перечисление путей* хорошо подходит для встраивания отладочных операторов в пользовательские интерфейсы, но эта конструкция считается

хрупкой, поскольку в ней не удается принудительно установить целостность на уровне ссылок и приходится хранить избыточные данные.

- *Вложенные множества* — умное решение, возможно, даже слишком. Оно также не поддерживает целостность на уровне ссылок. Данное решение оптимально, когда требуется чаще запрашивать дерево, чем его изменять.
- *Таблица замыканий* является самой универсальной и, кстати, единственной конструкцией в этой главе, где допускается принадлежность узла нескольким деревьям. Для хранения взаимосвязей необходима дополнительная таблица. Данная конструкция также требует пространного описания глубоких иерархий, при этом за уменьшение объема вычислений приходится расплачиваться увеличением дискового пространства.

Конструкция	Таблицы	Запрос дочернего элемента	Запрос дерева	Вставка	Удаление	Целостность ссылок
Список соседства	1	Просто	Трудно	Просто	Просто	Да
Рекурсивный запрос	1	Просто	Просто	Просто	Просто	Да
Перечисление путей	1	Просто	Просто	Просто	Просто	Нет
Вложенные множества	1	Трудно	Просто	Трудно	Трудно	Нет
Таблица замыканий	2	Просто	Просто	Просто	Просто	Да

Рис. 3.5. Сравнение конструкций иерархических данных

Существует много учебного материала, посвященного хранению и обработке иерархических данных в SQL. Автор Джо Селко (Joe Celko) написал хорошую книгу «Trees and Hierarchies in SQL for Smarties» [2], где рассматриваются иерархические запросы. Еще одна книга, где описываются деревья и даже графы, — это «SQL Design Patterns» [16] Вадима Тропашко (Vadim Tropashko), написанная в академическом стиле.

ВНИМАНИЕ!
Иерархия состоит из записей и связей.
Для соблюдения требований конкретной работы моделируйте оба элемента.

Существа, находившиеся снаружи, переводили взор со свиньи на человека и с человека на свинью, а потом снова со свиньи на человека; но уже невозможно было сказать, кто есть кто.

Джордж Оруелл «Скотный двор»

ГЛАВА 4. ОБЯЗАТЕЛЬНЫЕ ИДЕНТИФИКАТОРЫ

Недавно я ответил на вопрос, который мне задал разработчик программного обеспечения, пытавшийся предотвратить дублирование строк. Кстати, этот вопрос встречается довольно-таки часто. Сначала я думал, что у него отсутствует первичный ключ. Но проблема была в другом.

В своей базе управления контентом он хранил статьи для публикации на веб-сайте. Он пользовался таблицей пересечений для отношения «множество-множество» между таблицей статей и таблицей тегов.

Файл примера: *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)
);
```

При подсчете количества статей с заданным тегом он из запросов получал неправильные результаты. Ему было известно, что существовало только пять статей с тегом «economy» (экономика), а запрос выдавал, что этих статей семь.

Файл примера: *ID-Required/intro/articletags.sql*

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

Когда разработчик запрашивал все строки, соответствующие идентификатору `tag_id`, он видел, что тег связан с одной определенной статьей в трипликате; одну и ту же связь отображали три строки, хотя у них были разные значения `id`.

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

Данная таблица содержала первичный ключ, но этот первичный ключ не предотвращал появление в столбцах дубликатов, которые оказались причиной беспокойства. Возможным «лекарством» могло бы стать создание ограничения `UNIQUE` по двум другим столбцам, но зачем вообще нужен столбец `id` в этих условиях?

4.1. ЦЕЛЬ: СОЗДАНИЕ СОГЛАШЕНИЙ ПЕРВИЧНОГО КЛЮЧА

Цель состоит в проверке наличия первичного ключа в каждой таблице, однако результатом непонимания природы первичного ключа может быть антипаттерн.

Всем, кто знаком с проектированием баз данных, известно, что первичный ключ — это важная, более того, обязательная часть таблицы. Это действительно так. Наличие первичных ключей характерно для хорошей структуры базы данных. Первичный ключ является гарантированно уникальным для всех строк в таблице, так что он представляет собой логичный механизм адресации отдельных строк и предотвращения хранения дублированных строк. На первичный ключ ссылаются также с помощью внешних ключей для создания табличных связей.

Выбор столбца, который будет служить первичным ключом, является хитроумной задачей. Значение любого атрибута в большинстве таблиц может принадлежать более чем одной строке. Даже фамилия и имя человека с очевидностью подвержены дублированию, о чем говорится в учебниках. Кстати, адрес электронной почты или административные идентификационные номера, такие как номер социального страхования в США или идентификационный номер налогоплательщика, не являются, строго говоря, уникальными.

В таких таблицах необходим новый столбец для хранения искусственного значения, у которого нет значения в домене, моделируемом таблицей. Этот столбец используется в качестве первичного ключа, так что можно обращаться к столбцам уникальным способом, при этом допуская наличие дубликатов в столбце любого другого атрибута, если это уместно. Данный тип столбца первичного ключа иногда называется псевдоключом (*pseudokey*) или свернутым ключом (*surrogate key*).

Чтобы обеспечить возможность присвоения строкам уникальных значений псевдоключей, даже когда одновременно действующие клиенты вставляют новые строки, в большинстве баз данных предоставляется механизм серийного генерирования уникальных целочисленных значений вне области локализации транзакции.

ДЕЙСТВИТЕЛЬНО ЛИ НЕОБХОДИМ ПЕРВИЧНЫЙ КЛЮЧ?

Мне не раз приходилось слышать от некоторых разработчиков программного обеспечения, что их таблице не нужен первичный ключ.
Иногда эти программисты хотят избежать воображаемых трудностей по поддержке уникального индекса, или у них есть таблицы без столбцов, которые можно было использовать для этой цели.
Ограничение первичного ключа важно, когда требуется достичь следующих целей:
предотвращение появления дублированных строк в таблице;
ссылка на отдельные строки в запросах;
поддержка ссылок внешних ключей.
Если ограничения первичного ключа не используются, возникает необходимость в рутинной работе по проверке дубликатов строк.

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

Как часто следует выполнять эту проверку? Что необходимо делать с дубликатом при его обнаружении?
Таблица без первичного ключа подобна упорядочиванию коллекции MP3-музыки без названий песен. Вы прослушиваете музыку, но не можете найти желаемую песню или исключить дублирование в коллекции.

Псевдоключи не были стандартизованы до SQL:2003, так что каждая база данных использовала свое собственное расширение SQL для реализации псевдоключей. Даже терминология, касающаяся псевдоключей, зависит от разработчика базы данных, как показано в нижеследующей таблице:

Функция	Базы данных, которыми поддерживается функция
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

Псевдоключи — полезная функция, но они не представляют собой единственный способ объявления первичного ключа.

4.2. АНТИПАТТЕРН: ОДИН РАЗМЕР ДЛЯ ВСЕХ СЛУЧАЕВ

Книги, статьи и программные инфраструктуры спровоцировали местные обобщенные соглашения о том, что для каждой таблицы базы данных должен существовать столбец первичного ключа со следующими характеристиками.

- Имя столбца первичного ключа — `id`.
- Тип его данных — 32-разрядные или 64-разрядные целые числа.
- Уникальные значения генерируются автоматически.

Наличие столбца с именем `id` в каждой таблице так распространено, что оно стало синонимом первичного ключа. Программисты, изучающие SQL, получают ложное представление о том, что первичный ключ всегда означает столбец, определенный таким способом.

Файл примера: *ID-Required/anti/id-ubiquitous.sql*

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

Добавление столбца `id` в каждую таблицу приводит к появлению нескольких факторов, благодаря которым его использование становится произвольным.

Создание избыточного ключа

Может показаться, что столбец `id` определяется в качестве первичного ключа исключительно ради соблюдения традиции, даже когда в той же таблице может быть определен в качестве обыкновенного первичного ключа еще один столбец. Другой столбец может даже определяться с ограничением `UNIQUE`. Например, в таблице `Bugs` приложение может помечать ошибки с использованием строки с мнемоникой проекта, к которому они относятся, или с помощью другой идентифицирующей информации.

Файл примера: *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', 'аварии при сохранении', ...);
```

Применение столбца bug_id в предыдущем примере аналогично применению столбца id, поскольку он также позволяет уникально идентифицировать каждую строку.

Разрешение дублированных строк

Составной ключ формируется из нескольких столбцов. Обычно составной ключ применяется в таблице пересечений, такой как BugsProducts. Первичный ключ должен гарантировать однократное появление в таблице любой указанной комбинации значений bug_id и product_id, даже если каждое из этих значений встречается в разных парах по несколько раз.

Однако когда в качестве первичного ключа используется обязательный столбец id, ограничение больше не действует в отношении двух столбцов, которые должны быть уникальными.

Файл примера: *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); -- присутствие дубликатов разрешается
```

Наличие дубликатов в данной таблице пересечений приводит к непредсказуемым результатам при использовании таблицы для сопоставления Bugs с Products. Чтобы предотвратить появление дубликатов, можно было бы объявить ограничение UNIQUE по двум столбцам помимо столбца id:

Файл примера: *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)
);
```

Но если по какой-то причине необходимо уникальное ограничение по этим двум столбцам, столбец `id` становится избыточным.

Неоднозначность ключа

У слова *код* есть несколько определений, одно из которых — способ передачи сообщения в сжатом или засекреченном виде. В программировании цель должна быть противоположной — сделать значение более понятным.

Имя `id` является общим, оно не имеет определенного значения. Это становится особенно очевидным, когда объединяют две таблицы, и у них одинаковые имена столбцов первичных ключей.

Файл примера: *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';
```

Как отличить столбец `id` ошибки от столбца `id` учетной записи в коде приложения, если ссылка на столбцы выполняется по имени, а не по порядку положения? Эта проблема особенно остро проявляется в динамических языках, таких как PHP, когда результатом запроса является ассоциативный массив: если в запросе не заданы псевдонимы столбцов, один столбец перезаписывает другой.

Имя столбца `id` не помогает сделать запрос более ясным. Но если столбцам были присвоены имена `bug_id` и `account_id`, прочитать результаты запроса будет намного легче. Первичный ключ применяется для адресации строк таблицы, поэтому имя столбца должно служить подсказкой о типе объекта в данной таблице.

Использование ключевого слова `USING`

Возможно, вы знакомы с SQL-синтаксисом операции объединения, использующей ключевые слова `JOIN` и `ON`, находящиеся перед выражением оценки сравниваемых строк в двух таблицах.

Файл примера: *ID-Required/anti/join.sql*

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

Языком SQL поддерживается также более лаконичный синтаксис для выражения объединения двух таблиц. Если у столбцов одинаковые имена в обеих таблицах, предыдущий запрос можно перезаписать следующим образом:

Файл примера: *ID-Required/anti/join.sql*

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

Однако если требуется определить первичный ключ псевдоключа с именем `id` во всех таблицах, тогда столбец внешнего ключа в зависимой таблице ни в коем случае не может использовать то же имя, что и первичный ключ, на который он ссылается. Вместо этого приходится применить более многословный синтаксис `ON`:

Файл примера: *ID-Required/anti/join.sql*

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

Трудности при использовании составных ключей

Некоторые разработчики отказываются от составных ключей, поскольку, как они утверждают, эти ключи очень трудны в применении. Любое выражение, с помощью которого ключ сравнивается с другим ключом, должно обеспечивать сравнение всех столбцов. Внешний ключ, который ссылается на составной первичный ключ, сам должен быть составным внешним ключом. При применении составных ключей пользователь должен вручную набивать массу символов.

СПЕЦИАЛЬНАЯ ОБЛАСТЬ ДЕЙСТВИЯ ПОСЛЕДОВАТЕЛЬНОСТЕЙ

Можно выделить для новой строки значение путем присвоения наибольшего значения, используемого в текущий момент, с добавлением к нему единицы.

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

Такой алгоритм ненадежен, когда есть клиенты, одновременно запрашивающие последующее значение. В этом случае оба клиента могли бы использовать одно и то же значение. Такая ситуация называется **race condition**.

Чтобы этого избежать, необходимо блокировать параллельные вставки во время чтения текущего максимального значения и последующего его использования

в новой строке. Чтобы сделать это, необходимо заблокировать всю таблицу. Блокирования на уровне строк недостаточно. Блокировки таблицы создают узкое место, так как они вынуждают параллельно работающих клиентов становиться в очередь на доступ к таблице.

Последовательности решают эту проблему путем функционирования вне области действия транзакций. Они никогда не присваивают одно и то же значение нескольким клиентам, поэтому присвоение значения никогда не откатывается независимо от того, зафиксировано значение в строке или нет. Из этого следует, что несколько клиентов могут генерировать уникальные значения параллельно и при этом быть уверенными, что они не будут использовать одно и то же значение.

Большинство баз данных поддерживают некую функцию для возврата последнего значения, сгенерированного последовательностью. Например, в MySQL такая функция называется `LAST_INSERT_ID()`, в Microsoft SQL Server используется `SCOPE_IDENTITY()`, а в Oracle применяется функция *ИмяПоследовательности*`.CURRVAL()`.

Эти функции возвращают значение, сгенерированное во время текущего сеанса работы, даже если другие клиенты параллельно генерируют свои собственные значения. При этом не возникает вопроса, кто быстрее.

Данное право первого выбора напоминает ситуацию, когда математик отказывается от использования двухмерных и трехмерных координат, выполняя расчеты в одномерном пространстве. Разумеется, подобная идеализация упростила бы геометрию и избавила нас от тригонометрии, но при этом не удалось бы описать объекты реального мира.

4.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Признак антипаттерна здесь легко распознать: в таблицах для первичного ключа слишком часто используется обобщенное имя `id`. Практически не существует причин, почему это имя столбца следует предпочесть другому имени, более описательному.

На присутствие антипаттерна могут также указывать следующие ситуации.

- «Я не думаю, что мне необходим первичный ключ в этой таблице».

Разработчик, который так утверждает, путает термин *«первичный ключ»* с *псевдоключом*. В каждой таблице должно существовать *ограничение* первичного ключа, чтобы предотвратить дублирование строк и однозначно идентифицировать отдельные строки. Возможно, возникнет желание вместо этого использовать естественный ключ или составной ключ.

- «Как я умудрился получить дублированные ассоциации «множество-множество»?»

Таблица пересечений для отношения «множество-множество» должна объявлять ограничение первичного ключа или, по крайней мере, ограничение *уникального ключа* по множеству столбцов внешних ключей.

- «Я читал, что по теории баз данных следует перемещать значения в таблицу поиска и ссылаться на них по идентификаторам. Но я не хочу делать это, так как потребуется выполнять операцию объединения каждый раз, когда мне будут нужны фактические значения».

Это распространенное неправильное понимание части теории проектирования баз данных, которая называется нормализацией и у которой в действительности нет ничего общего с псевдоключами. Дополнительные сведения по этому вопросу см. в приложении А.

4.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Некоторые объектно-реляционные интегрированные среды упрощают разработку, устанавливая определенные правила для конфигурации. Ожидается, что первичный ключ в каждой таблице определяется одним и тем же способом: как столбец псевдоключа целочисленного типа с именем `id`. Если используется такая интегрированная среда, возможно, возникнет желание соблюдать ее соглашения, так как это обеспечивает доступ к другим требуемым функциям интегрированной среды.

Нет ничего страшного и в использовании псевдоключа или присвоении значений с помощью механизма автоматического увеличения целых чисел на единицу. Но не всякой таблице нужен псевдоключ, кроме того, не каждый псевдоключ обязательно надо именовать как `id`.

Псевдоключ — хороший выбор в качестве заменителя естественного ключа, слишком длинного, чтобы быть практичным. Например, для таблицы, где записываются атрибуты файла в файловой системе, путь файла мог бы быть хорошим естественным ключом, но потребовалось бы затратить немало ресурсов, чтобы индексировать таблицу из строк такой длины.

4.5. РЕШЕНИЕ: СПЕЦИАЛЬНАЯ ПОДГОНКА

Первичный ключ — это ограничение, а не тип данных. Можно объявить первичный ключ по любому столбцу или набору столбцов, если типы данных поддерживают индексирование. Также должна существовать возможность определения столбца в качестве автоинкрементного целочисленного значения без объявления его первичным ключом таблицы. Эти два понятия независимы друг от друга.

При правильном проектировании не следует принимать соглашения, лишенные гибкости.

Присваивайте названия, отражающие суть явления

Выбирайте для первичного ключа осмысленное имя. Имя должно указывать на тип объекта, который идентифицируется первичным ключом. Например, первичный ключ таблицы `Bugs` должен бы называться `bug_id`.

Используйте такое же имя столбца во внешнем ключе там, где это возможно. Часто это означает, что имя первичного ключа должно быть уникальным в пределах используемой схемы; никакие две таблицы не должны использовать одно и то же имя для своих первичных ключей, если только один из них не является также внешним ключом, ссылающимся на другой первичный ключ. Однако существуют исключения: иногда внешнему ключу уместно присвоить другое имя, отличное от имени первичного ключа, на который ссылается внешний ключ, например с целью описания характера ассоциации.

Файл примера: *ID-Required/soln/foreignkey-name.sql*

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

Для описания соглашений по именованию метаданных существует отраслевой стандарт ISO/IEC 11179[1] — руководство для «управления схемами классификации» в системах информационных технологий. Другими словами, это принятый порядок разумного именования таблиц и столбцов. Как и большинство стандартов ISO, данный документ трудно назвать понятным, но Джо Селко (Joe Celko) применяет его в отношении языка SQL на практике в своей книге SQL Programming Style [3].

Стремитесь к нетрадиционным решениям

Предполагается, что в объектно-реляционных интегрированных системах вы должны использовать псевдоключ с именем `id`, но при этом разрешается переопределить это правило и объявить другое имя вместо стандартного. В следующем примере используется интегрированная среда разработки Ruby on Rails:[2]

---

[1] **metadata-standards.org/11179/**

[2] «Rails» и «Ruby on Rails» — товарные знаки Дэвида Хайнемайера Ханссона (David Heinemeier Hansson).

Файл примера: *ID-Required/soln/custom-primarykey.rb*

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

Некоторые разработчики полагают, что указание столбца первичного ключа необходимо только при поддержке баз данных устаревших версий, где они не могут использовать предпочитаемые ими соглашения. В действительности поддержка осмысленных имен столбцов важна и в новых проектах.

Применяйте естественные ключи и составные ключи одновременно

Если таблица содержит атрибут, который гарантированно будет уникальным, отличным от значения `Null` и может служить для идентификации строки, не обязательно ради соблюдения традиции добавлять псевдоключ.

С практической точки зрения нередко каждый атрибут в таблице подвергается изменениям или не является уникальным. Базы данных имеют тенденцию развиваться на протяжении срока жизни проекта, и ответственные лица могут не соблюдать требование неприкосновенности естественного ключа. Бывает, что столбец, который на первый взгляд мог быть хорошим естественным ключом, содержит допустимые дубликаты. В таких случаях единственным решением будет псевдоключ.

Используйте составные ключи там, где они уместны. Когда строка лучше всего идентифицируется по комбинации нескольких столбцов атрибутов, как в таблице `BugsProducts`, используйте эти столбцы в составном первичном ключе.

Файл примера: *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: дублированная запись
```

Обратите внимание, что внешние ключи, ссылающиеся на составной первичный ключ, также должны быть составными. Может показаться странным дублирование этих столбцов в зависимых таблицах, но столбцы могут обладать и достоинствами: они позволяют упрощать запрос, нуждающийся в операции объединения для выборки атрибутов строки, на которую выполняется ссылка.

ВНИМАНИЕ!

Соглашения хороши, только если они полезны.

Победоносные воины сначала выигрывают сражение, и затем идут на войну, тогда как проигравшие воины сначала идут на войну, а затем пытаются победить.

Сун Тзю

ГЛАВА 5. ЗАПИСИ БЕЗ КЛЮЧЕЙ

«Билл, похоже, два менеджера зарезервировали один и тот же сервер в нашей лаборатории на одни и те же дни. Как это могло случиться?» С таким вопросом ввалился в мой рабочий кабинет менеджер испытательной лаборатории. «Не можешь ли ты вникнуть в это дело и разрешить недоразумение? Они кричат на меня, заявляя, что им обоим нужно оборудование и что я срываю график выполнения их проекта».

Несколько лет назад я с помощью СУБД MySQL разработал приложение, отслеживающее использование оборудования. Стандартным механизмом хранения для MySQL была система MyISAM, которая не поддерживала ограничения внешнего ключа. База данных содержала много логических взаимосвязей, но не позволяла принудительно установить целостность на уровне ссылок.

В процессе развития проекта и применения новых способов обработки данных возникла проблема: когда целостность на уровне ссылок не обеспечивалась, в отчетах появлялись несоответствия, не согласовывались промежуточные суммы и расписания велись параллельно.

Руководитель проекта попросил меня написать скрипты контроля качества, которые можно было бы периодически исполнять, чтобы узнать, когда возникают несоответствия. Данные скрипты анализировали состояние базы данных, находили ошибки, такие как висячие строки в дочерних таблицах, и отправляли электронные сообщения с отчетами о них.

Эти скрипты должны были проверять все взаимосвязи таблиц. По мере роста объема данных и количества таблиц росло также число запросов контроля качества, и на исполнение скриптов уходило больше времени. Отчеты, отправляемые по электронной почте, также увеличивались в объеме. Наверное, и вам знакома такая ситуация?

Решение на основе скриптов, разумеется, действовало, правда, оно напоминало заново изобретенный дорогостоящий велосипед. Что мне требовалось, так это найти способ вызывать отказ приложения каждый раз, когда пользователем передавались недопустимые данные. Угадайте, что делают ограничения внешнего ключа?

5.1. ЦЕЛЬ: УПРОЩЕНИЕ АРХИТЕКТУРЫ БАЗЫ ДАННЫХ

В структуре реляционной базы данных взаимосвязям между таблицами уделяется столько же внимания, сколько и самим таблицам. *Целостность на уровне ссылок* — важная составляющая надлежащей структуры и правильного функционирования базы данных. Когда объявляется ограничение внешнего ключа для столбца или набора столбцов, значения из этих столбцов должны существовать в столбцах первичного ключа или уникального ключа родительской таблицы. На первый взгляд все это достаточно просто.

Тем не менее некоторые разработчики программного обеспечения рекомендуют избегать ограничений целостности на уровне ссылок. В качестве причин упоминают следующие.

- Обновления данных могут конфликтовать с ограничениями.
- Используется структура базы данных, настолько гибкая, что она не может поддерживать ограничения целостности на уровне ссылок.
- Считается, что индекс, создаваемый базой данных для внешнего ключа, отрицательно влияет на производительность.
- Используется разновидность базы данных, которой не поддерживаются внешние ключи.
- Необходимо искать в справочниках синтаксис объявления внешних ключей.

5.2. АНТИПАТТЕРН: ПРОПУСК ОГРАНИЧЕНИЙ

Даже если вам кажется, что игнорирование ограничений внешнего ключа делает структуру базы данных более простой, более гибкой или более скоростной, за это приходится расплачиваться. Вам придется написать программу для обеспечения целостности на уровне ссылок.

Предположение существования безупречного программного кода

Для многих решение по обеспечению целостности на уровне ссылок состоит в написании программного кода, обеспечивающего соблюдение взаимосвязей данных во всех случаях. Каждый раз, когда вставляют строку, проверьте, чтобы значения в столбцах внешнего ключа ссылались на существующие значения в таблице, на которую указывают ссылки. Каждый раз, когда удаляют строку, убедитесь, что все дочерние таблицы также обновляются соответствующим образом. Другими словами, придется следовать очевидному совету: *не делайте ошибок.*

Чтобы исключить ошибки целостности на уровне ссылок в случае, когда отсутствуют ограничения внешнего ключа, потребуется выполнить допол-

нительные запросы SELECT до применения изменений, для того чтобы убедиться, что изменение не приведет к появлению нарушенных ссылок. Например, чтобы вставить новую строку, следовало бы убедиться в существовании родительской строки:

Файл примера: *Keyless-Entry/anti/insert.sql*

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

Затем может быть добавлена ошибка, которая ссылается на строку:

Файл примера: *Keyless-Entry/anti/insert.sql*

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

Чтобы удалить строку, следовало бы убедиться в отсутствии дочерних строк:

Файл примера: *Keyless-Entry/anti/delete.sql*

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

Затем можно было бы удалить учетную запись:

Файл примера: *Keyless-Entry/anti/delete.sql*

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

Что случится, если пользователь с account_id 1 вклинится и введет новую ошибку в интервале времени после вашего запроса и до удаления этой учетной записи? Это может показаться невозможным, но согласно знаменитой фразе Гордона Летвина (Gordon Letwin), разработчика DOS 4: «Один шанс на миллион, что следующий день будет вторником». В такой ситуации остается нарушенная ссылка — ошибка, выводимая учетной записью, которой больше не существует.

Единственным выходом будет явная блокировка таблицы Bugs во время ее поверки и разблокирование таблицы после завершения удаления учетной записи. Любая архитектура, в которой требуется такой тип блокирования, не будет отвечать предъявляемым требованиям, когда необходимы параллельное выполнение нескольких операций и масштабируемость.

Проверка наличия ошибок

В антипаттерне, описанном в данной главе, используются написанные разработчиком скрипты для вывода отчетов о поврежденных данных.

Например, в нашей базе данных ошибок столбец Bugs.status ссылается на таблицу поиска BugStatus. Чтобы найти ошибки с недопустимым зна-

чением статуса, можно было бы воспользоваться запросом, подобным следующему:

Файл примера: *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;
```

Представьте себе, что вам требуется написать аналогичный запрос для каждого ссылочного отношения в базе данных.

Если у вас есть привычка проверять нарушенные ссылки, возникает вопрос, как часто надо выполнять эти проверки? Выполнение сотен проверок каждый день или даже чаще превращается в раздражающую рутину.

Что происходит, когда обнаруживается нарушенная ссылка? Можно ли ее исправить? Иногда такая возможность есть. Например, можно было бы изменить значение статуса недопустимой ошибки на целесообразное стандартное значение[1].

Файл примера: *Keyless-Entry/anti/set-default.sql*

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

Есть и другие случаи, когда невозможно синтезировать данные для исправления таких типов ошибок. Например, столбец `Bugs.reported_by` должен ссылаться на учетную запись пользователя, сообщившего о заданной ошибке, но если это значение недопустимо, какую пользовательскую учетную запись следует выбрать в качестве замены?

«Это не моя ошибка!»

Такого не бывает, чтобы весь программный код, касающийся базы данных, был совершенным. Сходные обновления базы данных можно было бы легко выполнить с помощью нескольких функций в приложении. Как можно быть уверенным в применении совместимых изменений для каждого случая в приложении, когда требуется изменить код?

Кроме того, бывают пользователи, которые вносят изменения непосредственно в базу данных, используя инструмент SQL-запросов или с помо-

[1] Как показано, языком SQL поддерживается ключевое слово `DEFAULT`.

щью отдельных скриптов. Нарушенные ссылки легко внедрить посредством специальных SQL-операторов. Кстати, это может случиться на любом этапе жизненного цикла приложения.

Необходимо, чтобы база данных была *непротиворечивой*. Иными словами, в базе данных в любой момент времени должна обеспечиваться работоспособность ссылок. Однако нельзя быть уверенным, что все приложения и скрипты, обращавшиеся к базе данных, выполнили изменения корректно.

Изменения Уловки-22

Многие разработчики избегают ограничений внешнего ключа, так как эти ограничения делают неудобным обновление связанных столбцов во многих таблицах. Например, если необходимо удалить строку, от которой зависят другие строки, потребуется удалить сначала дочерние строки, чтобы избежать нарушения ограничений внешнего ключа:

Файл примера: *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'; -- успешная по-
пытка
```

Необходимо написать множество операторов, по одному для каждой дочерней таблицы. Если добавить еще одну дочернюю таблицу в планируемое расширение базы данных, потребуется скорректировать программный код, чтобы выполнить удаление и из новой таблицы. Но эту проблему можно решить.

Неразрешимая проблема возникает, когда выполняется операция `UPDATE` для столбца, от которого зависят дочерние строки. Нельзя обновить дочерние строки до обновления родительского объекта, и невозможно обновить родительский объект до обновления значений дочерних элементов, которые ссылаются на него. Требуется выполнить оба изменения одновременно, но это невозможно сделать, используя два отдельных обновления. Это скрипт Уловки-22.

Файл примера: *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 эти скрипты трудными для управления, поэтому они решают вообще не использовать внешние ключи. Ниже будет показано, как внешние ключи просто и эффективно справляются с многотабличными обновлениями и удалениями.

5.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

О возможном применения на практике антипаттерна **Записи без ключей** могут свидетельствовать следующие высказывания.

- «Как выполнить запрос для проверки значения, которое существует в одной таблице и отсутствует в другой?»

Обычно эта задача заключается в поиске висячих дочерних строк, чей родительский объект был обновлен или удален.

- «Имеется ли быстрый способ проверки существования значения в одной таблице как части моей вставки во вторую таблицу?»

Эта задача состоит в проверке существования родительской строки. Проверка выполняется автоматически внешним ключом, который использует любой индекс в родительской таблице, чтобы сделать проверку максимально эффективной.

- «Внешние ключи? Мне говорили не использовать их, так как они замедляют работу базы данных».

Производительность часто служит оправданием, однако она обычно создает больше проблем, чем решает, включая проблемы с самой производительностью.

5.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Иногда вынужденно приходится работать с конкретной базой данных, которая не поддерживает ограничения внешнего ключа (например, механизм хранения MyISAM в СУБД MySQL или SQLite до версии 3.6.19). Если вы столкнулись именно с таким случаем, тогда надо найти способ корректировки, подобный скриптам контроля качества, описанным выше.

Существуют также некоторые сверхгибкие структуры баз данных, где внешние ключи не позволяют моделировать взаимосвязи. Должны быть серьезные основания для применения другого антипаттерна SQL, если нельзя использовать традиционные ограничения по целостности на уровне ссылок. Дополнительные сведения см. в главах 6 и 7.

5.5. РЕШЕНИЕ: ОБЪЯВЛЕНИЕ ОГРАНИЧЕНИЙ

Японская фраза «пока-йоке» переводится как «защита от ошибок»[1]. Этим термином обозначают производственный процесс, который помогает исключить дефекты путем предупреждения, исправления или привлечения внимания к ошибкам по мере их возникновения.

Такой порядок позволяет повысить качество и уменьшить потребность в исправлениях, затраты на которые намного превышают расходы, связанные с применением данного метода.

Принцип «пока-йоке» можно применить к проектированию базы данных путем использования ограничений внешних ключей для принудительного ввода в действие целостности на уровне ссылок. Вместо того чтобы искать и исправлять ошибки целостности данных, можно в первую очередь предотвращать появление этих ошибок.

Файл примера: *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, а также специальные запросы подчиняются одним и тем же ограничениям, так что никак не получится обойти установленные правила с помощью какого-нибудь забытого кода или «с заднего крыльца». База данных отклоняет любое неправильное изменение независимо от того, откуда оно пришло.

Применение внешних ключей исключает необходимость написания ненужного кода и гарантирует, что весь программный код функционирует единообразно при изменении базы данных. Это сокращает время на разработку кода, а также время, планируемое на отладку и обслуживание. В среднем по отрасли программного обеспечения на каждые 1000 строк кода приходится от 15 до 50 ошибок. При прочих равных условиях чем меньше строк, тем меньше ошибок.

---

[1] Фраза «пока-йоке» введена в обиход промышленным инженером доктором Сигео Синго (Shigeo Shingo) в его исследовании производственной системы компании «Тойота».

Поддержка многотабличных изменений

Внешние ключи обладают еще одной функцией, которую нельзя имитировать с помощью программного кода, — *каскадные обновления*.

Файл примера: *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
);
```

Данное решение предоставляет возможность обновления или удаления родительской строки, а также позволяет базе данных обрабатывать дочерние строки, которые ссылаются на нее. Обновления родительских таблиц `BugStatus` и `Accounts` распространяются автоматически на дочерние строки в `Bugs`. Проблемы Уловки-22 больше не существует.

Способ декларирования `ON UPDATE` и `ON DELETE` в ограничении внешнего ключа позволяет контролировать результат каскадной операции. Например, ключевое слово `RESTRICT` для внешнего ключа по столбцу `reported_by` означает, что учетную запись нельзя удалить, если на нее ссылаются какие-либо строки в `Bugs`. Ограничением блокируется удаление и генерируется ошибка. Если удалить значение `status`, все ошибки, связанные с этим статусом, автоматически сбрасываются в значение статуса по умолчанию.

В любом случае обе таблицы изменяются базой данных атомарно. Ссылки внешнего ключа соблюдаются как до, так и после изменений.

Если добавить в базу данных новую дочернюю таблицу, внешними ключами в дочерней таблице будет диктоваться каскадное поведение. Изменять программный код приложения не требуется. Не требуется также ничего изменять в отношении родительской таблицы независимо от того, сколько дочерних таблиц ссылается на нее.

Непроизводительные издержки? Их нет!

Это правда, что ограничения внешнего ключа связаны с небольшими непроизводительными издержками. Но по сравнению с возможными альтернативами внешние ключи оказываются намного эффективнее.

- Не нужны запросы `SELECT`, чтобы выполнить проверку до вставки, обновления или удаления.
- Нет необходимости блокирования таблицы для защиты многотабличных изменений.
- Не надо периодически выполнять скрипты контроля качества для корректировки неизбежных висячих строк.

Внешние ключи легко использовать, они повышают производительность и помогают обслуживать непротиворечивую целостность на уровне ссылок во время любых изменений данных, как простых, так и сложных.

ВНИМАНИЕ!

С помощью ограничений защитите свою базу данных от ошибок.

Если попытаться разобрать кошку, чтобы увидеть, как она работает, первое, что окажется у вас в руках, — это неработающая кошка.

Дуглас Адамс

ГЛАВА 6. EAV (ОБЪЕКТ-АТРИБУТ-ЗНАЧЕНИЕ)

«Как посчитать число строк по дате?» Это пример простой задачи для программиста баз данных. Данное решение описывается в любом учебнике по SQL. Решением используется базовый синтаксис SQL:

Файл примера: *EAV/intro/count.sql*

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

Однако простое решение базируется на двух предположениях:

- Значения хранятся в одном и том же столбце, как в `Bugs.date_reported`.
- Значения могут сравниваться одно с другим, так что с помощью оператора `GROUP BY` можно точно сгруппировать даты с одинаковыми значениями.

А что если данные предположения не верны? Что если даты, хранящиеся в столбце `date_reported` или `report_date`, или в столбце с любым другим именем, будут различными в каждой строке? Что если даты представляются в разных форматах, и компьютеру не удается сравнить две даты?

При использовании антипаттерна **Объект-Атрибут-Значение** могут встретиться эти и другие проблемы.

6.1. ЦЕЛЬ: ПОДДЕРЖКА АТРИБУТОВ ПЕРЕМЕННЫХ

Часто целью программных проектов является расширяемость. Хотелось бы разрабатывать программное обеспечение, которое могло бы плавно адаптироваться к будущим потребностям с минимальным перепрограммированием или вовсе без дополнительных трудозатрат.

Проблема не нова. Различные доводы против недостаточной гибкости метаданных реляционных баз выдвигались постоянно с 1970 года, когда реляционная модель была впервые предложена Е. Ф. Коддом (E. F. Codd) в работе «A Relational Model of Data for Large Shared Data Banks» [4].

Обычная таблица состоит из столбцов атрибутов, которые содержат значения для каждой строки в таблице, так как все строки представляют экземпляры сходных объектов. Разные наборы атрибутов представляют разные типы объекта, так что они принадлежат разным таблицам.

Тем не менее в современных объектноориентированных моделях программирования разные типы объектов могут быть связаны друг с другом, например, путем расширения одного и того же базового типа. В объектноориентированном проектировании эти объекты считаются экземплярами одного базового типа, а также экземплярами их соответствующих подтипов. Желательно было бы хранить объекты в виде строк в одной таблице базы данных, чтобы упростить сравнения и расчеты по нескольким объектам. Но желательно также предоставить объектам каждого подтипа возможность хранения их соответствующих столбцов атрибутов, которые могут не действовать в отношении базового типа или других подтипов.

Давайте воспользуемся примером из нашей базы данных ошибок. На рис. 6.1 можно заметить, что у `Bug` и `Feature Request` некоторые атрибуты являются общими, отображаемыми в базовом типе `Issue`. Каждая проблема связана с лицом, сообщившим о ней. Проблема также связана с продуктом, и она обладает приоритетом расширения. Однако у `Bug` есть некоторые индивидуальные атрибуты: версия продукта, в котором возникла ошибка, уровень серьезности или степень воздействия ошибки. Подобным образом, у `FeatureRequest` также могут быть свои собственные атрибуты. Для данного примера предположим, что функция связана со спонсором, из бюджета которого поддерживается разработка этой функции.

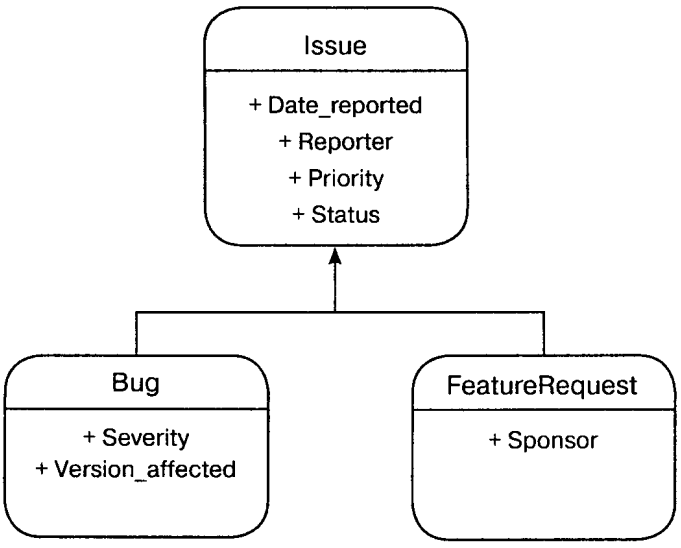


Рис. 6.1. Схема объектноориентированных классов для типов ошибок

6.2. АНТИПАТТЕРН: ИСПОЛЬЗОВАНИЕ ТАБЛИЦЫ ОБЩИХ АТРИБУТОВ

Существует решение, которое состоит в создании второй таблицы, где атрибуты хранятся в виде строк. Подобный ход привлекает некоторых программистов, когда им требуется обеспечить поддержку атрибутов переменных. См. схему с двумя таблицами на рис. 6.2. Каждая строка в таблице атрибутов содержит три столбца:

- *Entity* (Объект). Обычно это внешний ключ для родительской таблицы, содержащей по одной строке для каждого объекта.
- *Attribute* (Атрибут). Это просто имя столбца в обыкновенной таблице, однако в новом проекте требуется идентифицировать атрибут в каждой заданной строке.
- *Value* (Значение). У каждого объекта есть значение для каждого из его атрибутов.

Например, заданная ошибка является объектом, идентифицируемым по значению 1234 его первичного ключа. У нее есть атрибут с именем `status`. Значение этого атрибута для ошибки 1234 равно *NEW*.

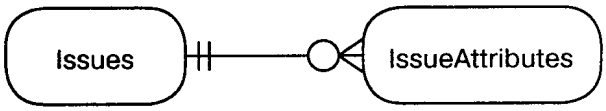


Рис. 6.2. Взаимосвязь объектов EAV (Объект-Атрибут-Значение)

Данная структура называется «Entity-Attribute-Value» (объект-атрибут-значение) или сокращенно *EAV*. Иногда она упоминается как *«открытая схема»*, *«бессхемная»* или *«пары имя-значение»*.

Файл примера: *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',           'НОВАЯ'),
      (1234, 'description'       'Сохранение не работает'),
      (1234, 'reported_by',      'Билл'),
      (1234, 'version_affected', '1.0'),
      (1234, 'severity',         'потеря функциональности'),
      (1234, 'priority',         'высокий');
```

Путем добавления еще одной таблицы можно получить следующие преимущества:

- обе таблицы содержат несколько столбцов;
- для поддержки новых атрибутов число столбцов не должно увеличиваться;
- исключается хаос столбцов, содержащих значение NULL в строках, где атрибут неприменим.

Такая структура выглядит более совершенной. Однако простота структуры базы данных не компенсирует усложнение работы с ней.

Запрос атрибута

Вашему начальнику требуется сгенерировать отчет по ошибкам, о которых выводились сообщения за день. При обычной табличной структуре таблица `Issues` содержала бы простой столбец атрибутов, такой как `date_reported`. Чтобы запросить все ошибки с датами сообщений об этих ошибках, начальник мог бы использовать простой запрос, подобный следующему:

Файл примера: *EAV/anti/query-plain.sql*

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

Чтобы получить те же самые сведения, используя EAV-структуру, начальнику потребуется выбрать строки из таблицы `IssueAttributes`, в которой хранится атрибут, именуемый с помощью строки *date_reported*. Данный запрос будет более многословным и менее понятным.

Файл примера: *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 приходится жертвовать многими преимуществами, которые обеспечиваются структурой традиционной базы данных.

Невозможность создания обязательных атрибутов

Чтобы помочь начальнику сгенерировать точные отчеты по проекту, следует также потребовать, чтобы у атрибута `date_reported` было значение. В обыкновенной структуре базы данных было бы просто установить обязательный столбец, объявив его как `NOT NULL`.

В EAV-структуре каждый атрибут соответствует в таблице `IssueAtributes` строке, а не столбцу. Требуется ограничение, которым проверяется существование строки для каждого значения `issue_id`, а для строки должна существовать символьная строка *`date_reported`* в ее столбце *`attr_name`*.

Однако языком SQL не поддерживается ограничение, с помощью которого можно было бы сделать это. Поэтому необходимо написать код приложения для принудительного ввода ограничения. Если будет найдена ошибка без даты сообщения о ней, следует ли добавить значение для этого атрибута? Какое значение надо ему присвоить? Если сделать предположение или использовать какое-нибудь стандартное значение для отсутствующего атрибута, как это повлияет на точность отчетов?

Невозможность использования типов данных SQL

Начальник сообщает вам, что у него возникли проблемы с генерированием отчета, так как даты вводились в разных форматах, а порой задавалась символьная строка, не являвшаяся датой. В обычной базе данных это можно предотвратить, если объявить столбец с типом данных `DATE`.

Файл примера: *EAV/anti/insert-plain.sql*

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

В EAV-структуре тип данных столбца `IssueAttributes.attr_value` является обычно символьной строкой для размещения всех возможных атрибутов в одном столбце. Поэтому здесь нет способа отбраковки недопустимых данных.

Файл примера: *EAV/anti/insert-eav.sql*

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

Некоторые пытаются расширить EAV-структуру путем определения отдельного столбца `attr_value` для каждого типа данных SQL, оставляя значение `NULL` в неиспользуемых столбцах. Это дает возможность использовать типы данных, но дополнительно усложняет запросы:

Файл примера: *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 пользовательских типов данных или доменов потребуется добавить еще больше столбцов.

Невозможность принудительного ввода целостности на уровне ссылок

В обычной базе данных можно ограничить диапазон некоторых атрибутов путем определения внешнего ключа для таблицы поиска. Например, атрибут `status` ошибки или проблемы должен быть одним из значений короткого списка в таблице `BugStatus`.

Файл примера: *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 EAV-структуре невозможно наложить этот тип ограничений на столбец `attr_value`. Ограничение целостности на уровне ссылок применяется в таблице к каждой строке.

Файл примера: *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 данное ограничение, им будет принудительно устанавливаться соответствие *каждого* атрибута значению в `BugStatus`, а не только атрибута `status`.

Невозможность создания имен атрибутов

Отчеты вашего начальника все еще не заслуживают доверия. Обнаружено, что атрибуты не именуются единообразно. Одной ошибкой используется атрибут, именуемый символьной строкой *date_reported*, а другой ошибкой атрибут именуется с помощью символьной строки *report_date*. Очевидно, что оба имени предназначены для представления одной и той же информации.

Как сосчитать ошибки по дате?

Файл примера: *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;
```

Как узнать, хранится ли заданной ошибкой атрибут по еще одному имени? Как узнать, хранится ли ошибкой заданный атрибут дважды, под двумя разными именами? Как можно предотвратить подобные ошибки?

Одним из спасительных средств может быть объявление внешнего ключа по столбцу `attr_name` для таблицы поиска, в которой содержатся одобренные имена атрибутов. Однако данным средством не поддерживаются атрибуты, определяемые на лету для каждого объекта. Все это присуще EAV-структуре.

Восстановление строки

Рассмотрим несложный пример извлечения из таблицы `Issues` строки со всеми ее атрибутами, расположенными в столбцах. Требуется выбрать проблему в одной строке так, как если бы она хранилась в стандартной таблице.

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

В связи с тем, что каждый атрибут хранится в отдельной строке таблицы `IssueAttributes`, для извлечения их всех как части одной строки требуется объединение для каждого атрибута. На момент написания данного запроса необходимо знать все атрибуты. Следующий запрос позволяет восстановить строку, показанную выше:

Файл примера: *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_re-
ported'
   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 = 'priori-
      ty';
   LEFT OUTER JOIN IssueAttributes AS i4
      ON i.issue_id = i4.issue_id AND i4.attr_name = 'descrip-
      tion';
WHERE i.issue_id = 1234;
```

Необходимо использовать внешние объединения, так как внутренние объединения приведут к тому, что запросом не будут возвращаться какие-либо

строки, если в таблице `IssueAttributes` отсутствует какой-то из атрибутов. По мере увеличения числа атрибутов растет число объединений, и стоимость данного запроса возрастает экспоненциально.

6.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

На использование антипаттерна **EAV (Объект-Атрибут-Значение)** указывают приводимые ниже высказывания, которые можно услышать от членов команды разработчиков.

- «Эта база данных является полностью расширяемой без изменения метаданных. Новые атрибуты можно определить во время выполнения».

Реляционные базы данных не поддерживают такой уровень гибкости. Когда кто-то заявляет, что разработал произвольно расширяемую базу данных, вероятно, он использует EAV-структуру.

- «Какое максимальное число объединений можно сделать в запросе?»

Если необходим запрос для поддержки такого большого числа объединений, что возникает необходимость расширения пределов базы данных, возможно, есть проблема в структуре базы данных. Возникновение такой проблемы характерно для EAV-структуры.

- «Не понимаю, как написать отчет для нашей коммерческой электронной платформы? Необходимо нанять консультанта, чтобы он сделал это для нас».

По-видимому, во многих готовых программных пакетах на основе базы данных, разработанных в соответствии с требованиями заказчика, используется EAV-структура. Это делает большинство стандартных запросов для подготовки отчетов очень сложными или непрактичными.

6.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Применение антипаттерна **EAV (Объект-Атрибут-Значение)** в реляционной базе данных трудно оправдать. Потребуется пожертвовать слишком многими функциями, которые признаны сильными сторонами реляционной парадигмы. Но это не касается законного требования поддержки динамических атрибутов в некоторых программах.

Большинство приложений, которым требуются бессхемные данные, в действительности нуждаются в них только для нескольких таблиц или всего для одной таблицы. Остальные требования к данным соответствуют стандартным табличным структурам. Если принять во внимание дополнительную работу и повышенный риск использования EAV в проекте, возможно, что применение данного антипаттерна в виде исключения будет наимень-

шим злом. Однако учтите: опытные консультанты по базам данных утверждают, что системы, использующие EAV, в течение года становятся недопустимо громоздкими.

Если есть необходимость в управлении нереляционными данными, лучшим решением будет использование *нереляционной* технологии. Эта книга посвящена SQL, а не альтернативам SQL, поэтому ниже перечисляются лишь отдельные примеры таких технологий.

- *Berkley DB* — популярное хранилище пар «ключ-значение», которое легко вставляется в разнообразные приложения.

www.oracle.com/technology/products/berkeley-db/

- *Cassandra* — распределенная база данных, ориентированная на использование столбцов, разработанная для веб-сайта Facebook и применявшаяся в проекте Apache.

incubator.apache.org/cassandra/

- *CouchDB* — база данных, ориентированная на использование документов, является распределенным хранилищем пар «ключ-значение», в котором значения кодируют в формате JSON.

couchdb.apache.org/

- *Hadoop* и *HBase* составляют СУБД с открытым исходным кодом, которая работает по алгоритму MapReduce компании Google, применяемому для распределения запросов в слабоструктурированных крупномасштабных хранилищах данных.

hadoop.apache.org/

- *MongoDB* — база данных подобная CouchDB; ориентирована на работу с документами.

www.mongodb.org/

- *Redis* — база данных подобная CouchDB; ориентирована на работу с документами.

code.google.com/p/redis/

- *Tokyo Cabinet* — хранилище пар «ключ-значение», разработанное в духе СУБД POSIX DBM, GNU GDBM и Berkley DB.

1978th.net/

Можно перечислить много других нереляционных проектов. Однако недостатки подхода EAV в реляционных базах данных характерны и для альтернативных подходов. Когда метаданные изменчивы, трудно формулировать простые запросы. Приложения тратят немало времени на раскрытие структуры данных и их адаптацию под обнаруженные структуры.

6.5. РЕШЕНИЕ: МОДЕЛИРОВАНИЕ ПОДТИПОВ

Если EAV представляется правильной структурой, следует еще раз взглянуть на нее, прежде чем реализовать. Например, если выполнить какой-нибудь анализ в духе старых времен, возможно, вы обнаружите, что данные проекта могут моделироваться в структуре традиционной таблицы более легко и с большей гарантией целостности данных.

Существует несколько способов хранения таких данных без применения EAV. Большинство решений работает лучше всего тогда, когда существует конечное число подтипов и известен атрибут каждого подтипа. Какое решение будет оптимальным для применения, зависит от того, как предполагается запрашивать данные, поэтому решение о структуре следует принимать по каждому конкретному случаю.

Наследование одиночной таблицы

Простейшая структура обеспечивает хранение всех связанных типов в одной таблице, с отдельными столбцами для каждого атрибута, существующего в каком-либо типе. Используйте один атрибут для определения подтипа заданной строки. В этом примере атрибуту присвоено имя `issue_type`. Некоторые атрибуты являются общими для всех подтипов. Многие атрибуты зависят от подтипов, и этим столбцам должны присваиваться значения `NULL` во всех строках, хранящих объекты, к которым не применяется атрибут. Со временем столбцы со значениями, отличными от `NULL`, становятся разреженными.

Название этой структуры взято из книги Мартина Фоулера (Martin Fowler) «Patterns of Enterprise Application Architecture» [6].

Файл примера: *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), -- ОШИБКА или ФУНКЦИЯ
   severity           VARCHAR(20), -- только для ошибок
   version_affected   VARCHAR(20), -- только для ошибок
   sponsor            VARCHAR(50),-- только для запросов функ-
   ций
```

```
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

По мере ввода новых типов объектов база данных должна вмещать атрибуты, которыми описываются эти новые типы объектов. При добавлении отдельных атрибутов для новых типов объектов требуется изменить таблицу для добавления дополнительных столбцов. Со временем можно столкнуться с фактическим ограничением числа столбцов в таблице.

Другим ограничением **Наследования одиночной таблицы** является отсутствие метаданных, необходимых для определения подтипов, которым принадлежат те или иные атрибуты. В используемом приложении можно проигнорировать некоторые атрибуты, если известно, что они не применяются к подтипам объектов в заданной строке. Однако необходимо отслеживать вручную, какие атрибуты применимы для каждого подтипа. Если это возможно, следует отдать предпочтение определению применимости атрибутов в базе данных с помощью метаданных.

Наследование одиночной таблицы оптимально, когда существует несколько подтипов и несколько атрибутов, характерных для подтипов, при этом требуется использовать шаблон доступа к базе данных на основе одиночной таблицы, сходный с активной записью.

Наследование конкретной таблицы

Другое решение заключается в создании отдельной таблицы для каждого подтипа. Во всех таблицах содержатся одни и те же атрибуты, которые являются общими для базового типа, а также соответствующий атрибут, характерный для подтипа. Название данной структуры также взято из книги Мартина Фоулера.

Файл примера: *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 функ-
   ций
   FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
   FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

Преимущество подхода **Наследование конкретной таблицы** по сравнению с подходом **Наследование одиночной таблицы** заключается в том, что предотвращается хранение строки, содержащей значения атрибутов, которые не применяются к подтипу этой строки. Если выполняется ссылка на столбец атрибутов, который отсутствует в этой таблице, пользователь автоматически уведомляется базой данных об ошибке. Например, столбец `severity` не отображается в таблице `FeatureRequests`:

Файл примера: *EAV/soln/insert-concrete.sql*

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

Другое преимущество подхода **Наследование конкретной таблицы** — отсутствие необходимости в дополнительном атрибуте для определения подтипа в каждой.

Однако трудно отличить общие атрибуты от атрибутов, характерных для подтипов. К тому же если добавить новый атрибут в набор общих атрибутов, необходимо изменить все таблицы подтипов.

Отсутствие метаданных показывает, что данные, хранящиеся в таблицах этих подтипов, принадлежат к связанным объектам. То есть если программист, не знакомый с вашим проектом, взглянет на определения таблиц, он увидит, что некоторые столбцы являются общими для всех таблиц подтипов, но метаданные не позволяют ему узнать, существует ли здесь какая-то логическая связь или сходство таблиц обусловлено простой случайностью.

Когда требуется искать все объекты независимо от их подтипов, задача усложняется, если каждый подтип хранится в отдельной таблице. Чтобы упростить данный запрос, определите представление, являющееся объединением таблиц, путем выбора только общих атрибутов.

Файл примера: *EAV/soln/view-concrete.sql*

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

Структура **Наследование конкретной таблицы** оптимальна в том случае, когда редко возникает необходимость в запросе одновременно всех подтипов.

Наследование таблицы классов

Третье решение имитирует наследование примерно так, как если бы таблицы были объектноориентированными классами. Создайте одну таблицу для базового типа, содержащую атрибуты, общие для всех подтипов. Затем для каждого подтипа создайте еще одну таблицу с первичным ключом, который служит также в качестве внешнего ключа для базовой таблицы. Название данной структуры также взято из книги Мартина Фоулера.

Файл примера: *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)
);
```

Взаимнооднозначное соответствие принудительно устанавливается метаданными, поскольку внешний ключ зависимой таблицы является также первичным ключом и соответственно должен быть уникальным. Данное решение предлагает эффективный способ поиска по всем подтипам до тех пор, пока поиск ссылается только на атрибуты базового типа. Как только найдены записи, соответствующие условиям поиска, можно получить характерные для подтипов атрибуты путем выполнения запроса по таблицам соответствующих подтипов.

Не обязательно знать из строки в базовой таблице, какой подтип представляется строкой. До тех пор, пока число подтипов невелико, можно написать объединение по всем одновременно, создавая разреженный набор результатов, как в таблице структуры **Наследование одиночной таблицы**. Атрибуты равны NULL там, где атрибут не применяется в подтипе для заданной строки.

Файл примера: *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);
```

Этот скрипт является также хорошим кандидатом для определения представления (VIEW).

Данная структура оптимальна, когда приходится часто выполнять запрос по всем подтипам, ссылаясь на столбцы, которые являются общими для подтипов.

Слабоструктурированные данные

Если в вашем распоряжении есть много подтипов или если необходимо часто обеспечивать поддержку новым атрибутам, можно добавить столбец BLOB для хранения данных в форматах XML или JSON, которые позволяют кодировать как имена атрибутов, так и их значения. Эту модель Мартин Фоулер назвал *Сериализованный LOB*.

Файл примера: *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), -- ОШИБКА или ФУНКЦИЯ
      attributes     TEXT NOT NULL, -- все динамические атрибуты
      для строки
   FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
   FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

Преимущество данной структуры — ее полная расширяемость. Новые атрибуты можно сохранить в столбце blob в любой момент времени. В каждой строке хранится потенциально индивидуальный набор атрибутов, поэтому в вашем распоряжении столько же подтипов, сколько строк.

Недостаток здесь в том, что в SQL предоставляется слабая поддержка доступа к конкретным атрибутам в такой структуре. Невозможно запросто выбрать отдельные атрибуты в столбце blob для ограничения, основанного на строках, вычисления составного значения, сортировки и других операций. Требуется выбрать весь столбец blob атрибутов в качестве одиночного значения и написать прикладной код, чтобы декодировать и интерпретировать атрибуты.

Данная структура оптимальна, когда нельзя ограничиться конечным набором подтипов и когда требуется полная гибкость для определения новых атрибутов в любой момент времени.

Постобработка

Трудности с EAV-структурой могут возникнуть не по вашей вине. Например, если наследуется проект и его нельзя изменить, или если ваша компания приобрела у сторонней организации программную платформу, которая использует EAV. Если сложилась такая ситуация, ознакомьтесь с проблемными областями в разделе «Антипаттерн», чтобы можно было предвидеть и планировать дополнительную работу, которая обусловлена данной структурой.

Прежде всего, не пытайтесь писать запросы, которыми выбираются записи в виде одиночной строки, как в случае хранения данных в обычной таблице. Вместо этого запросите атрибуты, связанные с записью, и выберите их как набор строк в том виде, в каком они хранятся.

Файл примера: *EAV/soln/post-process.sql*

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

Результат данного запроса может выглядеть подобно следующей таблице:

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	НОВАЯ

Этот запрос проще написать, и он проще для обработки базой данных. Запросом возвращаются все атрибуты, связанные с проблемой, даже если на момент написания запроса неизвестно, сколько существует атрибутов.

Чтобы использовать результат в данном формате, необходимо написать прикладной код для организации цикла по строкам набора результатов и установки свойств объекта в приложении. См. в качестве примера следующий PHP-код:

Файл примера: *EAV/soln/post-process.php*

```
<?php
$objects = array();

$stmt = $pdo->query(
        "SELECT issue_id, attr_name, attr_value
        FROM IssueAttributes
        WHERE issue_id = 1234");
while ($row = $stmt->fetch()) {
   $id = $row['issue_id'];
   $field = $row['attr_name'];
   $value = $row['attr_value'];
   if (!array_key_exists($id, $objects)) {
   $objects[$id] = new stdClass();
   }
   $objects[$id]->$field = $value;
}
```

Данный пример может выглядеть излишне трудоемким, но это ни что иное, как следствие применения системы-внутри-системы, подобной EAV. Язык SQL уже предлагает способ идентификации отдельных атрибутов — в отдельных столбцах. Используя EAV, вы вводите в SQL новый способ идентификации атрибутов, так что нет ничего удивительного в том, что SQL поддерживает эту функцию неуклюже и неэффективно.

ВНИМАНИЕ!

Используйте метаданные для метаданных.

Несомненно, некоторые люди идут обоими путями.

Слова Страшилы из «Волшебника страны Оз»

ГЛАВА 7. ПОЛИМОРФНЫЕ АССОЦИАЦИИ

Предоставьте пользователям возможность оставлять *комментарии* по ошибкам. Какая-нибудь заданная ошибка может сопровождаться несколькими комментариями, но любой заданный комментарий должен относиться к одной ошибке. Таким образом, между ошибками (Bugs) и комментариями (Comments) существует отношение «один-множество». Схема взаимосвязей блоков для данного типа простой ассоциации показана на рис. 7.1, а следующий код SQL показывает, как создать данную таблицу:

Файл примера: *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)
);
```

Тем не менее возможно наличие двух таблиц, которые можно комментировать. Bugs и FeatureRequests являются сходными объектами, хотя они могут храниться как отдельные таблицы (см. раздел 6.5, «Наследование конкретной таблицы»). Скорее всего, вы захотите хранить комментарии (Comments) в одной таблице независимо от того, принадлежат они какому-то одному типу проблем (ошибкам или функциям) или нет. Однако, как известно, объявить внешний ключ, ссылающийся на несколько родительских таблиц, нельзя. Следующее объявление лишено смысла:

Файл примера: *Polymorphic/intro/nonsense.sql*

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

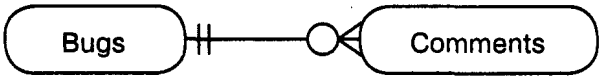


Рис. 7.1. Простая ассоциация

Для запроса нескольких таблиц разработчики пытаются писать недопустимый SQL-код, подобный показанному ниже:

Файл примера: *Polymorphic/intro/nonsense.sql*

```
SELECT c.*, i.summary, i.status
FROM Comments AS c
JOIN c.issue_type AS i USING (issue_id);
```

Но в SQL нельзя выполнить построчное объединение с другой таблицей. Согласно требованиям синтаксиса SQL все таблицы должны именоваться точно на момент передачи запроса. Таблицы не могут изменяться во время запроса.

Что же здесь не так, и как разрешить проблему?

7.1. ЦЕЛЬ: ССЫЛКА НА НЕСКОЛЬКО РОДИТЕЛЬСКИХ ОБЪЕКТОВ

Когда Дороти в книге «Волшебник страны Оз» спрашивает, какую дорогу на развилке следует выбрать, чтобы добраться до Изумрудного города, Страшила дает ей неопределенные указания. Вроде бы все должно быть однозначно, тем не менее Страшила просто сбивает девочку с толку, когда пытается дать ей одновременно два ответа.

Данный вид ассоциации иллюстрируется схемой взаимосвязей объектов на рис. 7.2. Внешний ключ в дочерней таблице «разветвляется», так что строка в таблице `Comments` соответствует либо строке в таблице `Bugs`, либо строке в таблице `FeatureRequests`. Искривленная дуга на схеме показывает взаимоисключающий выбор: любой заданный комментарий может ссылаться или на запрос одной ошибки, или на запрос одной функции.

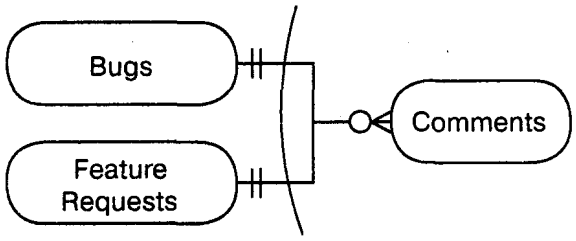


Рис. 7.2. Полиморфная ассоциация

7.2. АНТИПАТТЕРН: ИСПОЛЬЗОВАНИЕ ВНЕШНЕГО КЛЮЧА ДВОЙНОГО НАЗНАЧЕНИЯ

Решение для этих случаев приобрело такую популярность, что ему было дано имя — **Полиморфные ассоциации**. Данное решение иногда называют также смешанной ассоциацией, так как оно может ссылаться на несколько таблиц.

Определение полиморфной ассоциации

Чтобы заставить работать полиморфные ассоциации, надо добавить дополнительный столбец строковых значений рядом с внешним ключом по `issue_id`. Дополнительный столбец содержит имя родительской таблицы, на которую ссылается текущая строка. В данном примере новый столбец называется `issue_type`, и он содержит либо *Bugs*, либо *FeatureRequests*, соответствующие именам двух родительских таблиц в этой ассоциации.

Файл примера: *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)
);
```

Можно сразу заметить одну особенность: отсутствует объявление внешнего ключа для `issue_id`. В действительности, так как внешним ключом должна задаваться ровно одна таблица, использование полиморфной ассоциации означает, что данную ассоциацию невозможно объявить в метаданных. В результате отсутствует принудительный ввод в действие целостности данных для обеспечения соответствия значения в `Comments.issue_id` значению в родительской таблице.

Подобным образом отсутствие метаданных гарантирует, что строковое значение в `Comments.issue_type` соответствует таблице, которая существует в этой базе данных.

СМЕШИВАНИЕ ДАННЫХ С МЕТАДАННЫМИ

Возможно, вы заметили схожесть между антипаттерном **Полиморфные ассоциации** и антипаттерном **EAV (Объект-Атрибут-Значение)**, описанным в предыдущей главе. В обоих антипаттернах имя объекта метаданных хранится как строковое значение. В EAV имя столбца атрибутов хранится как строковое значение в столбце `attr_name`. В **Полиморфных ассоциациях** имена родительских таблиц хранятся в столбце `issue_type`. Иногда это называется **смешением данных с метаданными**. Такая концепция описывается в несколько иной форме в главе 8.

Запрос полиморфной ассоциации

Значение `issue_id` в таблице `Comments` может встречаться в столбце первичных ключей обеих родительских таблиц, `Bugs` и `FeatureRequests`. Значение может встретиться в одной родительской таблице, но отсутствовать в другой. Поэтому крайне важно правильно использовать `issue_type` при объединении дочерней таблицы с родительской. Не следует сопоставлять значение `issue_id` с таблицей `FeatureRequests`, если оно было предназначено для сопоставления с таблицей `Bugs`.

Например, с помощью следующего программного кода извлекают комментарии для какой-нибудь заданной ошибки по значению 1234 первичного ключа:

Файл примера: *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 такой запрос работает, если ошибки хранятся в одной таблице `Bugs`, проблема возникает, когда `Comments` связывается с обеими таблицами, `Bugs` и `FeatureRequests`. В SQL необходимо указать все таблицы в объединении. Невозможно объединить `Comments` с двумя отдельными таблицами, переключаясь между ними строка за строкой, в зависимости от значения в столбце `Comments.issue_type`.

Чтобы извлечь какой-то конкретный комментарий или по ошибке, или по функции, требуется выполнить запрос с помощью внешнего объединения с обеими родительскими таблицами. Только одна из родительских таблиц будет удовлетворять своему объединению, поскольку часть условия объединения основано на значении в столбце `Comments.issue_type`. Использование внешнего объединения означает, что поля из таблицы, у которых нет соответствия, содержат значение `Null` в наборе результатов.

Файл примера: *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 = 'Fea-
      tureRequests');
```

Результат может выглядеть подобно тому, как показано ниже:

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

Необъектноориентированный пример

В примере с Bugs и FeatureRequests подразумевается, что этими двумя родительскими таблицами моделируются связанные подтипы. **Полиморфные ассоциации** могут также применяться, когда родительские таблицы полностью не связаны друг с другом. Например, в базе данных электронной торговли обе таблицы, Users и Orders, могут быть связаны с Addresses, как показано на рис. 7.3.

Файл примера: *Polymorphic/anti/addresses.sql*

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

В данном случае таблица Addresses содержит полиморфный столбец, который именуется или как *Users*, или как *Orders* и служит в качестве родительской таблицы для заданного адреса. Обратите внимание, что необходимо выбрать одну или другую таблицу. Нельзя связать заданный адрес как с пользователем, так и с заказом, даже когда заказ размещен этим пользователем с тем, чтобы отправить товары самому себе.

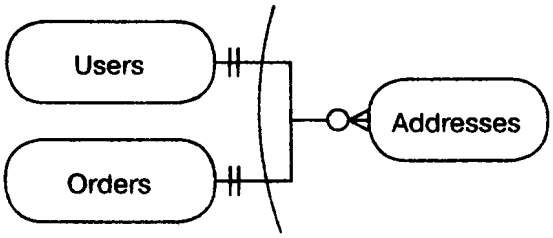


Рис. 7.3. Полиморфные ассоциации для адресов

К тому же если у пользователя есть адрес отгрузки, а также адрес выставления счета, необходимо придумать какой-нибудь способ, чтобы различать их в таблице `Addresses`. Аналогично любым другим родительским объектам требуется отмечать особое использование адресов в таблице `Addresses`. Эти заметки могут распространяться как сорняки.

Файл примера: *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
);
```

7.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

На использование антипаттерна **Полиморфные ассоциации** указывают следующие заявления.

- «Данная схема тегирования позволяет связать тег (или другой атрибут) с *любым* другим ресурсом в базе данных».

Здесь, как и в EAV (объект-атрибут-значение), к любым заявлениям о неограниченной гибкости следует относиться с недоверием.

- «В структуре нашей базы данных невозможно объявить внешние ключи».

Это еще один красный флаг. Внешние ключи являются фундаментальной характеристикой реляционных баз данных, и структура, которая не может работать с надлежащей целостностью на уровне ссылок, сталкивается с многочисленными проблемами.

- «Для чего нужен столбец `entity_type`? Ах да, он сообщает, на что указывает этот другой столбец».

Любой внешний ключ должен ссылаться на одну и ту же таблицу во всех строках.

Интегрированная среда Ruby on Rails поддерживает полиморфные ассоциации путем объявления классов активных записей (Active Record) с атрибутом `polymorphic`. Например, можно связать `Comments` с `Bugs` и `FeatureRequests` следующим образом:

Файл примера: *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
```

Интегрированной средой Hibernate для Java поддерживается антипаттерн **Полиморфные ассоциации**, использующий различные объявления схем[1].

7.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Следует избегать применения антипаттерна **Полиморфные ассоциации.** Используйте ограничения, такие как внешние ключи, чтобы обеспечить целостность на уровне ссылок. Антипаттерн **Полиморфные ассоциации** часто чрезмерно полагается на код приложения вместо метаданных.

Возможно, вы посчитаете, что без данного антипаттернане обойтись, если используется объектноориентированная интегрированная среда программирования, такая как Hibernate. Подобная интегрированная среда помогает уменьшить риски, характерные для полиморфных ассоциаций, путем инкапсуляции логики приложения с целью поддержания целостности на уровне ссылок. Если выбрана зрелая интегрированная среда, пользующаяся хо-

[1] См. веб-сайт **www.hibernate.org/hib_docs/reference/en/html/inheritance.html**. «Hibernate» — знак обслуживания, принадлежащий корпорации Red Hat.

рошей репутацией, тогда до определенной степени можно быть уверенным, что ее разработчики написали код для реализации ассоциаций без ошибок. Тем не менее если вы реализуете полиморфные ассоциации с нуля, без помощи интегрированной среды разработки, скорее всего вы заново будете изобретать велосипед.

7.5. РЕШЕНИЕ: УПРОЩЕНИЕ ОТНОШЕНИЙ

Лучше перепроектировать базу таким образом, чтобы избежать слабых сторон полиморфных ассоциаций, но по-прежнему поддерживать требуемую модель данных. В следующих разделах описываются некоторые решения, которые приспосабливают требуемую взаимосвязь данных и при этом улучшают способ использования метаданных для принудительного ввода в действие целостности данных.

Реверс ссылки

Одно решение для данного антипаттерна является простым, если взглянуть на природу проблемы: *полиморфные ассоциации являются обратными.*

Создание таблиц пересечения

Внешний ключ в дочерней таблице `Comments` не может ссылаться на несколько родительских таблиц, так что вместо этого используйте несколько внешних ключей для ссылки на таблицу `Comments`. Для каждой родительской таблицы создайте отдельную таблицу пересечений и в каждой таблице пересечений определите внешний ключ для `Comments`, а также внешний ключ для соответствующей родительской таблицы. Данная структура иллюстрируется схемой взаимосвязи объектов на рис. 7.4.

Файл примера: *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)
);
```

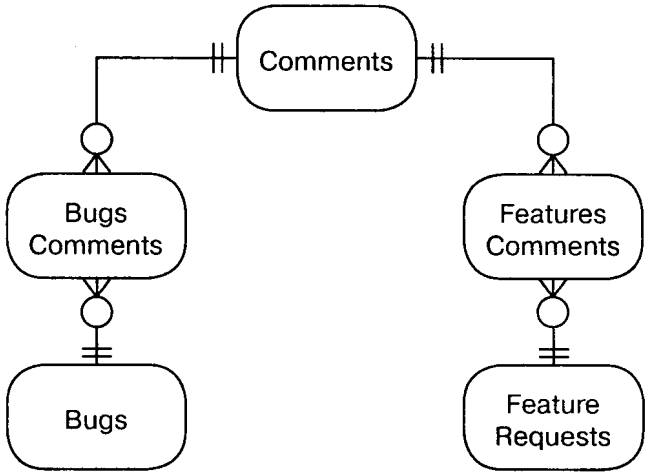


Рис. 7.4. Изменение направления полиморфной ассоциации на обратное

Данное решение устраняет потребность в столбце `Comments.issue_type`. Теперь метаданные могут принудительно устанавливать целостность данных, вместо того чтобы полагаться на код приложения для управления ассоциациями без ошибок.

Установка светофоров

Недостаток данного метода заключается в том, что оно разрешает ассоциации, которые могут быть нежелательными. Обычно таблицы пересечений моделируют ассоциации «множество-множество», так что это позволяет связать заданный комментарий с несколькими ошибками или несколькими запросами функций. Однако может потребоваться, чтобы каждый комментарий относился только к одной ошибке или одному запросу функции. Можно ввести в действие по крайней мере часть данного правила, объявив ограничение `UNIQUE` по столбцу `comment_id` для каждой таблицы пересечений.

Файл примера: *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)
);
```

Это гарантирует, что на заданный комментарий может быть только одна ссылка в таблице пересечений, которая, естественно, предотвращает связь с несколькими ошибками или несколькими запросами функций. Однако метаданными не запрещается однократная ссылка на заданный комментарий в обеих таблицах пересечений, связывающих комментарий как с ошибкой, так и с запросом функции. Возможно, это не то, что требуется, но защита от такого алгоритма по-прежнему должна обеспечиваться кодом приложения.

Просмотр в обоих направлениях

Можно затребовать комментарии, выдаваемые по конкретной ошибке или по определенному запросу функции, просто используя таблицу пересечений.

Файл примера: *Polymorphic/soln/reverse-join.sql*

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

Можно затребовать соответствующую ошибку или соответствующий запрос функции на основе экземпляра комментария путем использования внешнего объединения с обеими таблицами пересечения. Требуется присвоить имена всем возможным родительским таблицам, и это не сложнее запроса, который необходим в антипаттерне **Полиморфные ассоциации**. К тому же при использовании таблицы пересечений может проявиться зависимость от целостности на уровне ссылок, в то время как в случае полиморфных ассоциаций такой зависимости нет.

Файл примера: *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;
```

Объединение путей

Иногда требуется сделать так, чтобы результат запроса по нескольким родительским таблицам выглядел так, как если бы родительские таблицы хранились в одной таблице (см. раздел 6.5 «Наследование одиночной таблицы»). Это можно осуществить двумя способами.

Сначала взгляните на следующий запрос, использующий оператор UNION:

Файл примера: *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;
```

Данный запрос должен гарантированно возвращать одиночную строку, если каждый комментарий связывается приложением исключительно с одной родительской таблицей. Так как результаты запроса могут объединяться с помощью оператора UNION только в том случае, если их столбцы одинаковы

по количеству и типу данных, необходимо предоставить заполнители NULL для столбцов, которые являются уникальными для каждой родительской таблицы. Столбцы следует перечислять в одном и том же порядке в обоих запросах, задействованных в операторе UNION.

В качестве альтернативы взгляните на следующий запрос, использующий SQL-функцию COALESCE(). Данной функцией возвращается ее первый аргумент, отличный от NULL. Поскольку в запросе используется внешнее объединение, комментарий, принадлежащий запросу функции, у которого нет соответствующей строки в Bugs, все поля в b.* возвращаются как Null. Аналогичным образом, все поля в f.* будут Null, если комментарий относится к ошибке, а не к запросу функции. Перечислите поля, характерные для одной или другой родительской таблицы в простом виде. Если они не относятся к соответствующей родительской таблице, они возвращаются как Null.

Файл примера: *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 достаточно сложные, поэтому они хорошо подходят для просмотра базы данных, а в приложении их можно использовать в упрощенном виде.

Создание общей супертаблицы

В объектноориентированном полиморфизме ссылки на два подтипа могут осуществляться аналогичным образом, поскольку подтипами неявно совместно используется общий супертип. В SQL антипаттерн **Полиморфные ассоциации** не учитывает такой важный объект, как общий супертип. Данный недостаток можно исправить путем создания базовой таблицы, которую расширяют все родительские таблицы (см. раздел 6.5, «Наследование таблицы классов»). Добавьте внешний ключ в дочернюю таблицу `Comments`, чтобы ссылаться на базовую таблицу. Столбец `issue_type` не требуется. Данное решение иллюстрируется схемой взаимосвязей объектов на рис. 7.5.

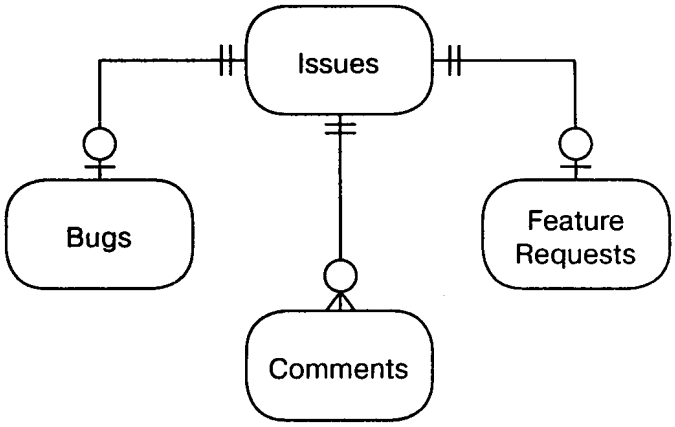


Рис. 7.5. Связь комментариев с таблицей `Base Issues`

Файл примера: *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 на то, что первичные ключи таблиц `Bugs` и `FeatureRequests` являются также внешними ключами. Они ссылаются на значение свернутого ключа, сгенерированного в таблице `Issues`, вместо создания нового значения для самих себя.

Если задан конкретный комментарий, ошибку или запрос функции, на которые дается ссылка, можно извлечь с помощью относительно простого запроса. Здесь не требуется включать в запрос таблицу `Issues`, если только в ней не определены столбцы атрибутов. К тому же, так как значения первичного ключа таблицы `Bugs` и ее таблицы-предка `Issues` являются одинаковыми, можно объединить таблицу `Bugs` непосредственно с `Comments`. Две таблицы можно объединить, даже если не существует ограничения внешнего ключа, связывающего их напрямую, до тех пор, пока используются столбцы, которые представляют сравнимые сведения в базе данных.

Файл примера: *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;
```

Если задана конкретная ошибка, ее комментарии можно извлечь так же просто.

Файл примера: *Polymorphic/soln/super-join.sql*

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

Следует иметь в виду, что когда используется таблица-предок, подобная `Issues`, можно полагаться на принудительную установку целостности данных в базе, выполняемую посредством внешних ключей.

ВНИМАНИЕ!

В каждой взаимосвязи таблиц существует одна таблица, выполняющая ссылку, и одна таблица, на которую ссылаются.

Возвышенное и смешное часто так тесно связаны, что порой их сложно разделить.

Томас Пейн

ГЛАВА 8. МНОГОСТОЛБЧАТЫЕ АТРИБУТЫ

Не могу сосчитать, сколько раз мне приходилось создавать таблицу для хранения контактных данных людей. Этот тип таблицы всегда содержит банальные столбцы, такие как фамилия, форма обращения, адрес и, возможно, название компании.

Номера телефонов — тут чуть сложнее. Люди используют несколько номеров: чаще всего встречаются номер домашнего телефона, номер рабочего телефона, номер факса и номер мобильного телефона. В таблице контактных данных эти сведения проще хранить в четырех столбцах.

А что скажете о дополнительных телефонных номерах? Персонального помощника, второго мобильного телефона, филиала. Кстати, могут встречаться и другие непредвиденные категории. Можно было бы создать дополнительные столбцы для менее распространенных случаев, но это выглядело бы неуклюже, поскольку в формы записей данных добавляются редко используемые поля. Сколько же столбцов достаточно для всего этого?

8.1. ЦЕЛЬ: ХРАНЕНИЕ МНОГОЗНАЧНЫХ АТРИБУТОВ

Здесь мы встречаемся с тем же, что и в главе 2. Кажется, что атрибут принадлежит одной таблице, но у атрибута несколько значений. Выше было показано, что объединение нескольких значений в строковое значение, разделяемое запятыми, затрудняет проверку достоверности значений, чтение и изменение отдельных значений, а также вычисление составных выражений, например вычисление количества отдельных значений.

Для иллюстрации данного антипаттерна мы рассмотрим новый пример. База данных ошибок должна допускать использование *тегов*, чтобы можно было классифицировать ошибки. Некоторые ошибки могут классифицироваться по программным подсистемам, на которые они воздействуют, например, *печать*, *отчеты* или *электронная почта*. Другие ошибки могут классифицироваться по природе дефекта. Например, ошибке, вызывающей аварийный отказ, может быть присвоен тег *аварийный отказ*, в то время как отчету о замедлении работы может назначаться тег *производительность*, а неправильному выбору цветовой палитры в пользовательском интерфейсе можно назначить тег *косметика*.

Функция тегирования ошибок должна поддерживать несколько тегов, так как они не обязательно взаимноисключают друг друга. Дефект может воздействовать на несколько систем либо может влиять на производительность печати.

8.2. АНТИПАТТЕРН: СОЗДАНИЕ НЕСКОЛЬКИХ СТОЛБЦОВ

По-прежнему необходимо принимать во внимание наличие нескольких значений у атрибута, но известно, что новое решение должно позволять хранить только одно значение в каждом столбце. Возможно, покажется естественным создание в данной таблице нескольких столбцов, каждый из которых содержит один тег.

Файл примера: *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 заданной ошибке значения помещают в один из этих трех столбцов. Неиспользуемые столбцы остаются равными значению `Null`.

Файл примера: *Multi-Column/anti/update.sql*

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

bug_id	description	tag1	tag2	tag3
1234	Отказ при сохранении	Crash	NULL	NULL
3456	Увеличение производительности	Printing	performance	NULL
5678	Поддержка XML	NULL	NULL	NULL

Большинство задач, которые можно было бы легко выполнить с помощью обычного атрибута, теперь стали более сложными.

Поиск значений

При поиске ошибок с заданным тегом следует проводить поиск по всем трем столбцам, так как строковое значение тега может занимать любой из этих столбцов.

Например, чтобы извлечь ошибки, которые ссылаются на тег *performance* (производительность), используйте запрос, подобный приведенному ниже:

Файл примера: *Multi-Column/anti/search.sql*

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

Возможно, потребуется выполнить поиск ошибок, которые ссылаются на оба тега — *performance* (производительность) и *printing* (печать). Чтобы сделать это, воспользуйтесь запросом, подобным приведенному ниже. Не забывайте правильно расставлять круглые скобки, так как приоритет оператора OR ниже, чем оператора AND.

Файл примера: *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');
```

Как видите, синтаксическое выражение для поиска одиночного значения по нескольким столбцам оказывается чересчур длинным. Его можно сделать более компактным, применяя предикат IN в несколько нетрадиционной манере:

Файл примера: *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

Добавление и удаление значения из набора столбцов привносит дополнительные проблемы. Простое использование оператора UPDATE для измене-

ния одного из столбцов небезопасно, поскольку нельзя быть уверенным в том, что некий столбец не заполнен, если он вообще существуют. Чтобы увидеть это, возможно, потребуется извлечь строку в приложение.

Файл примера: *Multi-Column/anti/add-tag-two-step.sql*

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

Например, в данном случае результат показывает, что значение `tag2` — `Null`. Затем можно сформировать оператор `UPDATE`.

Файл примера: *Multi-Column/anti/add-tag-two-step.sql*

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

Есть вероятность, что в момент после запроса таблицы и до ее обновления вами, другой клиент выполняет те же самые шаги по чтению строки и ее обновлению. В зависимости от того, кто первый выполнит обновление, у вас или другого клиента есть риск получить ошибку конфликта обновления или перезапись внесенных изменений изменениями, сделанными другим лицом. Этого двухшагового запроса можно избежать, используя сложные SQL-выражения. В следующем операторе задействована функция `NULLIF()`, чтобы сделать каждый столбец равным `Null`, если он равен конкретному значению. Функцией `NULLIF()` возвращается значение `Null`, если два ее аргумента равны друг другу[1].

Файл примера: *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;
```

Следующий оператор добавляет новый тег *performance* (производительность) в первый столбец, который в текущий момент равен `Null`. Однако если ни один из трех столбцов не равен `Null`, тогда оператор не вносит никаких изменений в строку, и значение нового тега вообще не записывается. К тому же конструирование данного оператора утомительно. Обратите внимание, что строковое значение *performance* надо повторить шесть раз.

[1] `NULLIF()` — стандартная функция в SQL; она поддерживается всеми производителями баз данных за исключением Informix и Ingres.

Файл примера: *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

Возможно, нежелательно, чтобы одно и то же значение появлялось в нескольких столбцах, но при использовании антипаттерна **Многостолбчатые атрибуты** такая ситуация не может быть предотвращена базой данных. Другими словами, трудно запретить следующий оператор:

Файл примера: *Multi-Column/anti/insert-duplicate.sql*

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

Обработка увеличивающихся наборов значений

Другой недостаток данной структуры состоит в том, что трех столбцов может оказаться недостаточно. Для поддержания структуры с одним значением на столбец необходимо определить число столбцов, равное максимальному количеству тегов, которые могут быть у ошибки. Как можно предсказать, чему будет равно максимальное число на момент определения таблицы?

Один из тактических подходов состоит в прогнозировании среднего количества столбцов. Позднее, если возникает необходимость, можно расширить их количество путем добавления столбцов. Большинство баз данных позволяют реструктурировать существующие таблицы, так что по мере необходимости можно добавить `Bugs.tag4` или даже больше столбцов.

Файл примера: *Multi-Column/anti/alter-table.sql*

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

Однако данное изменение можно считать затратным по трем причинам.

- Реструктурирование таблицы базы, которая уже содержит данные, может потребовать блокирования всей таблицы, блокирования доступа для других одновременно работающих клиентов.
- Данный тип реструктурирования таблицы реализуется в некоторых базах данных путем определения новой таблицы для соответствия требуемой структуре, копирования данных из прежней таблицы и последующего удаления старой таблицы. Если интересующая вас таблица содержит много данных, этот перенос займет немало времени.
- Когда в набор многостолбчатого атрибута добавляется столбец, требуется пересмотреть все SQL-операторы в каждом приложении, которое использует данную таблицу, редактируя оператор для поддержки новых столбцов.

Файл примера: *Multi-Column/anti/search-four-columns.sql*

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

Это задача, требующая длительной, скрупулезной работы. Если пропустить какие-то запросы, нуждающиеся в редактировании, это может привести к ошибкам, которые будет трудно обнаружить.

8.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Если в пользовательском интерфейсе или в документации проекта описывается атрибут, которому допускается присваивание нескольких значений, но их количество ограничено фиксированным максимумом, это может указывать на использование антипаттерна **Многостолбчатые атрибуты**.

ШАБЛОНЫ СРЕДИ АНТИПАТТЕРНОВ

У антипаттернов **Блуждания без ориентиров** и **Многостолбчатые атрибуты** есть одна особенность. Они придуманы для достижения одной цели: хранения атрибута, который может содержать несколько значений.

В примерах для антипаттерна **Блуждания без ориентиров** видно, как этот антипаттерн соотносится с взаимосвязями «множество-множество». В этой главе показана более простая взаимосвязь «один-множество». Следует иметь в виду, что для взаимосвязей обоих типов иногда используются оба антипаттерна.

Общеизвестно, что для некоторых атрибутов существует предел по количеству целевых выборов, но более распространена ситуация, когда такой предел отсутствует. Если предел кажется произвольным или неоправданным, причиной этого может быть и данный антипаттерн.

Еще одной подсказкой, указывающей на возможное применение антипаттерна, может быть любое из следующих заявлений:

- «Какое наибольшее число тегов требуется поддерживать?»

Необходимо принять решение, сколько столбцов должно быть определено в таблице для многозначного атрибута, такого как `tag`.

- «Как в SQL одновременно выполнить поиск по нескольким столбцам?»

Если выполняется поиск некоторого заданного значения по нескольким столбцам, то это может указывать на реальную необходимость хранения нескольких столбцов в виде единого логического атрибута.

8.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

В некоторых случаях атрибут допускает фиксированное число вариантов выбора, при этом может быть важно расположение или порядок этих вариантов. Например, какая-нибудь заданная ошибка может быть связана с учетными записями нескольких пользователей, однако природа каждой связи является уникальной. Одной из связей является пользователь, сообщивший об ошибке, другой — программист, которому поручено исправление ошибки, еще одной — инженер службы технического контроля, которому вменена в обязанность проверка исправления. Даже если значения в каждом из этих столбцов являются совместимыми, их важность и применение делают их в действительности логически разными атрибутами.

Было бы правомерно определить три обычных столбца в таблице `Bugs` для хранения каждого из этих трех атрибутов. Недостатки, описываемые в данной главе, не являются настолько важными, поскольку, скорее всего, столбцы будут использоваться раздельно. Возможно, иногда по-прежнему требуется выполнять запрос по всем трем столбцам, например, чтобы сообщить какие-либо сведения всем работающим с заданной ошибкой. Но можно принять эту сложность для нескольких случаев в обмен на большую простоту в большинстве других применений.

Еще один способ разработки структуры в рамках этого решения состоит в создании зависимой таблицы для нескольких связей из таблицы `Bugs`, таблицы `Accounts` и ввод в эту новую таблицу дополнительного столбца с целью пометки роли каждой учетной записи по отношению к рассматриваемой ошибке. Однако данная структура может привести к некоторым проблемам, описанным в главе 6.

8.5. РЕШЕНИЕ: СОЗДАНИЕ ЗАВИСИМОЙ ТАБЛИЦЫ

Как показано в главе 2, оптимальное решение состоит в создании зависимой таблицы с одним столбцом для многозначного атрибута. Храните несколько значений в нескольких строках вместо использования для этого нескольких столбцов. К тому же определите в зависимой таблице внешний ключ, чтобы связать значения с их родительской строкой в таблице `Bugs`.

Файл примера: *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, 'perfor-
   mance');
```

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

Файл примера: *Multi-Column/soln/search.sql*

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

Легко прочитать даже более сложные виды поиска, например ошибки, связанной с двумя конкретными тегами.

Файл примера: *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';
```

Добавить или удалить связь можно гораздо легче, чем с помощью антипатерна **Многостолбчатые атрибуты**. Просто вставьте или удалите строку в зависимой таблице. Нет необходимости просматривать несколько столбцов, чтобы увидеть, куда можно добавить значение.

Файл примера: *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 PIRMARY KEY обеспечивает запрет дублирования. Заданный тег может быть применен к заданной ошибке только один раз. Если попытаться вставить дубликат, языком SQL возвращается ошибка дублированного ключа.

Число тегов, приходящихся на ошибку, не ограничивается тремя, как было, когда существовало всего три столбца tagN в таблице Bugs. Теперь можно применить столько тегов в расчете на каждую ошибку, сколько необходимо.

ВНИМАНИЕ!

Храните все значения, обладающие одним и тем же смыслом, в одном столбце.

Я хочу выбросить эти создания с судна. Не важно, если для этого потребуются все до последнего человека. Я хочу убрать их с корабля.

Джеймс Т. Кирк

ГЛАВА 9. ТРИББЛЫ МЕТАДАННЫХ

На протяжении многих лет моя жена профессионально программировала на Oracle PL/SQL и Java. Она рассказывала случай, который показывает, как структура базы данных, предназначавшаяся для упрощения работы, вместо этого стала причиной дополнительной работы.

Таблица `Customers`, использовавшаяся отделом продаж в ее компании, содержала сведения, такие как контактные данные клиентов, вид их бизнеса и объем доходов, полученных от клиентов:

Файл примера: *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)
);
```

Но отделу продаж требовалось разбить доход по годам, чтобы можно было отслеживать клиентов, проявляющих активность в текущее время. Было решено добавить ряд новых столбцов, название каждого столбца показывало год, данные за который содержались в этом столбце:

Файл примера: *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 были введены неполные данные, только для клиентов, которых, как считали в компании, будет интересно отслеживать. В большинстве строк было оставлено значение `Null` в этих столбцах дохода. Программисты начали интересоваться, можно ли хранить другие сведения в этих, большей частью неиспользуемых столбцах.

Каждый год требовалось добавлять один дополнительный столбец. Администратор базы данных отвечал за управление табличными областями Oracle. Таким образом, каждый год требовалось проводить серию деловых встреч, планировать перенос данных для реструктуризации табличной области и добавлять новый столбец. В конечном счете они потратили впустую немало времени и денег.

9.1. ЦЕЛЬ: ПОДДЕРЖКА МАСШТАБИРУЕМОСТИ

По мере увеличения объема сведений ухудшается производительность выполнения любого запроса в базе данных. Даже если результаты с несколькими тысячами строк возвращаются запросом оперативно, таблицы, естественно, накапливают данные до момента, когда производительность того же самого запроса, возможно, будет неприемлемой. Рациональное применение индексов помогает, но тем не менее таблицы разрастаются, и этот рост влияет на скорость запросов в этих таблицах.

Цель заключается в систематизации базы данных для повышения производительности запросов и поддержки постоянно увеличивающихся таблиц.

9.2. АНТИПАТТЕРН: КЛОНИРОВАНИЕ ТАБЛИЦ ИЛИ СТОЛБЦОВ

В телевизионном сериале «Звездный путь»[1] «трибблы» — это маленькие мохнатые животные, которых держат в качестве домашних любимцев. Вначале трибблы выглядят очень привлекательно, но вскоре оказывается, что они склонны к бесконтрольному размножению, и управление перенаселенностью трибблов становится серьезной проблемой.

Где их размещать? Кто несет ответственность за них? Сколько времени займет поимка каждого триббла? В конце концов капитан Кирк обнаруживает, что его корабль и команда не могут функционировать, и он вынужден отдать первоочередную команду своим людям — избавиться от трибблов.

Из опыта известно, что при прочих равных условиях запрос таблицы, содержащей всего несколько строк, выполняется быстрее, чем запрос таблицы, содержащей много строк. Это ведет к общему заблуждению, что необходимо сделать так, чтобы каждая таблица содержала мало строк, не важно, что для этого надо сделать. Подобные аргументы ведут к двум формам антипаттерна:

- Разбивка одиночной длинной таблицы на несколько таблиц меньшего размера, с использованием названий таблиц на основе индивидуальных значений данных в одном из табличных атрибутов.

[1] «Star Trek» и соответствующие знаки являются товарными знаками корпорации CBS Studios.

- Разбивка одиночного столбца на несколько столбцов, используя имена столбцов на основе индивидуальных значений в другом атрибуте.

Однако невозможно что-нибудь получить просто так: чтобы достичь цели в виде всего нескольких строк в каждой таблице, необходимо либо создать таблицы, содержащие слишком много столбцов, либо создать большее число таблиц. В обоих случаях можно обнаружить, что количество таблиц или столбцов продолжает расти, поскольку новые значения данных могут вынудить создавать новые объекты схемы.

СМЕШИВАНИЕ МЕТАДАННЫХ С ДАННЫМИ

Обратите внимание, что путем присоединения года к базовому имени таблицы значение данных объединяется с идентификатором метаданных.
Это операция, обратная **смешиванию данных с метаданными**, которая рассматривалась ранее в антипаттернах **EAV (Объект-Атрибут-Значение)** и **Полиморфные ассоциации**. В тех случаях идентификаторы метаданных (имя столбца и имя таблицы) хранились в виде строчных данных.
В **Многостолбчатых атрибутах** и **Трибблах метаданных** значение данных вносится в имя столбца или имя таблицы. Если используется любой из этих антипаттернов, число возникающих проблем будет превышать количество решаемых вопросов.

Порождение таблиц

Чтобы разбить данные на отдельные таблицы, потребуется некоторая методика определения принадлежности строк таблицам. Например, можно было бы разбить данные по годам в столбце `date_reported`:

Файл примера: *Metadata-Tribbles/anti/create-tables.sql*

```
CREATE TABLE Bugs_2008 ( . . . );
CREATE TABLE Bugs_2009 ( . . . );
CREATE TABLE Bugs_2010 ( . . . );
```

При вставке строк в базу данных вы отвечаете за использование правильной таблицы в зависимости от вставляемых значений:

Файл примера: *Metadata-Tribbles/anti/insert.sql*

```
INSERT INTO Bugs_2010 (..., date_reported, ...) VALUES (...,
'2010-06-01', ...);
```

Быстрый переход вперед к 1 января следующего года. Приложение начинает получать ошибку из всех новых отчетов об ошибках, так как вы забыли создать таблицу `Bugs_2011`.

Файл примера: *Metadata-Tribbles/anti/insert.sql*

```
INSERT INTO Bugs_2011 (..., date_reported, ...) VALUES (...,
'2011-02-20', ...);
```

Это означает, что ввод значения новых *данных* может привести к необходимости создания нового объекта *метаданных*. Данная ситуация обычно не является взаимосвязью между данными и метаданными в SQL.

Управление целостностью данных

Предположим, что ваш начальник пытается посчитать ошибки, о которых сообщалось в течение года, однако его числа не суммируются. После анализа вы обнаруживаете, что некоторые ошибки 2010 года были введены по ошибке в таблицу `Bugs_2009`. Следующий запрос должен всегда возвращать пустой результат, и если он этого не делает, значит, существует проблема:

Файл примера: *Metadata-Tribbles/anti/data-integrity.sql*

```
SELECT * FROM Bugs_2009
WHERE date_reported NOT BETWEEN '2009-01-01' AND '2009-12-31';
```

Отсутствует способ автоматически ограничивать данные в зависимости от имени их таблицы, но в каждой из таблиц можно объявить ограничение `CHECK`:

Файл примера: *Metadata-Tribbles/anti/check-constraint.sql*

```
CREATE TABLE Bugs_2009 (
   -- другие столбцы
   date_reported DATE CHECK (EXTRACT(YEAR FROM date_reported)
   = 2009)
);

CREATE TABLE Bugs_2010 (
   -- другие столбцы
   date_reported DATE CHECK (EXTRACT(YEAR FROM date_reported)
   = 2010)
);
```

При создании `Bugs_2011` не забудьте изменить значение в ограничении `CHECK`. В случае внесения ошибки можно будет создать таблицу, отклоняющую строки, которые, как предполагалось, ею должны приниматься.

Синхронизация данных

В один прекрасный день аналитик службы поддержки клиентов просит изменить дату отчета об ошибках. В базе данных она, согласно сообщениям, представлена как 2010-01-03, но клиент, сообщивший о ней, в действительности прислал отчет по факсу неделей раньше — 2009-12-27. Дату можно было бы изменить с помощью простого оператора UPDATE:

Файл примера: *Metadata-Tribbles/anti/anomaly.sql*

```
UPDATE Bugs_2010
SET date_reported = '2009-12-27'
WHERE bug_id = 1234;
```

Однако данное исправление делает строку недопустимой записью в таблице Bugs_2010. Потребуется удалить строку из одной таблицы и вставить ее в другую таблицу при возникновении редкого случая, когда простой оператор UPDATE вызовет данную аномалию.

Файл примера: *Metadata-Tribbles/anti/synchronize.sql*

```
INSERT INTO Bugs_2009 (bug_id, date_reported, ...)
   SELECT bug_id, date_reported, ...
   FROM Bugs_2010
   WHERE bug_id = 1234;

DELETE FROM Bugs_2010 WHERE bug_id = 1234;
```

Обеспечение уникальности

Следует убедиться, что значения первичного ключа являются уникальными по всем разделенным таблицам. Если требуется переместить строку из одной таблицы в другую, необходима некоторая гарантия того, что значение первичного ключа не конфликтует с другой строкой.

Если используется база данных, поддерживающая объекты последовательности, можно воспользоваться одной последовательностью, чтобы сгенерировать значения для всех разделенных таблиц. Для баз данных, поддерживающих только потабличную уникальность идентификаторов, это может быть более затруднительным делом. Необходимо определить одну дополнительную таблицу, предназначенную исключительно для генерирования значений первичных ключей:

Файл примера: *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_2010 (bug_id, . . .)
   VALUES (LAST_INSERT_ID(), . . .);
```

Запрос по таблицам

Вашему начальнику неизбежно потребуется запрос, который ссылается на несколько таблиц. Например, он может запросить количество всех открытых ошибок независимо от года, когда они были созданы. Можно восстановить полный набор ошибок, используя оператор UNION для всех разделенных таблиц, и запросить это как производную таблицу:

Файл примера: *Metadata-Tribbles/anti/union.sql*

```
SELECT b.status, COUNT(*) AS count_per_status FROM (
   SELECT * FROM Bugs_2008
      UNION
   SELECT * FROM Bugs_2009
      UNION
   SELECT * FROM Bugs_2010 ) AS b
GROUP BY b.status;
```

По мере того как с течением времени создают дополнительные таблицы, например Bugs_2011, необходимо поддерживать код приложения обновленным для ссылки на вновь создаваемые таблицы.

Синхронизация метаданных

Ваш начальник просит вас добавить столбец для отслеживания времени работы, необходимой для устранения каждой ошибки.

Файл примера: *Metadata-Tribbles/anti/alter-table.sql*

```
ALTER TABLE Bugs_2010 ADD COLUMN hours NUMERIC(9,2);
```

Если разделить таблицу, тогда новый столбец будет относиться только к одной изменяемой таблице. Никакая из других таблиц не содержит нового столбца.

Если использовать запрос UNION по разделенным таблицам, как в предыдущем разделе, возникает новая проблема: можно объединить таблицы, применяя оператор UNION, когда таблицы содержат одни и те же столбцы. Если столбцы отличаются, тогда надо именовать только столбцы, которые являются общими для всех таблиц, без использования подстановочного знака «*».

Управление целостностью на уровне ссылок

Если зависимая таблица, такая как Comments, ссылается на Bugs, зависимой таблицей не может быть объявлен внешний ключ. Внешний ключ должен задавать одиночную таблицу, но в данном случае родительская таблица разбита на несколько таблиц.

Файл примера: *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)
);
```

Разделенная таблица также может быть сопряжена с проблемами, являясь зависимой, а не родительской. Например, Bugs.reported_by ссылается на таблицу Accounts. Если надо запросить все ошибки, о которых сообщалось заданным лицом независимо от года, потребуется запрос, подобный следующему:

Файл примера: *Metadata-Tribbles/anti/join-union.sql*

```
SELECT * FROM Accounts a
JOIN (
        SELECT * FROM Bugs_2008
        UNION ALL
        SELECT * FROM Bugs_2009
        UNION ALL
        SELECT * FROM Bugs_2010
   ) t ON (a.account_id = t.reported_by)
```

Идентификация столбцов триббло в метаданных

Столбцы также могут быть триббламн метаданных. Можно создать таблицу, содержащую столбцы, которые вынуждены размножаться по своей природе, как видно из истории, описанной в начале этой главы.

Еще одним примером для нашей базы данных ошибок является таблица, в которую записываются сводные данные для метрики проекта, где в отдельных столбцах хранятся промежуточные суммы. Например, в следующей таблице по прошествии некоторого времени потребуется добавить столбец `bugs_fixed_2011`:

Файл примера: *Metadata-Tribbles/anti/multi-column.sql*

```
CREATE TABLE ProjectHistory (
   bugs_fixed_2008 INT,
   bugs_fixed_2009 INT,
   bugs_fixed_2010 INT
);
```

9.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Индикаторами появления антипаттерна **Триббльі метаданных** в базе данных могут служить следующие фразы.

- «Тогда необходимо создать таблицу (или столбец) по...»

Когда базу данных описывают с помощью фраз, использующих *«по...»* в указанной фразе, выполняют разделение таблиц по отдельным значениям в одном из столбцов.

- «Какое максимальное число таблиц (или столбцов) поддерживается базой данных?»

Базы данных большинства производителей позволяют обрабатывать гораздо больше таблиц и столбцов, чем может потребоваться, когда используется осмысленная структура базы данных. Если возникает подозрение, что максимум может быть превышен, это верный признак необходимости перепроектирования структуры.

- «Нами обнаружено, почему приложению не удалось добавить новые данные этим утром: мы забыли создать новую таблицу для нового года».

Это распространенное последствие **Триббльов метаданных**. Когда для новых данных требуются новые объекты базы данных, необходимо заранее определить эти объекты, в противном случае существует риск непредвиденных отказов.

- «Как выполнить запрос, чтобы найти несколько таблиц одновременно? Все таблицы содержат одни и те же столбцы».

Если требуется выполнить поиск по многим таблицам с идентичной структурой, их следует хранить вместе в одной таблице с одним дополнительным столбцом атрибутов, чтобы различать строки.

- «Как передать параметр для имени таблицы? Мне требуется выполнить запрос имени таблицы, к которому динамически присоединяется номер года».

Этого не придется делать, если данные находились в одной таблице.

9.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Разделяемые вручную таблицы находят полезное применение в *архивировании* — удалении из повседневного использования данных за прошлый период. Часто необходимость выполнять запросы по данным за прошлый период времени существенно уменьшается, после того как данные теряют свою «свежесть».

Если не требуется запрашивать вместе текущие данные и данные за прошлый период времени, целесообразно скопировать более старые данные в другое место и удалить их из активных таблиц. Архивирование позволяет хранить данные в совместимой табличной структуре для выполнения время от времени анализа и дает возможность ускорить запросы по текущим данным.

РАЗДЕЛЕНИЕ БАЗ ДАННЫХ НА ВЕБ-САЙТЕ WORDPRESS.COM

На конференции по MySQL и Expo 2009 я обедал с Барри Абрахамсоном (Barry Abrahamson), разработчиком структуры базы данных на веб-сайте WordPress.com, популярной службе хостинга программного обеспечения для ведения блогов.

Барри рассказал, когда он начинал размещать блоги, он размещал вместе всех своих клиентов в одной базе данных. Как-никак, содержимое одного сайта блогов в действительности не было таким уж большим. Важным аргументом в пользу единой базы данных была ее оптимальная управляемость.

Вначале такая база данных хорошо работала на сайте, но вскоре она выросла до крупномасштабных операций. Сейчас на сайте размещается 7 миллионов блогов на 300 серверах баз данных. На каждом сервере находится подмножество клиентов.

Когда Барри добавляет сервер, бывает очень трудно в пределах одной базы данных разделить данные, принадлежащие блогу отдельного клиента. Разделив сведения в отдельных базах данных по клиентам, он значительно упростил операции перемещения любого отдельного блога с одного сервера на другой. Так как клиенты приходят и уходят, при этом блоги некоторых клиентов являются оживленными, в то время как другие блоги становятся неинтересными, его работа по перераспределению нагрузки по многим серверам приобретает дополнительную важность.

Проще создать резервную копию и восстановить отдельные базы данных среднего размера, чем единую базу данных, содержащую терабайты информации. Например, если клиент звонит и говорит, что в его данных полнейшая неразбериха из-за плохого ввода данных, возникает проблема, как восстановить данные одного

клиента, если все клиенты совместно пользуются единой, монолитной резервной копией базы данных?

Хотя с точки зрения моделирования данных кажется правильным сохранять все в единой базе данных, ее разделение заметно упрощает задачи администрирования базы данных после того, как размер базы данных превысит некий порог.

9.5. РЕШЕНИЕ: РАЗДЕЛЕНИЕ И НОРМАЛИЗАЦИЯ

Когда таблица становится слишком большой, вместо разделения таблицы вручную можно воспользоваться более оптимальными способами повышения производительности работы. К таким способам относятся горизонтальное разделение, вертикальное разделение и применение зависимых таблиц.

Использование горизонтального разделения

Преимущества разделения большой таблицы можно получить без сопутствующих недостатков, если воспользоваться функцией, называемой либо *горизонтальным разделением* (horizontal partitioning), либо *разбиением на части* (sharding). Следует определить логическую таблицу с некоторым правилом разделения строк на отдельные разделы, а остальным управляет база данных. Физически таблица разделена, но в отношении таблицы можно по-прежнему выполнять операторы SQL, как если бы таблица была целой.

Предоставляется гибкая возможность определения способа разделения для строк каждой отдельной таблицы в индивидуальное хранилище. Например, используя поддержку разделения в MySQL версии 5.1, можно задать разделы в качестве дополнительной части оператора `CREATE TABLE`.

Файл примера: *Metadata-Tribbles/soln/horiz-partition.sql*

```
CREATE TABLE Bugs (
   bug_id SERIAL PRIMARY KEY,
   -- другие столбцы
   date_reported DATE
) PARTITION BY HASH ( YEAR(date_reported) )
   PARTITIONS 4;
```

В предыдущем примере достигается разделение, похожее на то, которое было показано ранее в данной главе, — разделение строк на основе года в столбце `date_reported`. Однако его преимущества по сравнению с разделением таблицы вручную заключаются в том, что строки никогда не помещаются в неправильную разделенную таблицу, даже если значение столбца `date_reported` обновляется, при этом можно выполнять запросы по таблице `Bugs` без необходимости ссылки на отдельные разделенные таблицы.

В этом примере число отдельных физических таблиц, используемых для хранения строк, фиксировано и равно четырем. Когда в наличии есть строки, охватывающие более четырех лет, один из разделов будет использоваться для хранения данных более чем за один год. Работа этого алгоритма будет продолжаться по мере того, как будут проходить годы. Новые разделы не требуется добавлять, если только объем данных не станет таким большим, что вы посчитаете необходимым разделить его дополнительно.

Разделение не определяется в стандарте SQL, так что в базе данных каждого конкретного производителя оно реализуется своим, нестандартным способом. Терминология, синтаксис и специальные функции разделения варьируются между базами данных разных марок. Тем не менее в настоящее время каждой известной базой данных поддерживается какой-нибудь вид разделения.

Использование вертикального разделения

В то время как с помощью горизонтального разделения таблицу разбивают по строкам, вертикальное разделение позволяет разбить таблицу по столбцам. Разделение таблицы по столбцам может обладать преимуществами, когда некоторые столбцы являются объемными или редко востребованы.

Размер столбцов `BLOB` и `TEXT` варьируется, и они могут быть очень большими. Для обеспечения эффективности как хранения, так и извлечения в базах данных многих производителей столбцы с этими типами данных автоматически сохраняются отдельно от других столбцов заданной строки. Если выполнять запрос без ссылки на какой-либо столбец `BLOB` или `TEXT` в таблице, доступ к другим столбцам будет более эффективным. Но если использовать в запросе знак подстановки «*» для столбца, базой данных извлекаются из этой таблицы все столбцы, включая все столбцы `BLOB` и `TEXT`.

Например, в таблице `Products` нашей базы данных ошибок может храниться копия установочного файла для соответствующего продукта. Этот файл обычно является самораспаковывающимся архивом с расширением, таким как «.exe» в Windows или «.dmg» в операционной системе Mac OS. Данные файлы обычно очень большие, но в столбце BLOB могут храниться двоичные данные огромного размера.

По логике файл инсталлятора должен быть атрибутом таблицы `Products`. Но в большинстве запросов по данной таблице не будет требоваться инсталлятор. Хранение в таблице `Products` такого большого объема данных, используемых достаточно редко, могло бы привести к непреднамеренным проблемам с производительностью, если существует привычка извлекать все столбцы с помощью знака подстановки «*».

Спасительное средство состоит в хранении столбца `BLOB` в другой отдельной таблице, отличной от таблицы `Products`, но зависимой от нее. Сделай-

те так, чтобы ее первичный ключ служил также внешним ключом для таблицы Products с целью обеспечения наличия максимум одной строки, приходящейся на каждую строку таблицы продуктов.

Файл примера: *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)
);
```

Предыдущий пример является предельным случаем разделения, но он показывает преимущество хранения некоторых столбцов в отдельной таблице. Например, в механизме хранения MyISAM, используемом в MySQL, запрос таблицы является самым эффективным, когда размер строк фиксирован. VARCHAR — тип данных переменной длины, поэтому присутствие в таблице одного столбца с таким типом данных не позволяет воспользоваться данным преимуществом в таблице. Если все столбцы переменной длины хранить в отдельной таблице, тогда запросы по первичной таблице могут выиграть от этого (хотя, возможно, в очень малой степени).

Файл примера: *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)
);
```

Фиксация столбцов трибблов метаданных

Аналогично решению, приведенному в главе 8, спасительное средство для столбцов «трибблов метаданных» состоит в создании зависимой таблицы.

Файл примера: *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 для каждого года используйте несколько строк с одним столбцом для исправленных ошибок. Если определить таблицу таким способом, не надо добавлять новые столбцы для поддержки последующих лет. С течением времени в такой таблице можно хранить любое количество строк по каждому проекту.

ВНИМАНИЕ!

Не допускайте порождения данными метаданных.

ЧАСТЬ II. АНТИПАТТЕРНЫ ФИЗИЧЕСКОЙ СТРУКТУРЫ БАЗЫ ДАННЫХ

10,0 помноженное на 0,1, почти никогда не равно 1,0.

Брайан Керниган

ГЛАВА 10. ОШИБКИ ОКРУГЛЕНИЯ

Ваш начальник просит подготовить отчет о стоимости времени программистов по проекту на основе общей работы по устранению всех ошибок. У каждого программиста в таблице `Accounts` есть своя почасовая ставка, поэтому записывается количество часов (`hours`), необходимых для устранения каждой ошибки в таблице `Bugs`, и это значение умножается на почасовую ставку (`hourly_rate`) программиста, которому выделена работа.

Файл примера: *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);
```

Для поддержки данного запроса необходимо создать новые столбцы в таблицах `Bugs` и `Accounts`. Обоими столбцами должны поддерживаться дробные значения, поскольку затраты требуется отслеживать с высокой точностью. Решено определить новые столбцы как `FLOAT`, так как этим типом данных поддерживаются дробные значения.

Файл примера: *Rounding-Errors/intro/float-columns.sql*

```
ALTER TABLE Bugs ADD COLUMN hours FLOAT;

ALTER TABLE Accounts ADD COLUMN hourly_rate FLOAT;
```

Вы обновляете столбцы сведениями из рабочих журналов ошибок, а также ставками программистов, тестируете отчет и ежедневно вызываете его.

На следующий день в офис пришел ваш начальник с копией отчета по стоимости проекта. «Эти числа не суммируются», — сообщает он сквозь стиснутые зубы. «Я выполнил расчеты вручную для сравнения, и ваш отчет неточен — незначительно, всего на несколько долларов. Как вы объясните это?» Вы начинаете покрываться испариной. Что могло быть неправильным в таком простом вычислении?

10.1. ЦЕЛЬ: ИСПОЛЬЗОВАНИЕ ДРОБНЫХ ЗНАЧЕНИЙ ВМЕСТО ЦЕЛЫХ ЧИСЕЛ

Целочисленное значение (`integer`) — полезный тип данных, но он позволяет хранить только целые числа, такие как 1, или 327, или −19. Он не позволяет представлять дробные значения, такие как 2,5. Требуется другой тип данных, если нужны числа с большей точностью, чем целочисленные значения. Например, суммы денег обычно представляются числами с двумя знаками после запятой, например RUR19,95.

Таким образом, цель состоит в хранении числовых значений, которые не являются целыми числами, и использовании их в арифметических расчетах. Существует дополнительная цель, хотя она должна достигаться безоговорочно: результаты арифметических расчетов должны быть *правильными*.

10.2. АНТИПАТТЕРН: ИСПОЛЬЗОВАНИЕ ТИПА ДАННЫХ FLOAT

Большинством языков программирования поддерживается тип данных для реальных чисел, носящий название `float` или `double`. Языком SQL поддерживается похожий тип данных с таким же именем. Многие программисты, естественно, используют SQL-тип данных `FLOAT` везде, где необходимы дробные числовые данные, так как они привыкли программировать с типом данных `float`.

Тип данных `FLOAT` в SQL, подобно `float` в большинстве языков программирования, позволяет кодировать действительное число в двоичном формате согласно стандарту IEEE 754. Необходимо понимать некоторые свойства чисел с плавающей запятой в данном формате, чтобы использовать их эффективно.

Округление по необходимости

Многие программисты не знают о свойствах данного формата с плавающей запятой: не все значения, которые можно описать в десятичных дробях, могут храниться в двоичном виде. По необходимости некоторые числа должны округляться до очень близкого значения.

Чтобы привести некоторый контекст поведения такого округления, сравните его с рациональными числами, такими как $1/3$, представляемыми периодической дробью, например 0,333.... Истинное значение не может быть представлено десятичной дробью, поскольку потребовалось бы писать бесконечное число цифр. Число цифр является точностью числа, поэтому периодическая дробь потребовала бы *бесконечной точности*.

Компромисс состоит в использовании *конечной точности* путем выбора числового значения, находящегося как можно ближе к исходному значе-

нию, например 0,333. Однако это означает, что значение не является в точности тем значением, которое предполагалось использовать.

$$\frac{1}{3}+\frac{1}{3}+\frac{1}{3} = 1{,}000$$
$$0{,}333 + 0{,}333 + 0{,}333 = 0{,}999$$

Даже если увеличить точность, по-прежнему нельзя будет сложить три из этих приближений одной третьей, чтобы получить истинное значение, равное 1,0. Это необходимый компромисс использования конечной точности для представления чисел, которые могут содержать периодические дроби.

$$\frac{1}{3}+\frac{1}{3}+\frac{1}{3} = 1{,}000\dot{0}00$$
$$0{,}333333 + 0{,}333333 + 0{,}333333 = 0{,}999999$$

Это означает, что некоторые допустимые числа, которые можно себе вообразить, не могут быть представлены с конечной точностью. Можно полагать, что это хорошо, так как в действительности нельзя ввести число с бесконечным количеством цифр, поэтому, естественно, любое число, которое допустимо для ввода, представляется с конечной точностью и должно храниться точно, не так ли? К сожалению, нет.

Согласно стандарту IEEE 754 числа с плавающей запятой представляются в формате двоичной системы счисления. Значения, для которых требуется бесконечная точность в двоичном формате, отличаются от чисел, которые ведут себя таким же образом в десятичной дроби. Некоторые значения, которым требуется только конечная точность в десятичной дроби, например 59,95, нуждаются в бесконечной точности для точного представления в двоичном виде. Тип данных `FLOAT` не позволяет делать этого, поэтому им используется ближайшее значение, сохраняемое в двоичном формате, которое равно 59,950000762939 в десятичном формате.

Некоторыми значениями по случайному стечению обстоятельств используется конечная точность в обоих форматах. В теории, если есть понимание деталей хранения чисел в формате IEEE 754, можно предсказать, как заданное десятичное значение представляется в двоичном виде. Но на практике большинство людей не делают данного расчета для каждого используемого значения с плавающей запятой. Нельзя гарантировать, что в столбец `FLOAT` в базе данных будут заноситься только скоординированные значения, поэтому приложение должно исходить из того, что любое значение в данном столбце может округляться.

Некоторыми базами данных поддерживаются родственные типы данных, называемые DOUBLE PRECISION и REAL. Точность, обеспечиваемая этими типами данных и FLOAT, изменяется в зависимости от реализации базы данных, но все они представляют значения в формате с плавающей запятой, используя конечное число двоичных разрядов, поэтому во всех этих форматах округление производится одинаковым образом.

Использование FLOAT в SQL

В некоторых базах данных неточность может компенсироваться, и отображаются предполагаемые значения.

СОБЛЮДЕНИЕ ФОРМАТА IEEE 754

Предложения по стандартному двоичному формату для чисел с плавающей запятой датируются 1979 годом. Официально этот формат был стандартизован в 1985 году, и сейчас он широко реализуется в программном обеспечении, большинстве языков программирования и в микропроцессорах.

Формат содержит три поля для кодирования значения с плавающей запятой: поле для **дробной** части значения, поле для экспоненты, в которую возводится дробь, и одноразрядное поле **знака**.

Преимущество формата IEEE 754 заключается в том, что, используя экспоненту, он позволяет представить как очень маленькие, так и очень большие дробные значения. Форматом не только поддерживаются действительные числа, но к тому же диапазон поддерживаемых им значений больше, чем диапазон целых чисел в формате с фиксированной запятой. Форматом с двойной точностью поддерживается еще больший диапазон значений. Таким образом, эти форматы удобны для научных приложений.

Однако дробные числовые значения, возможно, наиболее широко используются для представления денежных сумм. Нет необходимости использовать формат IEEE 754 для денег, поскольку масштабированный десятичный формат, описываемый в данной главе, позволяет обрабатывать значения денежных сумм так же просто и с большей точностью.

Хорошими источниками дополнительных сведений о данном формате являются статья в Википедии (**en.wikipedia.org/wiki/IEEE_754-1985**) и статья Давида Гольдберга (David Goldberg) «What Every Computer Scientist Should Know About Floating-Point Arithmetic» [8].

Статья Гольдберга также представлена по адресу **www.validlab.com/goldberg/paper.pdf**.

Файл примера: *Rounding-Errors/anti/select-rate.sql*

```
SELECT hourly_rate FROM Accounts WHERE account_id = 123;
```

Возвращает: 59,95.

Но фактическое значение, хранящееся в столбце FLOAT, может быть не равно точно данному значению. Если увеличить значение в миллион раз, будет видно различие:

Файл примера: *Rounding-Errors/anti/magnify-rate.sql*

```
SELECT hourly_rate * 1000000000 FROM Accounts WHERE account_id
= 123;
```

Возвращает: 59950000762,939.

Можно было бы ожидать, что увеличенное значение, возвращаемое предыдущим запросом, будет равно 59950000000,000. Это показывает, что значение 59,95 округляется до значения, которое может быть представлено с конечной точностью, обеспечиваемой двоичным форматом IEEE 754. В данном случае значение находится в пределах одной десятимиллионной, что вполне достаточно для многих расчетов.

Однако округленное значение недостаточно точно для некоторых других видов расчетов. Один из примеров — применение FLOAT в сравнениях на определение равенства.

Файл примера: *Rounding-Errors/anti/inexact.sql*

```
SELECT * FROM Accounts WHERE hourly_rate = 59.95;
```

Результат: пустой набор; совпадающих строк нет.

Выше было показано, что значение, хранящееся в `hourly_rate`, в действительности несколько больше, чем 59,95. Поэтому хотя данному столбцу и было присвоено значение 59,95 для `account_id` 123, теперь строка не соответствует условиям предыдущего запроса.

Один из распространенных способов обхода этой проблемы состоит в том, чтобы считать значения с плавающей запятой как «эффективно равные», если они близки друг к другу в пределах небольшого порога. Вычтите одно значение из другого и воспользуйтесь SQL-функцией абсолютного значения `ABS()`, чтобы убрать знак у разницы. Если результат равен нулю, тогда два значения были в точности равны друг другу. Если результат относительно мал, тогда два значения могут считаться как эффективно равные. Следующий запрос позволяет найти требуемую строку:

Файл примера: *Rounding-Errors/anti/threshold.sql*

```
SELECT * FROM Accounts WHERE ABS(hourly_rate - 59.95) < 0.000001;
```

Однако разница все еще достаточно велика, так что сравнение с большей точностью потерпит неудачу:

Файл примера: *Rounding-Errors/anti/threshold.sql*

```
SELECT * FROM Accounts WHERE ABS(hourly_rate - 59.95) <
0.0000001;
```

Подходящий порог варьируется, так как изменяется абсолютная разница между значением в десятичном формате и округленным значением в двоичном виде.

Еще одним примером природы приближенного характера FLOAT, являющейся источником проблем точности, служит вычисление сводных показателей по многим значениям. Например, если используется функция SUM() для суммирования значений с плавающей запятой в столбце, суммой накапливается разница, вызываемая округлением всех значений.

Файл примера: *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);
```

Кумулятивное влияние неточных чисел с плавающей запятой еще более серьезно, когда рассчитывается совокупное произведение набора чисел вместо их суммы. Разница кажется маленькой, но она комбинируется. Например, если умножить значение 1 на коэффициент, точно равный 1,0, результат всегда будет равен 1. Не важно, сколько раз выполняется умножение на данный коэффициент. Однако если коэффициент на самом деле равен 0,999, то будет получаться другой результат. Если умножить значение, равное единице, на 0,999 последовательно тысячу раз, получится результат около 0,3677. Чем больше количество умножений, тем больше будет разница.

Хорошим примером применения многократного последовательного умножения служит вычисление сложного процента в финансовом приложении. При использовании неточных чисел с плавающей запятой вводится ошибка, которая кажется крошечной, но при комбинировании она возрастает. Поэтому использование точных значений в финансовых документах крайне важно.

10.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

В сущности, любое применение типов данных FLOAT, REAL или DOUBLE PRECISION вызывает подозрения. В большинстве приложений, использующих числа с плавающей запятой, не требуется диапазон значений, поддерживаемых форматом IEEE 754.

Применение типов данных FLOAT в SQL кажется естественным, так как название этого типа данных является общим для типов данных в большинстве языков программирования. Тем не менее существует лучший вариант выбора типа данных.

10.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

FLOAT — хороший тип данных, когда требуются значения действительных чисел с диапазоном, который шире диапазонов, поддерживаемых типами данных INTEGER или NUMERIC. Научные приложения часто цитируются как наилучшая область применения FLOAT.

В Oracle применение типа данных FLOAT означает точные масштабируемые числа, тогда как тип данных BINARY_FLOAT является неточным числовым форматом, использующим кодирование IEEE 754.

10.5. РЕШЕНИЕ: ИСПОЛЬЗОВАНИЕ ТИПА ДАННЫХ NUMERIC

Для дробных чисел фиксированной точности вместо FLOAT и родственных ему типов используйте SQL-типы данных NUMERIC или DECIMAL.

Файл примера: *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 обеспечивают хранение точных числовых значений вплоть до точности, указываемой в определении столбца. Укажите точность как аргумент для типа данных аналогично синтаксису, который используется для длины типа данных VARCHAR. Точность — это общее количество десятичных разрядов, которое может использоваться в значении, содержащемся в данном столбце. Точность, равная 9, означает, что можно хранить значение, подобное 123456789, но при этом может отсутствовать возможность хранения числа 1234567890[1].

Можно также задать *масштаб* во втором аргументе типа данных. Масштаб — это количество разрядов справа от десятичной точки. Эти разряды включаются в разряды точности, так что точность, равная 9, с масштабом, равным 2, означает, что можно хранить значение, подобное 1234567,89, но не 12345678,91 и не 123456,789.

Задаваемые значения точности и масштаба применяются к столбцу во всех строках в таблице. Другими словами, нельзя хранить значения с масштабом, равным 2, в некоторых строках и с масштабом, равным 4, в других строках. Обычно в SQL тип данных столбца применяется единообразно ко всем строкам (точно как столбец, определяемый как VARCHAR(20), разрешал бы символьные строки этой длины в каждой строке).

---

[1] В базах данных некоторых марок размер столбца округляется внутри до ближайшего байта, слова или двойного слова, так что максимальное значение столбца NUMERIC может содержать больше разрядов, чем указанная точность.

Преимущество типов данных NUMERIC и DECIMAL заключается в том, что они позволяют хранить рациональные числа без округления, которое выполняется для типа данных FLOAT. После того как задано значение 59,95, можно быть уверенным, что это значение хранится точным. При сравнении его на равенство с буквенным значением 59,95 операция сравнения завершается успехом.

Файл примера: *Rounding-Errors/soln/exact.sql*

```
SELECT hourly_rate FROM Accounts WHERE hourly_rate = 59.95;
```

Возвращает: 59,95.

Подобным образом, если увеличить значение, умножив его на миллион, получится ожидаемое значение:

Файл примера: *Rounding-Errors/soln/magnify-rate-exact.sql*

```
SELECT hourly_rate * 1000000000 FROM Accounts WHERE hourly_
rate = 59.95;
```

Возвращает: 59950000000.

Типы данных NUMERIC и DECIMAL ведут себя одинаково; между ними не должно быть разницы. Синонимом типа данных DECIMAL является также DEC.

По-прежнему нельзя хранить значения, для которых требуется бесконечная точность, например одну третью. Но, по крайней мере, теперь нам стали более знакомы значения, на которые распространяется это ограничение в десятичном формате.

Если требуются точные десятичные значения, используйте тип данных NUMERIC. Тип данных FLOAT не позволяет представлять многие десятичные рациональные числа, поэтому они должны обрабатываться как неточные значения.

ВНИМАНИЕ!

Не используйте тип данных FLOAT, если можно обойтись без него.

Наука состоятельна, когда переменных всего несколько и их можно перечислить; когда их сочетания индивидуальны и ясны.

Поль Валери

ГЛАВА 11. 31 РАЗНОВИДНОСТЬ

В таблице персональных контактных сведений *приветствие* является хорошим примером столбца, в котором находится только несколько значений. Если поддерживаются приветствия «мистер» (*Mr.*), «миссис» (*Mrs.*), «мисс» (*Ms.*), «доктор» (*Dr.*), «преподобие» (*Rev.*), можно считать, что охвачены практически все варианты обращения. Данный список можно задать в определении столбца, используя тип данных или ограничение, так что никто не сможет случайно ввести неправильную строку в столбец `salutation`.

Файл примера: *31-Flavors/intro/create-table.sql*

```
CREATE TABLE PersonalContacts (
-- другие столбцы
   salutation VARCHAR(4)
   CHECK (salutation IN ('Mr.', 'Mrs.', 'Ms.', 'Dr.', 'Rev.')),
);
```

Этот код должен решить проблему, так как не существует других приветствий, которые надо поддерживать, не так ли?

К сожалению, ваш начальник сообщает вам, что ваша компания открывает филиал во Франции. В связи с этим требуется обеспечить поддержку приветствий «*мадам*» и «*мадемуазель*». Ваше задание заключается в изменении таблицы контактов, чтобы разрешить использование этих значений. Это деликатная работа, и она невозможна без отключения доступа к этой таблице.

Вы также вспоминаете мимоходную реплику начальника о планах открыть в следующем месяце офис в Бразилии.

11.1. ЦЕЛЬ: ОГРАНИЧЕНИЕ СТОЛБЦА КОНКРЕТНЫМИ ЗНАЧЕНИЯМИ

Ограничение значений столбца фиксированным набором значений очень полезно. Если мы сможем обеспечить, чтобы столбец никогда не содержал недопустимых записей, это поможет упростить использование такого столбца.

31 ВИД МОРОЖЕНОГО КОМПАНИИ БАСКИН-РОББИНС

В 1953 году эта знаменитая сеть кафе-мороженых предлагала по одному виду мороженого в каждый день месяца. Этой сетью в течение многих лет использовался девиз: «31 вид мороженого».

Сегодня, спустя более чем 60 лет, компания Баскин-Роббинс (Baskin-Robbins) предлагает 21 вид классического мороженого, 12 сезонных видов, 16 региональных видов, а также многообразие привлекательных вариантов и разновидностей каждый месяц. Хотя их разновидности мороженого были когда-то постоянным набором, который определял марку компании, Баскин-Роббинс расширили предлагаемые варианты и сделали его перестраиваемым и изменяемым.

Подобные вещи могут происходить в проекте, для которого разрабатывается база данных, — в реальности следует рассчитывать на это.

Например, в таблице `Bugs` нашего примера базы данных столбец `status` показывает, является ли заданная ошибка *NEW*, *IN PROGRESS*, *FIXED* и так далее. Важность каждого из этих значений статуса зависит от того, как осуществляется управление ошибками в проекте, но вопрос заключается в том, что данные в столбце должны относиться к одному из этих типов.

В идеальном случае базой данных должны отклоняться недопустимые данные:

Файл примера: *31-Flavors/obj/insert-invalid.sql*

```
INSERT INTO Bugs (status) VALUES ('NEW'); -- OK

INSERT INTO Bugs (status) VALUES ('BANANA'); -- Ошибка!
```

11.2. АНТИПАТТЕРН: ЗАДАНИЕ ЗНАЧЕНИЙ В ОПРЕДЕЛЕНИИ СТОЛБЦА

Многие предпочитают задавать допустимые значения данных при определении столбца. Определение столбца является частью *метаданных* — определения самой структуры таблицы.

Например, можно было бы определить *ограничение проверки* по столбцу. Этим ограничением запрещаются любые вставки и обновления, которые бы сделали это ограничение ложным.

Файл примера: *31-Flavors/anti/create-table-check.sql*

```
CREATE TABLE Bugs (
   -- другие столбцы
   status VARCHAR(20) CHECK (status IN ('NEW', 'IN PROGRESS',
   'FIXED'))
);
```

Базой данных MySQL поддерживается нестандартный тип данных, называемый ENUM, который ограничивает столбец конкретным набором значений.

Файл примера: *31-Flavors/anti/create-table-enum.sql*

```
CREATE TABLE Bugs (
   -- другие столбцы
   status ENUM('NEW', 'IN PROGRESS', 'FIXED'),
);
```

В реализации MySQL значения объявляются как символьные строки, но внутри системы столбец хранится как порядковый номер символьной строки в нумерованном списке. Поэтому хранилище является компактным, но при сортировке запроса по данному столбцу результаты упорядочиваются по порядковому номеру, а не в алфавитном порядке строковых значений. Возможно, это поведение будет неожиданным для вас.

Другие решения включают *домены* и *пользовательские типы*. Эти решения можно использовать для ограничения столбца конкретным набором значений, и один и тот же домен или тип данных легко применять к нескольким столбцам в базе данных. Но данные функции пока не поддерживаются широко среди СУБД многих марок.

В конце концов, можно было бы написать триггер, содержащий набор разрешенных значений и генерирующий ошибку в случае, если состояние (status) не совпадает с одним из заданных значений.

Все эти решения характеризуются рядом недостатков. В следующих разделах описываются некоторые из этих проблем.

Что было средним?

Предположим, что вы разрабатываете пользовательский интерфейс для системы отслеживания ошибок, позволяющий редактировать отчеты об ошибках. Чтобы сделать интерфейс, направляющий пользователя на выбор одного из допустимых значений статуса (status), вы решаете заполнить элемент управления выпадающего меню этими значениями. Как запросить в базе данных нумерованный список значений, который в текущий момент разрешен в столбце status?

Возможно, у вас сразу возникнет желание запросить все значения, используемые в текущий момент с помощью простого запроса, подобного приведенному ниже:

Файл примера: *31-Flavors/anti/create-table-enum.sql*

```
SELECT DISTINCT status FROM Bugs;
```

Однако если все ошибки являются новыми, предыдущим запросом будет возвращаться только значение *NEW*. Если использовать этот результат, чтобы заполнить элемент управления пользовательского интерфейса для статуса (`status`) ошибок, можно попасть в ситуацию, когда трудно определить причину и следствие; нельзя будет присвоить ошибке статус, отличающийся от тех, что используются в текущий момент.

Чтобы получить полный список разрешенных значений статуса (`status`), необходимо запросить определение метаданных этого столбца. Большинством SQL-баз данных поддерживаются для этих видов запросов системные представления, но их применение может быть сложным. Например, если использовать тип данных `ENUM`, присутствующий в MySQL, то для запроса системных представлений `INFORMATION_SCHEMA` можно воспользоваться следующим запросом:

Файл примера: *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';
```

Нельзя просто получить дискретные значения перечисления из `INFORMATION_SCHEMA` в обычном наборе результатов. Вместо этого следует получить строку, содержащую определение ограничения проверки или тип данных `ENUM`. Например, предыдущий запрос в MySQL возвращает столбец типа `LONGTEXT` со значением `ENUM('NEW', 'IN PROGRESS', 'FIXED')`, включая скобки, запятые и одинарные кавычки. Необходимо написать код приложения для синтаксического анализа этой строки и извлечения отдельных значений в кавычках, прежде чем их можно будет использовать для заполнения элемента управления пользовательского интерфейса.

Запросы, необходимые для вывода сообщений об ограничениях проверки, доменах или пользовательских типах, последовательно усложняются. Большинство выбирает оптимальный вариант поведения и вручную сопровождают параллельный список значений в прикладном коде. Это простой способ влияния ошибок на проект, так как прикладные данные перестают координироваться с метаданными базы данных.

Добавление новой разновидности

Наиболее распространенные изменения — это добавления и удаления одного из разрешенных значений. Не существует синтаксиса для добавления (или удаления) значения в ENUM или в ограничение проверки; можно только переопределить столбец с новым набором значений. Ниже приводится пример добавления *DUPLICATE* в качестве одного нового значения статуса (status) в ENUM базы данных MySQL:

Файл примера: *31-Flavors/anti/add-enum-value.sql*

```
ALTER TABLE Bugs MODIFY COLUMN status
   ENUM('NEW', 'IN PROGRESS', 'FIXED', 'DUPLICATE');
```

Необходимо убедиться, что предыдущее определение столбца разрешало применение значений *NEW*, *IN PROGRESS* и *FIXED*. Эта задача возвращает нас к трудности запроса текущего набора значений, как было описано ранее.

Базы данных некоторых марок не позволяют изменять определение столбца, если таблица не пустая. Возможно, потребуется выгрузить содержимое таблицы, переопределить таблицу, а затем импортировать сохраненные данные, что делает таблицу недоступной в это время. Данная операция достаточно распространена, поэтому у нее даже есть имя *ETL*, обозначающее «извлечение, преобразование и загрузку» (ETL — extract, transform and load). Базы данных других марок поддерживают реструктуризацию заполненной таблицы с помощью команд ALTER TABLE, однако выполнение этих изменений по-прежнему отличается сложностью и дороговизной. В рамках проводимой политики изменение метаданных — то есть изменение определения таблиц и столбцов — должно быть нечастым, при этом следует уделять внимание тестированию и обеспечению качества. Если требуется изменить данные с целью добавления или удаления значения из ENUM, тогда надо или пропустить соответствующую проверку, или потратить немало усилий разработчиков программного обеспечения на краткую заметку, чтобы внести изменение. При применении любого из способов эти изменения связаны с риском и дестабилизируют проект.

Старые разновидности никогда не теряют актуальности

Если вывести значение из употребления, можно нарушить данные за длительный период времени. Например, вы изменяете процесс контроля качества для замены ступени *FIXED* двумя этапами, *CODE COMPLETE* и *VERIFIED*:

Файл примера: *31-Flavors/anti/remove-enum-value.sql*

```
ALTER TABLE Bugs MODIFY COLUMN status
   ENUM('NEW', 'IN PROGRESS', 'CODE COMPLETE', 'VERIFIED');
```

Если удалить значение *FIXED* из перечисления, что тогда делать с ошибками, у которых статус (`status`) был *FIXED*? Следует ли перевести все ошибки со статусом *FIXED* в состояние *VARIFIED*? Следует ли вместо этого установить значения, вышедшие из употребления, равными `Null` или равными значениям по умолчанию?

Возможно, потребуется сохранить вышедшее из употребления значение, на которое ссылаются старые строки. Но тогда как можно отличать вышедшие из употребления значения и исключить их из пользовательского интерфейса, чтобы никто не смог установить статус ошибки равным вышедшему из употребления значению?

Переносимость — трудный процесс

Ограничения проверки, домены и пользовательские типы не поддерживаются единообразно в SQL-базах данных разных марок. Тип данных `ENUM` является фирменной функцией в MySQL. В базе данных каждой модели могут быть разные пределы на длину списка, которую можно задавать в определении столбца. Языки триггеров так же варьируются в разных базах данных. Эти различия затрудняют выбор решения, если необходимо обеспечить поддержку баз данных нескольких марок.

11.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Проблемы, связанные с использованием `ENUM` или ограничения проверки, возникают, когда набор значений не фиксирован. Если рассматривается возможность использования `ENUM`, сначала задайте себе вопрос, предполагается ли изменять набор значений, или, точнее, могут ли изменяться значения набора. Если да, то, вероятно, сейчас неподходящее время для использования `ENUM`.

- «Нам требуется перевести базу данных в автономный режим, чтобы можно было добавить новый вариант выбора в одно из меню приложения. Это должно занять не более 30 минут, если все пойдет хорошо».

Такое заявление — признак вставки набора значений в определение столбца. Для внесения подобного изменения никогда не требуется прерывать работу службы.

- «Столбец `status` может содержать одно из следующих значений. В изменении этого списка нет необходимости».

«Нет необходимости» — обтекаемое выражение, и этим говорится кое-что отличное от *«нельзя»*.

- «Список значений в прикладном коде перестал согласовываться с бизнес-правилами в базе данных — опять».

Такая ситуация связана с риском сохранения информации в двух разных местах.

11.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Как уже обсуждалось выше, тип данных `ENUM` может приводить к нескольким проблемам, если набор значений не допускает изменений. По-прежнему трудно запросить метаданные для набора значений, но можно вести согласующийся список значений в коде приложения, избегая рассогласования двух списков.

Вполне вероятно, что тип данных `ENUM` может успешно применяться, когда нет смысла изменять набор разрешенных значений, например когда столбец представляет альтернативный выбор с двумя взаимно исключающими значениями: *LEFT/RIGHT*, *ACTIVE/INACTIVE*, *ON/OFF*, *INTERNAL/EXTERNAL* и так далее.

Ограничения проверки могут использоваться многими способами, отличными от простой реализации ENUM-подобного механизма, например для проверки того, что начало временного интервала меньше, чем его конец.

11.5. РЕШЕНИЕ: ЗАДАНИЕ ЗНАЧЕНИЙ В ДАННЫХ

Существует гораздо лучшее решение по ограничению значений в столбце: создайте *таблицу поиска* с одной строкой для каждого значения, которое допускается в столбце `Bugs.status`. Затем объявите ограничение внешнего ключа по таблице `Bugs.status`, ссылающееся на новую таблицу.

Файл примера: *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
);
```

Когда вставляют или обновляют строку в таблице `Bugs`, следует использовать значение `status`, существующее в таблице `BugStatus`. Этим принудительно вводятся в действие значения статуса, подобно `ENUM` или ограничению проверки, однако существует несколько способов, в которых данное решение обеспечивает большую гибкость.

Запрос набора значений

Набор разрешенных значений теперь хранится в данных, а не в метаданных, как это было в случае типа данных `ENUM`. Можно запросить значения данных из таблицы поиска с помощью оператора `SELECT`, как и в любой другой таблице. Это упрощает извлечение набора значений как набора данных для представления в пользовательском интерфейсе. Существует даже возможность сортировки набора значений, из которого пользователь осуществляет выбор.

Файл примера: *31-Flavors/soln/query-canonical-values.sql*

```
SELECT status FROM BugStatus ORDER by status;
```

Обновление значений в таблице поиска

Когда применяется таблица поиска, можно добавить значение в набор с помощью обычного оператора `INSERT`. Изменение, подобное упомянутому выше, можно внести без прерывания доступа к таблице. Не надо переопределять какие-либо столбцы, планировать простой или выполнять операцию ETL. К тому же, чтобы добавить или удалить значение, не требуется знать текущий набор значений в таблице поиска.

Файл примера: *31-Flavors/soln/insert-value.sql*

```
INSERT INTO BugStatus (status) VALUES ('DUPLICATE');
```

Также легко можно переименовать значение, если объявлен внешний ключ с помощью параметра `ON UPDATE CASCADE`.

Файл примера: *31-Flavors/soln/update-value.sql*

```
UPDATE BugStatus SET status = 'INVALID' WHERE status = 'BOGUS';
```

Поддержка устаревших значений

Нельзя удалить строку (выполнив операцию DELETE) из таблицы поиска, если на нее ссылается строка в таблице Bugs. Внешний ключ по столбцу status принудительно вводит в действие целостность на уровне ссылок, так что значение должно присутствовать в таблице поиска.

Тем не менее в таблицу поиска можно добавить еще один столбец атрибутов, чтобы обозначить некоторые значения как устаревшие. Это позволит сохранять данные за прошлый период времени в столбце Bugs.status, при этом различая устаревшие значения от тех, которые могут появляться в пользовательском интерфейсе.

Файл примера: *31-Flavors/soln/inactive.sql*

```
ALTER TABLE BugStatus ADD COLUMN active
   ENUM('INACTIVE', 'ACTIVE') NOT NULL DEFAULT 'ACTIVE';
```

Используйте оператор UPDATE вместо DELETE, чтобы перевести значение в разряд устаревших:

Файл примера: *31-Flavors/soln/update-inactive.sql*

```
UPDATE BugStatus SET active = 'INACTIVE' WHERE status = 'DU-
PLICATE';
```

Когда извлекаете набор значений для отображения в пользовательском интерфейсе в качестве вариантов выбора пользователями, ограничьте запрос значениями статуса, представленными значениями *ACTIVE*:

Файл примера: *31-Flavors/soln/select-active.sql*

```
SELECT status FROM BugStatus WHERE active = 'ACTIVE';
```

Этот прием обеспечивает большую гибкость по сравнению с ENUM и ограничением проверки, так как данными решениями не поддерживаются дополнительные атрибуты по каждому значению.

Простая переносимость

В отличие от типа данных ENUM, ограничений проверки, доменов и пользовательских типов решение в виде таблицы поиска основывается только на стандартной SQL-функции декларативной целостности на уровне ссылок,

использующей ограничения внешнего ключа. Этот метод характеризуется расширенными возможностями переносимости.

В таблице поиска можно также сохранять практически неограниченное число значений, так как каждое значение хранится в отдельной строке.

ВНИМАНИЕ!

Используйте метаданные для проверки достоверности по фиксированному набору значений.

Для проверки достоверности по изменяющемуся набору значений используйте данные.

Всякий раз, когда теория представляется вам как единственно возможная, это свидетельствует о том, что вы не понимаете ни теорию, ни проблему, которую она призвана решить.

Карл Поппер

ГЛАВА 12. ФАНТОМНЫЕ ФАЙЛЫ

С вашим сервером базы данных случилась катастрофа. При перемещении в другое место стойки, заполненной жесткими дисками, она опрокинулась, и произошел аварийный отказ. К счастью, никто не получил травм, но массивные жесткие диски разбились. Даже фальшпол проломился в том месте, где упала стойка. К счастью, отдел информационных технологий готов к таким событиям: каждый день ими делаются свежие резервные копии всех важных систем, и они быстро развернули новый сервер и восстановили базу данных.

Во время испытаний быстро обнаружили проблему: приложение связывает графические изображения с несколькими объектами базы данных, однако все изображения отсутствуют. Вы сразу же вызываете специалиста из ИТ-отдела.

«Мы восстановили базу данных и убедились в ее полноте на момент последнего резервного копирования», — сообщил технический специалист. «Где хранились изображения?»

Только сейчас вы вспомнили, что в данном приложении изображения хранились вне базы данных, а обычные файлы хранились в файловой системе. В базе данных хранится путь к изображению, и приложением открывается каждый файл изображения по указанному пути. «Изображения хранились как файлы. Они находились в файловой системе `/var`, также как и базы данных».

Специалист качает головой. «Мы не создаем резервные копии файлов в файловой системе `/var`, если нам не указывают конкретно, какие файлы надо резервировать. Мы, конечно же, делаем резервное копирование всех баз данных, а другие файлы в папке `/var` обычно являются всего лишь журналами, данными кэша или другими временными файлами. По умолчанию они не являются объектами резервного копирования».

У вас сердце уходит в пятки. В базе данных каталогов продуктов находилось свыше 11 000 изображений. Большинство их, возможно, существует в других местах, но их отслеживание, переформатирование и создание для них миниатюрных версий займет несколько недель.

12.1. ЦЕЛЬ: ХРАНЕНИЕ ИЗОБРАЖЕНИЙ И ДРУГИХ БОЛЬШЕРАЗМЕРНЫХ ФАЙЛОВ

В наше время в большинстве приложений используются изображения и другие виды информации. Иногда информация связана с объектами, хранящимися в базе данных. Например, пользователю разрешается хранение портрета или образа, который отображается при размещении комментария. Ошибкам в нашей базе данных ошибок часто необходим мгновенный снимок экрана, чтобы проиллюстрировать обстоятельства, в которых возникла неполадка.

Цель, описываемая в данной главе, состоит в хранении изображений и их связывании с объектами базы данных, такими как учетные записи пользователей или ошибки. При запросе этих объектов из базы данных требуется возможность извлекать связанные изображения в приложение.

12.2. АНТИПАТТЕРН: ПРЕДПОЛОЖЕНИЕ О НЕОБХОДИМОСТИ ИСПОЛЬЗОВАНИЯ ФАЙЛОВ

На концептуальном уровне изображение является атрибутом в таблице. Например, таблица `Accounts` может содержать столбец `portrait_image`.

Файл примера: *Phantom-Files/anti/create-accounts.sql*

```
CREATE TABLE Accounts (
   account_id       SERIAL PRIMARY KEY
   account_name     VARCHAR(20),
   portrait_image   BLOB
);
```

Подобным образом можно хранить несколько изображений одного типа в зависимой таблице. Например, у каждой ошибки может быть несколько мгновенных снимков экрана, которые поясняют возникшую ошибку.

Файл примера: *Phantom-Files/anti/create-screenshots.sql*

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

Приведенный выше фрагмент программного кода не представляется сложным, но выбор типа данных для изображения является предметом споров. Как показано выше, исходные двоичные данные изображения могут храниться в типе данных BLOB. Однако многие вместо этого хранят изображение как файл в файловой системе и путь к этому файлу хранят как VARCHAR.

Файл примера: *Phantom-Files/anti/create-screenshots-path.sql*

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

Разработчики программного обеспечения страстно обсуждают эту проблему. Существуют обоснованные причины для использования обоих решений, но для программистов характерно определенное стремление хранить файлы вне базы данных. Возможно, мое мнение будет непопулярным, но я собираюсь описать несколько реальных рисков, характерных для данной структуры, в следующих разделах.

Файлы не подчиняются удалению

Первая проблема заключается в сборе мусора. Если изображения находятся вне базы данных и вы удаляете строку, содержащую путь, отсутствует способ автоматического удаления файла, именуемого по этому пути.

Файл примера: *Phantom-Files/anti/delete.sql*

```
DELETE FROM Screenshots WHERE bug_id = 1234 and image_id = 1;
```

Если не разработать приложение так, чтобы при удалении строки базы данных удалялись и «осиротевшие» файлы изображений, на которые ссылается эта строка, «осиротевшие» файлы будут накапливаться.

Файлы не подчиняются локализации транзакции

Обычно, когда обновляют или удаляют данные, эти изменения не видны другим клиентам до тех пор, пока транзакция не будет завершена оператором COMMIT.

Однако все изменения, вносимые в файлы вне базы данных, не ведут себя подобным образом. Если удалить файл, он сразу становится недоступным другим клиентам. И если изменить содержимое файла, другие клиенты сразу увидят эти изменения вместо того, чтобы видеть предыдущее содержимое файла, пока транзакция не будет зафиксирована.

Файл примера: *Phantom-Files/anti/transaction.php*

```
<?php

$stmt = $pdo->query("DELETE FROM Screenshots
   WHERE bug_id = 1234 AND image_id =1");

unlink('images/screenshot1234-1.jpg');

// Другие клиенты по-прежнему видят строку в базе данных,

// но не файл изображения.

$pdo->commit();
```

На практике эти виды аномалий могут быть нечастыми. К тому же в данном примере отрицательный эффект минимальный: отсутствующее изображение — вряд ли редкий случай в веб-приложении. Но в других сценариях последствия могли бы быть серьезными.

Файлы не подчиняются операции отката

Обычное дело, когда откат транзакций выполняется в случае возникновения ошибок или даже если требуется отменить изменения в соответствии с логикой работы приложения.

Например, предположим, что при выполнении операции DELETE с целью удаления соответствующей строки в базе данных удаляется файл моментального снимка. Если выполнить откат данного изменения, удаление строки в базе данных отменяется, но файл по-прежнему будет отсутствовать.

Файл примера: *Phantom-Files/anti/rollback.php*

```
<?php

$stmt = $pdo->query("DELETE FROM Screenshots
   WHERE bug_id = 1234 AND image_id =1");

unlink("images/screenshot1234-1.jpg");

$pdo->rollback();
```

В базе данных восстанавливается строка, но не файл изображения.

Файлы не подчиняются инструментам резервного копирования базы данных

В большинстве баз данных клиенту предоставляется программное средство, облегчающее резервное копирование базы данных, находящейся в текущем пользовании. Например, в MySQL предоставляется `mysqldmp`, в Oracle — `rman`, в PostgreSQL — `pg_dump`, в SQLite предоставляется команда `.dump` и т. д. Применение средства резервного копирования важно, так как если другие клиенты вносят изменения параллельно, ваша резервная копия может содержать частичные изменения, потенциально нарушающие целостность на уровне ссылок или даже делающие резервную копию поврежденной или непригодной для восстановления.

Средству резервного копирования неизвестно, как включать файлы, упоминаемые по имени пути в столбце `VARCHAR` в таблице. Так что когда выполняется резервное копирование базы данных, необходимо помнить о двухэтапном процессе: использовании средства резервного копирования базы данных, а затем использовании средства резервного копирования файловой системы для сбора внешних файлов изображений.

Даже если включить внешние файлы с резервной копией, трудно обеспечить синхронизацию копий этих файлов с транзакцией, используемой для резервного копирования базы данных. Файлы изображений могут добавляться или изменяться приложением в любой момент времени, возможно, спустя лишь мгновение после начала резервного копирования базы данных.

Файлы не подчиняются правам доступа SQL

Внешними файлами обходятся любые права доступа, назначаемые с помощью SQL-операторов `GRANT` и `REVOKE`. С помощью SQL-прав управляют доступом к таблицам и столбцам, но они не применяются к внешним файлам, именуемым с помощью символьных строк в базе данных.

Файлы не являются типами данных SQL

Путь, хранящийся в `screenshot_path`, является просто символьной строкой. Базой данных не выполняется проверка того, что символьная строка представляет собой допустимое имя пути, и базой данных не может быть проверено существование файла по указанному пути. Если файл переименован, перемещен или удален, базой данных не обновляется автоматически символьная строка в базе данных. Любая логическая процедура, обрабатывающая эту символьную строку как имя пути, зависит от кода, написанного в приложении.

Файл примера: *Phantom-Files/anti/file-get.php*

```
<?php

define('DATA_DIRECTORY', '/var/bugtracker/data/');

$stmt = $pdo->query("SELECT image_path FROM Screenshots
   WHERE bug_id = 1234 AND image_id = 1");

$row = $stmt->fetch();

$image_path = $row[0];

// Чтение реального изображения -- Надеюсь, путь является правильным!

$image = file_get_contents(DATA_DIRECTORY . $image_path);
```

Преимущество использования базы данных заключается в том, что она помогает сохранять целостность данных. Когда некоторые данные помещают во внешние файлы, данное преимущество обходится стороной, и требуется написать прикладной код, чтобы выполнить проверки, которые должны осуществляться базой данных.

12.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Признаки данного антипаттерна нуждаются в небольшом анализе. Если по проекту существует какая-нибудь документация, предназначенная для администраторов программного обеспечения, или если есть возможность взять интервью у программистов, разрабатывающих программы (даже если это вы), найдите ответы на вопросы, подобные приведенным ниже.

- Что такое процедура резервного копирования и восстановления данных? Как может быть проверено резервное копирование? Проверяется ли восстановление данных на чистом сервере или на сервере, отличном от компьютера, на котором выполнялось резервное копирование?
- Накапливаются или удаляются изображения из системы, когда они устаревают? Какая процедура используется для их удаления? Выполняется ли эта процедура автоматически или вручную?
- Какие пользователи приложения обладают правом на просмотр изображений? Как вводятся в действие права пользователей? Что видят пользователи, если запрашивают разрешение увидеть изображения, на просмотр которых у них нет прав?

- Можно ли отменить изменение, внесенное в изображение? Если да, то должно ли предыдущее состояние изображения восстанавливаться приложением?

Для проектов, в которых присутствует антипаттерн, обычно не удается ответить на некоторые или все перечисленные выше вопросы. Не для всех приложений требуется надежное управление транзакциями или управление SQL-доступом к файлам изображений. Возможно, вы придете к выводу, что перевод базы данных в автономный режим во время резервных копирований является привлекательным компромиссом. Если ответы на эти вопросы неясные или неоткровенные, то это может свидетельствовать о небрежном использовании внешних данных в проекте.

12.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Существуют обоснованные причины хранения изображений и других больших объектов в файлах вне базы данных.

- Размер базы данных гораздо меньше без изображений, поскольку изображения обычно занимают много места по сравнению с простыми типами данных, такими как целые числа и символьные строки.
- Резервное копирование выполняется быстрее и получаемый результат меньше по размеру, если изображения не содержатся в базе данных. Изображения следует скопировать из файловой системы в качестве отдельного этапа резервного копирования, но эта процедура отличается большей управляемостью, чем резервное копирование огромной базы данных.
- Если изображения находятся в файлах вне базы данных, проще выполнять специальные операции предварительного просмотра и редактирования. Например, если требуется применить групповое изменение ко всем изображениям, то в этом случае особенно удобно хранить изображения вне базы данных.

Если перечисленные преимущества хранения изображений в файлах важны, и проблемы, описанные выше, не являются препятствиями, возможно, будет принято решение об использовании внешних файлов изображений в разрабатываемом проекте.

В некоторых базах данных поддерживаются специальные типы данных SQL, которые позволяют ссылаться на внешние файлы в более или менее явном виде. В Oracle этот тип данных называется `BFILE`, тогда как в SQL Server 2008 он называется `FILESTREAM`.

НЕ СЛЕДУЕТ ИСКЛЮЧАТЬ ИЗ РАССМОТРЕНИЯ НИКАКОЙ ИЗ ПРОЕКТОВ

В 1992 году я разработал в рамках договорного проекта приложение, которое позволяло сохранять изображения вне базы данных. Мой работодатель был нанят для разработки приложения регистрации на технической конференции. По прибытию участников конференции их снимали на видеокамеру, добавляли изображения в регистрационные записи и распечатывали на бейджиках.

Мое приложение было достаточно простым. Каждое изображение могло вставляться и обновляться только одним клиентским приложением (если человек мигал или ему не нравилось его фото, изображение можно было заменить во время регистрации). Отсутствовали требования по сложной обработке транзакций, одновременному доступу нескольких клиентов или выполнению отката. Нами не использовались права SQL-доступа. Предварительный просмотр изображений реализовывался проще, без необходимости выбирать фотографии из базы данных.

Я работал над этим проектом в то время, когда практические ограничения приложений и баз данных были гораздо жестче, чем те, которые существуют для сегодняшних технологий. При данных ограничениях имело бы смысл хранить изображения в наборе каталогов и управлять ими с помощью прикладного кода.

Необходимо планировать порядок использования приложением изображений, чтобы знать, будут ли возникать проблемы, описанные в разделе «Антипаттерн». Оптимальное решение всегда принимается на основе полной информации, а не общих рассуждений программистов, хранящих изображения во внешних файлах.

12.5. РЕШЕНИЕ: ИСПОЛЬЗОВАНИЕ ТИПОВ ДАННЫХ BLOB ПО МЕРЕ НЕОБХОДИМОСТИ

Если какие-либо из проблем, описанных в разделе «Антипаттерн» данной главы, присутствуют у вас, следует рассмотреть возможность хранения изображений внутри базы данных, а не во внешних файлах. Базы данных всех марок поддерживают тип данных BLOB, который можно использовать для хранения любых двоичных данных.

Файл примера: *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)
);
```

Если хранить изображение таким способом в столбце BLOB, все проблемы решаются.

- Данные изображений хранятся в базе данных. Отсутствует дополнительная операция по их загрузке. Отсутствует риск ошибки в имени пути к файлу.
- При удалении строки изображение удаляется автоматически.
- Изменения, вносимые в изображение, не видны другим клиентам до тех пор, пока изменение не будет зафиксировано.
- При откате транзакции восстанавливается предыдущее состояние изображения.
- При обновлении строки создается блокировка, так что никакой другой клиент не сможет одновременно обновить то же изображение.
- Резервные копии базы данных содержат все изображения.
- SQL-права управляют доступом как к изображению, так и к строке.

Максимальный размер для типа данных BLOB варьируется в зависимости от модели базы данных, но его достаточно для хранения большинства изображений. Всеми базами данных должны поддерживаться BLOB либо какой-нибудь родственный ему тип данных. Например, базой данных MySQL предоставляется тип данных MEDIUMBLOB, который позволяет сохранять до 16 мегабайт, которых достаточно для большинства изображений. Базой данных Oracle поддерживаются типы данных LONG RAW или BLOB, предоставляющие возможность хранения до 2 или 4 гигабайт, соответственно. Аналогичные типы данных доступны в базах данных других марок.

Во-первых, изображения обычно существуют в файлах, поэтому требуется некоторый способ их загрузки в столбец BLOB в базе данных. В ряде баз данных предусмотрены функции для загрузки внешних файлов. Например, в базе данных MySQL существует функция LOAD_FILE(), которая позволяет считывать файл обычно с целью сохранения содержимого в столбце BLOB.

Файл примера: *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;
```

Можно также сохранить содержимое столбца BLOB в файл. Например, в базе данных MySQL существует дополнительное предложение оператора

SELECT, позволяющее точно сохранять результат запроса без какого-либо форматирования для обозначения окончания столбца или строки.

Файл примера: *Phantom-Files/soln/dumpfile.sql*

```
SELECT screenshot_image
INTO DUMPFILE 'images/screenshot1234-1.jpg'
FROM Screenshots
WHERE bug_id = 1234 AND image_id =1;
```

Можно также выбрать информацию изображений из BLOB и вывести ее напрямую. В веб-приложении можно вывести двоичное содержимое, такое как изображение, но необходимо установить надлежащим образом тип содержимого.

Файл примера: *Phantom-Files/soln/binary-content.php*

```
<?php

header('Content-type: image/jpg');

$stmt = $pdo->query("SELECT screenshot_image FROM Screenshots
   WHERE bug_id = 1234 AND image_id = 1");
$row = $stmt->fetch();

print $row[0];
```

Всякий раз, когда какой-либо результат ищется аппаратом, возникает вопрос — какой процедурой расчетов могут быть получены машиной эти результаты за кратчайшее время.

Чарльз Беббидж,
«Эпизоды из жизни философа» (1864)

ГЛАВА 13. БЕСПОРЯДОЧНОЕ СОЗДАНИЕ ИНДЕКСОВ

«Эй! Есть минута? Хотел бы воспользоваться вашей помощью», — голос по телефону перекрывал шум вентиляции в центре обработки данных. Это был ведущий администратор базы данных вашей компании.

«Конечно», — был ваш ответ, с некоторой неуверенностью относительно того, что от вас может потребоваться.

«Дело в том, что здесь есть база данных, которая большей частью переведена на сервер», — продолжает администратор базы данных. «Я сходил туда, взглянул и обнаружил проблему. В некоторых таблицах полностью отсутствуют индексы, а в ряде других таблиц присутствуют все мыслимые индексы. Мы должны решить эту проблему или отдать весь сервер вам, так как ни у кого нет времени!»

«Извините — в действительности я не знаю так много о базах данных», — отвечаете вы, пытаясь успокоить администратора. — Мы сделаем все возможное, чтобы провести оптимизацию, но, очевидно, это та задача, которую может сделать такой эксперт, как вы. Нет ли там настроек базы данных, которые вы могли бы выполнить?»

«Парень, я настроил все, что мог; вот почему мы вообще-то все еще работаем здесь», — отвечает администратор. «Единственная возможность, оставшаяся у нас, — остановить твое приложение, и я не думаю, что ты хочешь этого. Мы прекратим гадать и начнем получать некоторые ответы на то, что требуется твоему приложению от базы данных».

Можно сказать, все это свалилось на вашу голову. Вы осторожно спрашиваете: «Что вы имеете в виду? Я сказал вам, что в нашей группе нет эксперта по базе данных».

«Это не проблема», — рассмеялся администратор базы данных. «Вы знаете свое приложение, не так ли? Это часть проблемы, которая важна и с которой я *не могу* помочь. У меня есть парень, который вооружит вас необходимыми инструментами, и затем мы ликвидируем ваше узкое место. Вам потребуется лишь небольшое обучение. Вот увидите».

13.1. ЦЕЛЬ: ОПТИМИЗАЦИЯ ПРОИЗВОДИТЕЛЬНОСТИ

Производительность — единственная распространенная проблема, о которой я слышал от разработчиков баз данных. Просто ознакомьтесь с разговорами, которые ведутся на любой технической конференции: в них нескончаемо говорят об инструментах и методах, позволяющих выжать как можно больший результат из базы данных. Когда я завожу разговор о способе структуризации базы данных или написании SQL для повышения надежности, безопасности и достоверности, я не удивляюсь, что из зала звучит единственный вопрос: «Хорошо, но как это повлияет на производительность?»

Лучший способ повышения производительности в базе данных состоит в оптимальном применении индексов. Индекс — это структура данных, используемая базой данных для корреляции значений со строками, в которых встречаются эти значения в заданном столбце. Индекс обеспечивает простой способ более быстрого обнаружения в базе данных значений по сравнению с прямолинейным методом поиска во всей таблице сверху вниз.

Разработчики программного обеспечения обычно не понимают, как и когда следует пользоваться индексами. В документации и книгах по базам данных отсутствуют или редко приводятся четкие инструкции о том, когда следует пользоваться индексами. Разработчики могут лишь догадываться о способах эффективного применения индексов.

13.2. АНТИПАТТЕРН: ИСПОЛЬЗОВАНИЕ ИНДЕКСОВ БЕЗ КАКОГО-ЛИБО ПЛАНА

Когда индексы выбирают по наитию, неизбежно делают неправильный выбор. Непонимание того, когда следует использовать индексы, ведет к ошибкам, относящимся к одной из трех категорий:

- отсутствие определения индексов или определение недостаточного количества индексов;
- определение избыточного количества индексов или бесполезных индексов;
- выполнение запросов, в которых индексы не могут помочь.

Отсутствие индексов

Часто мы читаем, что при обновлении базой данных индексов ею добавляются дополнительные служебные данные. Всякий раз, когда используют операторы `INSERT`, `UPDATE` или `DELETE`, базой данных обновляются структуры данных индекса с целью получения согласованной таблицы, чтобы последующие операции поиска использовали эти индексы для достоверного

нахождения правильного набора строк.

Многие привыкли думать, что служебные данные приводят к различным потерям. Поэтому когда мы читаем, что базой данных используется дополнительная служебная информация для поддержания индексов обновленными, мы хотим исключить эту служебную информацию. Некоторые разработчики делают вывод, что спасительным средством является избавление от индексов. Данный совет распространен, но им игнорируется тот факт, что индексы обладают преимуществами, которые оправдывают появление дополнительной служебной информации.

ИНДЕКСЫ НЕ ЯВЛЯЮТСЯ СТАНДАРТНЫМИ

Известно ли вам, что в ANSI-стандарте по языку SQL ничего не говорится об индексах? Реализация и оптимизация хранения данных не определяется языком SQL, так что в базе данных любого производителя предоставляется свобода реализации индексов.

В большинстве баз данных различных марок присутствует похожий синтаксис `CREATE INDEX`, однако в базе данных конкретного производителя предоставляется гибкий подход к рационализации и добавлению своих собственных технологий. Возможности индексов не стандартизованы. Аналогичным образом, отсутствует стандарт на обслуживание индексов, автоматическую оптимизацию запросов, отчет о плане запросов и команды, подобные `EXPLAIN`.

Чтобы извлечь максимальную пользу из индексов, необходимо изучить документацию по используемой базе данных. Конкретный синтаксис и функции индексов варьируются в широком диапазоне, но логические концепции применимы в равной степени к базам данных всех марок.

Не всякая дополнительная служебная информация означает потери в производительности или эффективности. Работают ли в вашей компании управленческий персонал, юристы, бухгалтеры, и приходится ли вам платить за коммунальные услуги, даже если эти расходы напрямую не приносят прибыли? Да, поскольку эти люди вносят большой вклад в успех компании по важным направлениям.

В типовом приложении выполняют сотни запросов в таблице для каждого одного обновления. Каждый раз, когда выполняется запрос, в котором используются индексы, компенсируются издержки, связанные с обслуживанием этого индекса.

Индекс может также быть полезным для оператора `UPDATE` или `DELETE`, обеспечивая быстрый поиск требуемых строк. Например, индекс по первичному ключу `bug_id` помогает выполнению следующего оператора:

Файл примера: *Index-Shotgun/anti/update.sql*

```
UPDATE Bugs SET status = 'FIXED' WHERE bug_id = 1234;
```

Оператору, с помощью которого выполняется поиск неиндексированного столбца, требуется выполнить полное сканирование таблицы, чтобы найти совпадающие строки.

Файл примера: *Index-Shotgun/anti/update-unindexed.sql*

```
UPDATE Bugs SET status = 'OBSOLETE' WHERE date_reported <
'2000-01-01';
```

Избыточные индексы

Пользу от индекса извлекают только в том случае, если выполняют запросы, использующие этот индекс. Нет никакого смысла в создании индексов, которые не применяются. Ниже приводится несколько примеров:

Файл примера: *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),
① INDEX (bug_id),
② INDEX (summary),
③ INDEX (hours),
④ INDEX (bug_id, date_reported, status)
);
```

В предыдущем примере присутствует несколько бесполезных индексов:

① `bug_id`: в большинстве баз данных индекс для первичного ключа создается автоматически, поэтому излишне определять еще один индекс. Он не обеспечивает преимуществ и может лишь приводить к дополнительным служебным данным. В базе данных каждой модели существуют свои собственные правила, определяющие, когда должен автоматически создаваться индекс. Рекомендуется прочитать документацию на используемую базу данных;

② `summary`: индексирование для длинного строкового типа данных, такого как `VARCHAR(80)`, больше, чем индекс для более компактного типа данных. К тому же вряд ли будут выполнять запросы, которые осуществляют поиск или сортировку по всему столбцу `summary`;

③ `hours`: это еще один пример столбца, в котором, вероятно, не будут производить поиск конкретных значений;

④ `bug_id`, `date_reported`, `status`: существуют веские причины использовать комбинированные индексы, но многие люди создают комбинированные индексы, которые являются излишними или редко применяются. Кроме того, важен порядок расположения столбцов в комбинированном индексе; следует использовать столбцы слева направо в условии поиска, критериях объединения или в порядке сортировки.

ХЕДЖИРОВАНИЕ СТАВОК

Билл Косби (Bill Cosby) рассказал историю о своем отпуске в Лас-Вегасе. Он был так расстроен проигрышем в казино, что решил все-таки что-нибудь выиграть — один раз — до того как покинет город. Так что он купил на 200 долларов 25-центовых фишек, подошел к столу с рулеткой и поставил фишки на каждую клетку, красную и черную. **Он покрыл весь стол**. Крупье крутанул шарик...и он упал на пол.

Некоторые создают индексы по всем столбцам — и по всем комбинациям столбцов, — так как они не знают, от каких индексов выиграют их запросы. Если покрыть весь «стол» базы данных индексами, появится дополнительно много служебных данных без гарантии какой-либо отдачи от них.

Когда отсутствие индексов помогает

Следующий тип ошибки заключается в выполнении запроса, которым не может использоваться какой-либо индекс. Разработчики создают все больше и больше индексов, пытаясь найти некоторую магическую комбинацию столбцов или параметров индекса, чтобы сделать свои запросы более быстрыми в выполнении.

Индекс базы данных можно представить, используя аналогию с телефонной книгой. Если попросить найти в телефонной книге всех, у кого *фамилия* Чарльз (Charles), то это будет простая задача. Все люди с одинаковой фамилией перечисляются вместе, поскольку именно так организована телефонная книга.

Однако если попросить найти в телефонной книге всех, у кого *имя* Чарльз (Charles), то порядок расположения имен в телефонной книге не облегчит решение этой задачи. У любого человека может быть такое имя, независимо от его фамилии, поэтому потребуется выполнить поиск по всей книге, строчка за строчкой.

Телефонная книга упорядочена по фамилиям, а затем по именам, как и комбинированный индекс по индексам `last_name`, `first_name`. Данный индекс не помогает выполнять поиск по имени.

Файл примера: *Index-Shotgun/anti/create-index.sql*

```
CREATE INDEX TelephoneBook ON Accounts(last_name, first_name);
```

Некоторые примеры запросов, не получающие преимуществ от данного индекса:

- `SELECT * FROM Accounts ORDER BY first_name, last_name;`

Данным запросом демонстрируется сценарий телефонной книги. Если создается комбинированный индекс для столбцов `last_name`, за которыми следуют `first_name` (как в телефонной книге), индекс не помогает первичной сортировке по значению `first_name`.

- `SELECT * FROM Bugs WHERE MONTH(date_reported) = 4;`

Даже если создать индекс для столбца `date_reported`, порядок расположения индекса не помогает выполнять поиск по месяцу. Порядок данного индекса основывается на всей дате, начинающейся с года. Но в каждом годе есть четвертый месяц, так что строки, в которых месяц равен 4, разбросаны по всей таблице.

Некоторыми базами данных поддерживаются индексы по выражениям или индексы по генерируемым столбцам, а также индексы по простым столбцам. Но прежде чем пользоваться индексом, его необходимо определить, и этот индекс помогает только для выражения, указываемого в его определении.

- `SELECT * FROM Bugs WHERE last_name = '`*`Charles`*`' OR first_name = '`*`Charles`*`';`

Мы вернулись к проблеме, когда строки с указанным конкретным именем разбросаны непредсказуемо относительно порядка индекса, определенного нами. Результат предыдущего запроса такой же, как и результат следующего запроса:

```
SELECT * FROM Bugs WHERE last_name = 'Charles'
UNION
SELECT * FROM Bugs WHERE first_name = 'Charles';
```

Индекс в нашем примере помогает найти фамилию, но он бесполезен при поиске требуемого имени.

- `SELECT * FROM Bugs WHERE description LIKE '%`*`crash`*`%';`

Так как шаблон в предикате данного поиска может присутствовать в любом месте символьной строки, структура данных отсортированного индекса никак не может помочь.

13.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Ниже перечисляются признаки антипаттерна **Беспорядочное создание индексов**.

- «Вот мой запрос; как можно ускорить его выполнение?»

Возможно, это единственный самый распространенный вопрос по SQL, но в нем отсутствуют сведения об описании таблицы, индексах, объеме данных, а также об измерениях производительности и оптимизации. Без этого контекста любой ответ будет лишь догадкой.

- «Я определил индекс по всем полям; почему запрос не выполняется быстрее?»

Это классическое антирешение **Беспорядочного создания индексов**. Вы опробовали все возможные индексы — но занимались беспорядочной «пальбой в темноту».

- «Я читал, что индексы замедляют работу в базе данных, поэтому я не использую их».

Подобно многим разработчикам, вы ищете на все случаи жизни одну стратегию повышения производительности. Такого всеохватывающего метода не существует.

ИНДЕКСЫ С НИЗКОЙ СЕЛЕКТИВНОСТЬЮ

Селективность — это статистика об индексе базы данных. Это отношение количества отдельных значений в индексе к общему количеству строк в таблице.

```
SELECT COUNT(DISTINCT status) /
COUNT(status) AS selectivity FROM Bugs;
```

Чем меньше коэффициент селективности, тем ниже эффективность индекса. Почему так происходит? Давайте рассмотрим аналогию.

В данной книге содержится индекс разного типа: в каждой записи индекса книги перечисляются страницы, на которых присутствует слова записи. Если слово часто встречается в книге, в индексе может перечисляться много номеров страниц. Чтобы найти искомую часть книги, следует пройтись по всем страницам в списке.

Индексы не должны содержать слова, которые встречаются на слишком большом количестве страниц. Если приходится часто листать страницы вперед и назад, переходя от индексов на страницы книги, тогда, возможно, потребуется прочитать всю книгу от корки до корки.

Аналогичным образом, если в индексе базы данных заданное значение встречается во многих строках в таблице, больше трудов уйдет на чтение индекса, чем на простое сканирование всей таблицы. На самом деле в этих случаях использование индекса может быть действительно связано с большими затратами.

В идеальном варианте базой данных отслеживается селективность индексов, и ею не должен использоваться индекс, который не дает никаких преимуществ.

13.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Если требуется разработать базу данных общего назначения, когда неизвестно, какие запросы важно оптимизировать, нельзя быть уверенным, что какие-то конкретные индексы будут самыми лучшими. Необходимо делать информационно обоснованные предположения. Вполне вероятно, что некоторые индексы, которые предоставляют преимущества, будут пропущены. Также правдоподобно, что будут созданы некоторые индексы, которые окажутся невостребованными. Но необходимо сделать как можно более точные предположения.

13.5. РЕШЕНИЕ: ИСПОЛЬЗОВАНИЕ ПРОЦЕДУРЫ MENTOR В ОТНОШЕНИИ ИНДЕКСОВ

Антипаттерн **Беспорядочное создание индексов** касается создания или пропуска индексов без причин, так что давайте предлагать способы анализа базы данных и находить обоснованные причины для включения индексов или их пропуска.

НЕ ВСЕГДА БАЗА ДАННЫХ ЯВЛЯЕТСЯ УЗКИМ МЕСТОМ

В сообществе разработчиков программного обеспечения распространено мнение, что база данных всегда является самой медленной частью приложения и источником проблем с производительностью. Однако это неправда.

Например, в приложении, над которым я работал, мой менеджер попросил меня выяснить, почему оно функционировало так медленно, причем он настаивал, что это был дефект базы данных. После того как я воспользовался средством профилирования для измерения производительности кода приложения, было найдено, что 80 процентов времени работы средства ушло на синтаксический анализ его собственных выходных HTML-данных с целью поиска полей формы для занесения в них значений. Проблема производительности никоим образом не была связана с запросами в базе данных.

Прежде чем делать предположения об узких местах, являющихся причиной снижения производительности, используйте программные диагностические средства для проведения измерений. В противном случае, возможно, придется заниматься непродуманной оптимизацией.

Для описания контрольного списка действий по анализу базы данных на правильность выбора индексов можно использовать процедуру *MENTOR* (*Measure*, *Explain*, *Nominate*, *Test*, *Optimize* и *Rebuild* — *измерение, объяснение, номинирование, тестирование, оптимизация и перекомпоновка*).

Измерение

Невозможно принимать информированные решения без информации. В большинстве баз данных предоставляется тот или иной способ регистрации времени выполнения SQL-запросов, так чтобы идентифицировать самые дорогостоящие операции. Приведем примеры:

- И в Microsoft SQL Server, и в Oracle существуют средства и инструменты *трассировки SQL* для создания отчетов и анализа результатов трассировки. В Microsoft этот инструмент называют *SQL Server Profiler*, а в Oracle этот инструмент называют *TKProf*.
- В базах данных MySQL и PostgreSQL могут регистрировать запросы, время выполнения которых превышает заданный порог. В базе данных MySQL это *журнал медленных запросов* (*slow query log*), и его параметр конфигурации `long_query_time` по умолчанию установлен равным 10 секундам. В PostgreSQL существует аналогичная переменная конфигурации `log_min_duration_statement`.

В PostgreSQL также существует сопутствующее средство, называемое *pgFouine*, которое помогает анализировать журнал запросов и определять запросы, требующие особого внимания (**pgfouine.projects.postgresql.org/**).

Узнав, какие запросы выполняются дольше всего в приложении, можно понять, в каком месте следует сосредоточить усилия по оптимизации для получения максимального эффекта. Возможно, будет обнаружено, что все запросы работают эффективно за исключением одного запроса, являющегося узким местом. С этого запроса и следует начать оптимизацию.

Область наибольшей стоимости в приложении не обязательно является самым продолжительным по времени запросом, если этот запрос выполняется от случая к случаю. Другие, более простые запросы могут выполняться часто, гораздо чаще, чем можно было бы ожидать, поэтому на них приходится больше суммарного времени. Если уделить внимание оптимизации этих запросов, то получится больший экономический эффект.

Отключите все кэширование результатов запросов во время измерения производительности запросов. Данный тип кэша предназначен для обхода выполнения запроса и использования индексов, так что он не обеспечивает точного измерения.

Более точную информацию можно получить путем профилирования приложения после его развертывания. Соберите совокупные показатели о том, что выполняется программным кодом, когда реальные пользователи работают с приложением и когда задействована реальная база данных. Время от времени следует отслеживать данные профилирования, чтобы быть уверенным в том, что не возникло новое узкое место.

Не забудьте отключить или уменьшить скорость генерирования отчетов профайлерами после выполнения измерения, так как этими инструментами вносятся дополнительные служебные данные.

Объяснение

После того как определен запрос, имеющий наибольшую стоимость, на следующем шаге выясняют, почему он выполняется медленно. В каждой базе данных применяется оптимизатор с целью выбора индексов для запроса. Можно заставить базу данных выводить отчет по ее анализу, называемый *планом выполнения запроса* (QEP — query execution plan).

Синтаксис запроса QEP варьируется в зависимости от модели базы данных:

Модель базы данных	Решение по генерированию отчета 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

Не существует стандарта, определяющего, какие сведения должны содержаться в отчете QEP и формат отчета. В общем случае в QEP показывается, какие таблицы задействованы в запросе, способ использования индексов оптимизатором и порядок, в котором осуществляется доступ к таблицам. Отчет может также содержать статистические данные, такие как количество строк, генерируемых на каждом этапе запроса.

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

Рис. 13.1. План выполнения запроса в MySQL

Давайте взглянем на пример SQL-запроса и запроса отчета QEP:

Файл примера: *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;
```

В QEP-отчете MySQL, приведенном на рис. 13.1, столбец `key` показывает, что данным запросом используется только таблица BugsProducts, индекс первичного ключа. Кроме того, дополнительные примечания в последнем столбце показывают, что запросом сортируются результаты во временной таблице без использования преимуществ индекса.

Оператором `LIKE` принудительно выполняется полное сканирование таблицы в `Bugs`, а по `Products.product_name` отсутствует индекс. Данный запрос можно улучшить, если создать новый индекс по product_name, а также использовать решение по полнотекстовому поиску[1].

Сведения в QEP-отчете варьируются в зависимости от поставщика базы данных. В данном примере следует прочитать страницу руководства по MySQL, «Optimizing Queries with EXPLAIN», чтобы разобраться, как интерпретировать сведения отчета[2].

Номинирование

Теперь, когда для запроса есть QEP-отчет оптимизатора, следует найти случаи, в которых запросом выполняется обращение к таблице без использования индекса.

ПОКРЫВАЮЩИЕ ИНДЕКСЫ

Если индексом предоставляются все требуемые столбцы, тогда вообще не надо считывать строки данных из таблицы.

Представьте, если записи телефонной книги содержали бы только номер страницы; после того как найдена фамилия, потребовалось бы перейти на страницу, на которую указывает ссылка, чтобы узнать фактический номер телефона. Более рационально искать информацию за один шаг. Поиск фамилии выполняется бы-

[1] См. главу 17.

[2] **dev.mysql.com/doc/refman/5.1/en/using-explain.html.**

стро, так как записи в книге упорядочены, и прямо на месте можно получить другие атрибуты, необходимые для данной записи, такие как номер телефона и, возможно, адрес.

Вот как работает **покрывающий индекс.** Можно определить индекс для включения дополнительных столбцов, даже если они в других обстоятельствах не нужны для индекса.

```
CREATE INDEX BugCovering ON Bugs
(status, bug_id, date_reported, reported_by, summary);
```

Если запрос ссылается только на столбцы, включенные в структуру данных индекса, базой данных генерируются результаты запроса путем чтения только индекса.

```
SELECT status, bug_id, date_reported, summary
FROM Bugs WHERE status = 'OPEN';
```

Базе данных не требуется считывать соответствующие строки из этой таблицы. Покрывающие индексы нельзя использовать для каждого запроса, но при возможности их применение обычно дает большой выигрыш в производительности.

В некоторых базах данных существуют инструменты, автоматически собирающие статистику трассировки запросов и предлагающие ряд изменений, включая создание новых индексов, которые пропущены вами, но от которых запрос получает преимущества. Например:

- Design Advisor в СУБД IBM DB2;
- Database Engine Tuning Advisor в СУБД Microsoft SQL Server;
- Enterprise Query Analyzer в СУБД MySQL;
- Automatic SQL Tuning Advisor в СУБД Oracle.

Даже не пользуясь автоматическими советниками, можно изучить, как узнать, когда запрос получает выигрыш от индекса. Чтобы интерпретировать показатели QEP-отчета, необходимо проштудировать документацию по базе данных.

Тестирование

Данный шаг важен: после создания индексов повторно выполните профилирование запросов. Важно убедиться, что внесенное изменение приводит к другому результату, чтобы быть уверенным в правильности выполненной работы.

Можно также использовать этот шаг, чтобы произвести впечатление на начальника и оправдать работу по выполненной оптимизации. Вы не хотите, чтобы ваше еженедельное состояние было подобно следующему: «Я опробовал все мыслимые варианты, чтобы устранить проблемы с производительностью, и теперь следует подождать и убедиться...» Вместо этого должна быть возможность предоставить следующий доклад: «Я определил, что

мы могли бы создать один новый индекс по активно используемой таблице и мною улучшена производительность наших критических запросов на 38 процентов.»

Оптимизация

Индексы — это компактные, часто используемые структуры данных, которые легко хранить в кэш-памяти. Чтение индексов из памяти повышает производительность на порядок по сравнению с их чтением с дискового устройства ввода-вывода.

Серверы баз данных позволяют настраивать объем системной памяти, выделяемой для кэширования. В большинстве баз данных размер буфера кэша устанавливается достаточно низким, чтобы обеспечить работу базы данных на различных системах. Может возникнуть желание увеличить размер кэша.

Как много памяти следует выделить для кэширования? На этот вопрос нет единого ответа, так как он зависит от размера базы данных и размера доступной системной памяти.

Выигрыш можно также получить от предварительной загрузки индексов в кэш-память вместо того, чтобы полагаться на действия базы данных по занесению в кэш наиболее часто используемых данных или индексов. Например, в базе данных MySQL используйте оператор `LOAD INDEX INTO CACHE`.

Перекомпоновка

Индексы обеспечивают наибольшую эффективность, когда они *сбалансированы*. Со временем, по мере обновления и удаления строк, индексы могут постепенно становиться несбалансированными, подобно тому как со временем происходит фрагментация файловых систем. На практике, возможно, не будет видна большая разница между оптимальным индексом и несбалансированным индексом. Но поскольку хотелось бы извлечь максимальную пользу от индексов, имеет смысл проводить текущее обслуживание базы данных на регулярной основе.

Как и в случае большинства функций, относящихся к индексам, в базе данных каждой модели применяются терминология, синтаксис и возможности, определяемые разработчиком.

Модель базы данных	Команда обслуживания индексов
IBM DB2	`REBUILD INDEX`
Microsoft SQL Server	`ALTER INDEX ... REORGANIZE`, `ALTER INDEX ... REBUILD` или `DBCC DBREINDEX`

Окончание табл.

Модель базы данных	Команда обслуживания индексов
MySQL	`ANALYZE TABLE` или `OPTIMIZE TABLE`
Oracle	`ALTER INDEX ... REBUILD`
PostgreSQL	`VACUUM` или `ANALYZE`
SQLite	`VACUUM`

Как часто следует выполнять перекомпоновку индекса? Можно услышать ответы общего характера, например «раз в неделю», но, по правде говоря, не существует универсального ответа, который бы подходил для всех приложений. Он зависит от того, как часто вводятся изменения в заданную таблицу, которыми может вноситься дисбаланс. Ответ также зависит от размера таблицы и ее важности для получения оптимальных преимуществ от индексов для этой таблицы. Стоит ли тратить часы на перекомпоновку большой, но редко используемой таблицы, если можно ожидать повышения производительности на какой-нибудь 1 процент? Кому как не вам лучше судить об этом, так как вы лучше других знаете свои данные и требования к операциям.

Немало сведений об извлечении максимальной пользы от индексов зависит от конкретного разработчика, поэтому необходимо выполнить поиск информации по используемой модели базы данных. Доступные вам ресурсы включают руководство по базе данных, книги и журналы, блоги и списки почтовых рассылок и, кроме того, большой объем собственных экспериментов. Самое важное правило — гадание вслепую относительно индексирования не является оптимальной стратегией.

ВНИМАНИЕ!

Знайте свои данные, знайте свои запросы и выполняйте процедуру MENTOR в отношении индексов.

ЧАСТЬ III. АНТИПАТТЕРНЫ ЗАПРОСОВ

Есть то, о чем мы знаем, что мы это знаем. Есть то, о чем знаем, что мы это не знаем. Но есть также и то, про что мы не знаем, что мы этого не знаем.

Дональд Рамсфелд

ГЛАВА 14. БОЯЗНЬ НЕИЗВЕСТНОГО

В примере базы данных ошибок таблица `Accounts` имеет столбцы `first_name` и `last_name`. Используя оператор конкатенации, вы можете объединить столбцы и форматировать полное имя пользователя в одном столбце:

Файл примера: *Fear-Unknown/intro/full-name.sql*

```
SELECT first_name || ' ' || last_name AS full_name FROM Accounts;
```

Представьте, что ваш начальник попросил изменить базу данных, чтобы добавить отчество пользователя в таблицу (у нескольких пользователей могут быть одинаковые имя и фамилия, добавление отчества — хороший выход из положения). Это довольно простое изменение. Вы также вручную добавляете отчество для нескольких пользователей.

Файл примера: *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;
```

Внезапно приложение прекращает показывать имена. При более детальном изучении вы понимаете, что таблица не универсальна. Отображаются только имена, у которых отмечено отчество; все остальные — не отображаются.

Что случилось со всеми остальными именами? Сможете ли вы изменить это до того, как ваш начальник заметит и начнет паниковать, решив, что вы потеряли данные в базе данных?

14.1. ЦЕЛЬ: РАСПОЗНАВАНИЕ ОТСУТСТВУЮЩИХ ЗНАЧЕНИЙ

В вашей базе данных некоторые данные неизбежно не будут иметь значений. Также вам может понадобиться вставить строку, прежде чем вы установите значения для всех столбцов, или же некоторые столбцы не будут иметь значений при определенных оправданных обстоятельствах. SQL поддерживает специальное пустое значение, соответствующее ключевому слову NULL.

Существует множество вариантов продуктивного использования значения NULL в таблицах SQL и запросах:

- Вы можете использовать значение NULL вместо значения, которое недоступно во время создания строки, например, дата прекращения работы служащего, который еще работает.
- В данном столбце может использоваться значение NULL, когда не существует какого-либо значения для данной строки, например, оценка эффективности расхода топлива для полностью электрической машины.
- Значение NULL может вернуться, если введено неверное значение, например, DAY('2009-12-32').
- Во внешнем соединении используется значение NULL в качестве указателя места заполнения для столбцов несогласованной таблицы.

Цель состоит в том, чтобы ввести запросы для столбцов, содержащих значения NULL.

14.2. АНТИПАТТЕРН: ИСПОЛЬЗОВАНИЕ NULL КАК ОБЫЧНОГО ЗНАЧЕНИЯ ИЛИ НАОБОРОТ

Многие разработчики программного обеспечения беззащитны перед поведением значения NULL в SQL. В отличие от большинства языков программирования, в SQL значение NULL обрабатывается как специальное значение, отличное от нуля, лжи или пустой строки. Это истинно для стандартного языка SQL и большинства торговых марок баз данных. Однако в таких СУБД, как Oracle и Sybase, значение NULL равнозначно строке нулевой длины. Значение NULL в них также имеет иное поведение.

Использование значения NULL в выражении

Многих удивляет, когда производятся арифметические действия в столбце или выражении, имеющих значение NULL. Например, многие программисты ожидают, что результатом выражения будет значение 10, если не была произведена оценка в столбце hours, однако в результате получают значение NULL.

Файл примера: *Fear-Unknown/anti/expression.sql*

```
SELECT hours + 10 FROM Bugs;
```

Значение NULL не равносильно нулю. Неизвестное, к которому прибавили 10, остается неизвестным.

Значение NULL — это не то же самое, что строка нулевой длины. Объединение строк, содержащих значение NULL в стандартном языке SQL, возвращает значение NULL (несмотря на специфическое поведение в Oracle и Sybase).

Значение NULL — это не ложное значение. Булевы выражения с операторами AND, OR и NOT также приводят к результатам, которые многие люди считают запутывающими.

Поиск столбцов, способных иметь значение NULL

Показанный ниже запрос выводит только строки, в которых значения столбца assigned_to равны 123, не отображая другие значения или строки, в которых данный столбец имеет значение NULL.

Файл примера: *Fear-Unknown/anti/search.sql*

```
SELECT * FROM Bugs WHERE assigned_to = 123;
```

Вы, возможно, предположите, что следующий запрос отображает оставшиеся строки, *не* выведенные при первом запросе:

Файл примера: *Fear-Unknown/anti/search-not.sql*

```
SELECT * FROM Bugs WHERE NOT (assigned_to = 123);
```

Однако ни один результат запроса не содержит строки, в которых столбец assigned_to имеет значение NULL. Результатом сравнения со значением NULL всегда будет неизвестность, а не истина или ложь. Даже отрицание значения NULL все равно даст в результате NULL.

При поиске значений NULL или всех, кроме значений NULL, распространены следующие ошибки.

Файл примера: *Fear-Unknown/anti/equals-null.sql*

```
SELECT * FROM Bugs WHERE assigned_to = NULL;

SELECT * FROM Bugs WHERE assigned_to <> NULL;
```

Условие оператора WHERE удовлетворяется только, когда выражение является истиной, однако в случае сравнения со значением NULL выражение не истинно; оно является неизвестным. Не важно, идет ли речь о равенстве или неравенстве, результат все так же остается неизвестным, что, конечно, не является истиной. Ни один из предыдущих запросов не вернет строки, в которых значение столбца assigned_to — NULL.

Использование значения NULL в параметрах запроса

Также трудно использовать значение NULL в параметризуемом выражении SQL так, будто оно является обычным значением.

Файл примера: *Fear-Unknown/anti/parameter.sql*

```
SELECT * FROM Bugs WHERE assigned_to = ?;
```

Указанный выше запрос возвращает предсказуемые результаты, когда вы вводите обычное целочисленное значение в качестве параметра, однако вы не можете использовать введенное вручную слово NULL в качестве параметра.

Выход из ситуации

Если манипуляции со значением NULL делают запросы более сложными, многие разработчики программного обеспечения отказываются от использования NULL в базах данных. Вместо этого они выбирают обычные значения, чтобы обозначить «неизвестность» или «неприменимость».

«МЫ НЕНАВИДИМ NULL!»

Джек, разработчик программного обеспечения, по требованию своего клиента отказался от использования значения NULL в своей базе данных. Объяснение было простым: «Мы ненавидим NULL!» — присутствие значения NULL привело бы к ошибкам в прикладном коде. Джек спросил, какое еще значение он может использовать, чтобы заменить отсутствующее значение.

Я сказал Джеку, что замена отсутствующего значения — это прямая цель значения NULL. Независимо от того, какое значение он выбрал бы для замены отсутствующего, ему пришлось бы изменить прикладной код, чтобы обработать это значение как специальное.

Отношение клиента Джека к значению NULL неправильно; аналогично я мог бы сказать, что мне не нравится писать код, предотвращающий ошибки, связанные с делением на ноль, но из-за этого было бы плохим выбором запретить все операции с нулем.

Что именно является неправильным в данной практике? В следующем примере столбцам assigned_to и hours, способным иметь значения NULL, присваивается значение NOT NULL.

Файл примера: *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)
);
```

Предположим, вы используете значение -1, чтобы заменить значение UNKNOWN.

Файл примера: *Fear-Unknown/anti/special-insert.sql*

```
INSERT INTO Bugs (assigned_to, hours) VALUES (-1, -1);
```

Столбец hours является числовым, то есть вы ограничены числовыми значениями и не можете оставить незаданное значение. В данном случае не важно, что вы выбрали именно отрицательное значение. Однако значение –1 отбрасывает такие калькуляции, как SUM() или AVG(). Вы должны будете исключить строки с этим значением, используя выражения для особых случаев, — это то, чего вы пытались избежать, запрещая использование значения NULL.

Файл примера: *Fear-Unknown/anti/special-select.sql*

```
SELECT AVG( hours ) AS average_hours_per_bug FROM Bugs
WHERE hours <> -1;
```

В другом столбце применение значения –1 могло бы быть существенным, поэтому вам придется выбирать различные значения в зависимости от конкретного случая для каждого столбца. Вы также должны запоминать или документировать специальные значения, используемые в каждом столбце. Это добавляет много дотошной и ненужной работы к проекту.

Теперь давайте вернемся к столбцу assigned_to. Он является внешним ключом к таблице Accounts. Когда возникает сообщение об ошибке, но значение ей еще не присвоено, какое значение, кроме NULL, можно использовать? Любое значение, кроме NULL, должно сослаться на строку в таблице Accounts, таким образом, вы должны создать строку, указывающую место заполнения, в таблице Accounts, обозначающую «никто» или «не назначе-

но». Может показаться нелепым создавать учетную запись для ссылки, но так вы сможете продемонстрировать отсутствие ссылки на реальную пользовательскую учетную запись.

Вам необходимо будет присвоить столбцу значение NOT NULL, так как строка потеряет свой смысл, не имея значения в данном столбце. Например, у столбца Bugs.reported_by должно быть значение, потому что о каждой ошибке кто-то должен был сообщить. Но ошибка может существовать, еще не имея присвоенного ей значения. Отсутствующие данные должны иметь значение NULL.

14.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Если вы или член вашей команды столкнулись со следующими проблемами, это могло произойти из-за неверной обработки значения NULL.

- «Как мне найти строку, в которой не было задано значение для столбца assigned_to (или любого другого)?»

Вы не можете использовать оператор равенства для значения NULL. Вы узнаете, как использовать предикат IS NULL далее в этой главе.

- «Полные имена некоторых пользователей не отображаются в представлении приложения, но они есть в базе данных.»

Проблема может состоять в том, что вы объединяете строки со значением NULL, получая в итоге NULL.

ЯВЛЯЮТСЯ ЛИ ЗНАЧЕНИЯ NULL РЕЛЯЦИОННЫМИ?

В SQL существуют некоторые противоречия. Э. Ф. Кодд (E. F. Codd), программист, разработавший реляционную теорию, выяснил необходимость использования значения NULL для обозначения отсутствующих данных. Однако К. Дж. Дейт (C. J. Date) показал, что поведение значения NULL, определенное в стандарте SQL, имеет некоторые спорные случаи, конфликтующие с реляционной логикой.

Факт в том, что большинство языков программирования несовершенны в реализации теорий информатики. Язык SQL поддерживает значение NULL, плохо это или хорошо. Мы узнали о некоторых рисках, но вы можете узнать, как подготовиться к этим случаям и использовать NULL продуктивно.

- «Отчет о целых часах, потраченных на работы над данным проектом, включает только часть ошибок, которые мы исправили! Только ту часть, для которой мы назначили приоритет».

Ваш агрегатный запрос суммировать часы, вероятно, включает выражение в операторе WHERE, которое не может быть истиной, когда приоритет имеет значение NULL. Вы получите неожиданные результаты при использовании выражений, которые *не равны*. Например, в строках, где priority — NULL, приоритет выражений не может быть <> 1.

- «Оказывается, мы не можем использовать строку, которую мы использовали для обозначения *неизвестности* в таблице Bugs, поэтому мы должны встретиться, чтобы обсудить, какое новое специальное значение мы можем использовать и оценить время, потраченное на перемещение данных и преобразования кода, чтобы стало возможным использовать это значение».

Это вероятное последствие назначения специального флагового значения, которое могло быть оправданным в домене вашего столбца. Однако вы можете обнаружить, что вам необходимо использовать это значение в его буквальном смысле, а не во флаговом.

Распознавание проблем с манипуляциями со значением NULL может быть труднодостижимым. Проблемы могут не проявиться во время тестирования приложения, особенно если вы пропустили некоторые спорные случаи, проектируя образцы данных для тестов. Однако когда ваше приложение будет использоваться в продукции, данные могут принять множество непредусмотренных форм. Если значение NULL может попасть в данные, то не сомневайтесь, что рано или поздно это произойдет.

14.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Использование значения NULL само по себе не является антипаттерном; им будет являться использование значения NULL в качестве обычного значения или использование обычного значения как NULL.

Одной из ситуаций, где необходимо обработать значение NULL как обычное, является импорт или экспорт внешних данных. В текстовом файле с отделенными запятыми полями все значения должны быть представлены текстом. Например, в mysqlimport, инструментальном средстве MySQL, используемом при загрузке данных из текстового файла, значение NULL обозначается следующим образом: \N.

Аналогично пользователь при вводе данных не может непосредственно ввести значение NULL. Приложение, в которое вводятся данные, может предоставить способ ввести определенную последовательность символов, чтобы обозначить значение NULL. Например, Microsoft.NET версии от 2.0 и далее поддерживает свойство, называемое ConvertEmptyStringToNull для пользовательских веб-интерфейсов. Параметры и связанные поля благодаря этому свойству автоматически преобразуют пустое значение строки («») в значение NULL.

Наконец, значение NULL не будет работать в некоторых случаях с отсутствующим значением. Скажем, вы хотите пометить ошибку, которой еще не дано значение или было дано работником, покинувшим проект, — в каждой из этих ситуаций необходимо использовать различные значения.

14.5. РЕШЕНИЕ: ИСПОЛЬЗОВАНИЕ NULL В КАЧЕСТВЕ УНИКАЛЬНОГО ЗНАЧЕНИЯ

Большинство проблем со значением NULL основано на распространенном заблуждении по поводу поведения трехзначной логики SQL. Для программистов, приученных к обычной логической схеме «истина/ложь» в большинстве языков программирования, это может стать основной проблемой. Вы можете обрабатывать данные, содержащие значение NULL в SQL-запросах, лишь немного изучив поведение значения NULL.

Значение NULL в скалярных выражениях

Предположим, Стену — 30 лет, в то время как возраст Оливера неизвестен. Если я спрашиваю вас, старше ли Стен Оливера, ваш единственный возможный ответ: «Я не знаю». Если я спрашиваю вас, того же возраста Стен по отношению к Оливеру или нет, ваш ответ также будет: «Я не знаю». Если я спрашиваю вас, каков суммарный возраст Стена и Оливера, ваш ответ снова: «Я не знаю».

Предположите, возраст Чарли также неизвестен. Если я спрашиваю вас, равен ли возраст Оливера возрасту Чарли, ваш ответ по-прежнему: «Я не знаю». Этот пример показывает, почему результат сравнения NULL = NULL является также значением NULL.

Таблица ниже описывает некоторые случаи, когда программисты ожидают один результат, но получают другой.

Выражение	Ожидаемый результат	Фактический результат	Причина
NULL = 0	TRUE	NULL	Значение NULL не равносильно нулю.
NULL = 12345	FALSE	NULL	Если неуказанное значение приравнивается к данному, итогом будет неизвестность.
NULL <> 12345	TRUE	NULL	Тот же итог при неравенстве.
NULL + 12345	12345	NULL	Значение NULL не равносильно нулю.
NULL || 'string'	'string'	NULL	Значение NULL не является пустой строкой.

Окончание табл.

Выражение	Ожидаемый результат	Фактический результат	Причина
NULL = NULL	TRUE	NULL	Если неуказанное значение приравнивается к неуказанному, итогом будет неизвестность.
NULL <> NULL	FALSE	NULL	Тот же итог при неравенстве.

Разумеется, данные примеры применимы не только к использованию ключевого слова NULL, но и к столбцам или выражениям, содержащим значение NULL.

Значение NULL в булевых выражениях

Ключевой концепцией понимания поведения значения NULL в булевых выражениях является то, что значение NULL — ни истина, ни ложь.

Таблица ниже описывает некоторые случаи, когда программисты ожидают один результат, но получают другой.

Выражение	Ожидаемый результат	Фактический результат	Причина
NULL AND TRUE	FALSE	NULL	Значение NULL — не ложь.
NULL AND FALSE	FALSE	FALSE	Любое правдивое значение AND FALSE — ложь.
NULL OR FALSE	FALSE	NULL	Значение NULL — не ложь.
NULL OR TRUE	TRUE	TRUE	Любое правдивое значение OR TRUE — истина.
NOT (NULL)	TRUE	NULL	Значение NULL — не ложь.

Значение NULL не является истиной, но оно и не ложно. Если бы оно было ложным, то применение NOT к значению NULL в результате давало бы истину. Но NULL работает по иному принципу: результатом NOT (NULL) является NULL. Это смущает некоторых программистов, пользующихся булевыми выражениями со значением NULL.

Поиск значения NULL

Ни равенство, ни неравенство не возвращают истину при сравнении одного значения со значением NULL, вам понадобится другая операция, чтобы найти значение NULL. Более ранние стандарты SQL определяют предикат IS

NULL, как возвращающий истину, если его единственный операнд — NULL. Обратный ему предикат IS NOT NULL возвращает ложь, если операнд — NULL.

Файл примера: *Fear-Unknown/soln/search.sql*

```
SELECT * FROM Bugs WHERE assigned_to IS NULL;

SELECT * FROM Bugs WHERE assigned_to IS NOT NULL;
```

ПРАВИЛЬНЫЙ РЕЗУЛЬТАТ ПО НЕПРАВИЛЬНОЙ ПРИЧИНЕ

Рассмотрите приведенный ниже пример, в котором поведение столбца, способного содержать значения NULL, интуитивно совпадает с желаемым результатом.

```
SELECT * FROM Bugs WHERE assigned_to <> 'NULL';
```

В данном примере столбец assigned_to, способный содержать значения NULL, сравнивается со строковым значением 'NULL' (обратите внимание на одинарные кавычки), а не с самим ключевым словом NULL.

В той ячейке, в которой столбец assigned_to принимает значение NULL, результат сравнения со строковым значением 'NULL'- не истина. Строка исключается результатом запроса, что и является целью программиста.

Возможен другой вариант — столбец, содержащий целочисленные значения, сравнивается со строкой 'NULL'. Целочисленное значение такой строки, как 'NULL', является нулем в большинстве баз данных. Целочисленное значение столбца assigned_to почти наверняка больше чем ноль, то есть не равно строке. Поэтому строка будет включена в результат запроса.

Таким образом, совершая одну часто допускаемую ошибку, такую как обособление кавычками ключевого слова NULL, некоторые программисты могут случайно получить желаемый результат. К сожалению, это совпадение не произойдет при иных вариантах поиска, например: WHERE assigned_to = 'NULL'.

Кроме того, стандарт SQL-99 определяет другой предикат сравнения: IS DISTINCT FROM. Он работает как обычный оператор неравенства <> за исключением того, что всегда возвращает истину или ложь, даже когда ее операнды — значения NULL.

Работа с данным предикатом освободит вас от написания громоздких выражений, предназначенных для тестирования IS NULL, перед началом сравнения со значением. Следующие два запроса эквивалентны:

Файл примера: *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;
```

Вы можете использовать этот предикат с параметрами запроса, которым вы хотите задать символьное значение или NULL:

Файл примера: *Fear-Unknown/soln/is-distinct-from-parameter.sql*

```
SELECT * FROM Bugs WHERE assigned_to IS DISTINCT FROM ?;
```

Поддержка предиката IS DISTINCT FROM среди различных производителей баз данных довольно противоречива. СУБД PostgreSQL, IBM DB2 и Firebird поддерживают его, когда как Oracle и Microsoft SQL Server — еще нет. MySQL предлагает собственный оператор <=>, работающий как предикат IS NOT DISTINCT FROM.

Присвойте столбцам значение NOT NULL

Рекомендуется задать ограничение NOT NULL для столбца в том случае, если значение NULL повредило бы концепции приложения или лишало бы его смысла. Лучше поставить ограничение на всю базу данных, чем полагаться на программный код.

Например, разумно, что любая запись в таблице Bugs имеет значение, отличное от NULL для столбцов date_reported, reported_by и status. Аналогично, строки, в данном случае — bug_id, в дочерних таблицах, таких как Comments, должны содержать значения, отличные от NULL, ссылающиеся на существующую ошибку. Вам необходимо присвоить таким столбцам значение NOT NULL.

Некоторые программисты рекомендуют присваивать каждому столбцу значение DEFAULT, чтобы, если вы не включите столбец в оператор INSERT, столбец получил некоторое значение, а не NULL. Этот вариант хорош для одних столбцов, но плох для других. Например, столбец Bugs.reported_by не должен содержать NULL. Какое значение по умолчанию вы должны задать для этого столбца, если таковое существует? Нередко столбцы нуждаются в ограничении NOT NULL, при этом не имея логического значения по умолчанию.

Динамические значения по умолчанию

В некоторых запросах вам понадобится умышленно поставить ограничение NOT NULL для столбца или выражения с целью упрощения логики запроса, но вы не хотите, чтобы это значение было сохранено. То, что вам нужно, — это способ присвоения значения по умолчанию для данного столбца или выражения в специальном запросе. Для этого вам необходимо воспользоваться функцией COALESCE(). Эта функция принимает варьируемое число параметров и возвращает первый, отличный от значения NULL, параметр.

В примере с конкатенации столбцов с именами пользователей в начале данной главы вы могли бы воспользоваться функцией COALESCE(), чтобы

создать выражение, использующее одиночный интервал вместо отсутствующего отчества, что не привело бы к тому, что ячейка с присвоенным значением NULL присваивала значение NULL всему выражению.

Файл примера: *Fear-Unknown/soln/coalesce.sql*

```
SELECT first_name || COALESCE(' ' || middle_initial || ' ' , '
' ) || last_name

AS full_name

FROM Accounts;
```

COALESCE() — это стандартная функция SQL. Некоторые производители баз данных поддерживают аналогичные функции такие, как NVL() или ISNULL().

ВНИМАНИЕ!

Используйте значение NULL, чтобы обозначить отсутствующее значение для любого типа данных.

Интеллект различает возможное и невозможное;
причина различает имеющее смысл и бессмысленное.
Даже возможное может быть бессмысленным.

Макс Борн

ГЛАВА 15. НЕОДНОЗНАЧНЫЕ ГРУППЫ

Предположим, вашему начальнику нужно знать, какие проекты в базе данных ошибок являются активными, а какие — нет. Один отчет, который он попросил вас сгенерировать, содержит последнюю ошибку, о которой сообщено в продукте. Вы пишете запрос, используя СУБД MySQL, чтобы вычислить наибольшее значение в столбце `date_reported` в группе ошибок, которая пользуется данными столбца `product_id`. Отчет будет выглядеть следующим образом:

product_name	latest	bug_id
Open RoundFile	01.06.2010	1234
Visual TurboBuilder	16.02.2010	3456
ReConsider	01.01.2010	5678

Ваш начальник — педантичный человек, он тратит некоторое время, изучая все ошибки, перечисленные в отчете. Он замечает, что последняя в списке строка «Open RoundFile» имеет идентификационный номер, указанный в столбце `bug_id`, говорящий о том, что эта ошибка не является последней. Одни данные в совокупности не соответствуют другим:

product_name	**date_reported**	**bug_id**	
Open RoundFile	19.12.2009	*1234*	Идентификационный номер данной ошибки...
Open RoundFile	*01.06.2010*	2248	не совпадает с выделенной датой.
Visual TurboBuilder	16.02.2010	3456	
Visual TurboBuilder	10.02.2010	4077	
Visual TurboBuilder	16.02.2010	5150	

Окончание табл.

product_name	date_reported	bug_id	
ReConsider	01.01.2010	5678	
ReConsider	19.11.2009	8063	

Как вы сможете объяснить эту проблему? Как это повлияет на данный продукт и повлияет ли на остальные? Как вам получить требуемый отчет?

15.1. ЦЕЛЬ: ПОЛУЧЕНИЕ СТРОКИ С НАИБОЛЬШИМ ЗНАЧЕНИЕМ В ГРУППЕ

Большинство программистов, изучавших SQL, приходят к использованию оператора `GROUP BY` в запросах, применяя некоторые одинарные функции в группах строк и получая в результате одну строку из группы. `GROUP BY` — мощная функция, облегчающая получение большого разнообразия комплексных отчетов и использующая относительно мало кода.

Ниже приведен пример запроса, чтобы вывести последнюю ошибку в каждом продукте в базе данных ошибок:

Файл примера: *Groups/anti/groupbyproduct.sql*

```
SELECT product_id, MAX(date_reported) AS latest
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

Запрос будет логично расширить до запроса идентификационного номера определенной ошибки, о которой было сообщено в последнюю очередь:

Файл примера: *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 приводит или к ошибке, или к ненадежному ответу. Это распространенный источник проблем для программистов, использующих язык SQL.

Цель состоит в том, чтобы найти не просто наибольшее (наименьшее или среднее) значение в группе, но и включить в запрос другие свойства строки, которой это значение принадлежит.

15.2. АНТИПАТТЕРН: ССЫЛКА НА НЕСГРУППИРОВАННЫЕ СТОЛБЦЫ

Причина создания этого антипаттерна проста: он показывает распространенное заблуждение многих программистов о том, как работает группировка в SQL.

Правило единственного значения

Строки в каждой группе — это те строки, которые имеют одинаковые значения в столбце или столбцах, указанные после предложения `GROUP BY`. Например, в указанном ниже запросе для группы назначена одна строка для каждого различного значения в столбце `product_id`.

Файл примера: *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 списке выборки запроса должен иметь одно-единственное значение для всех строк в группе. Таково *Правило единственного значения*. Столбцы, перечисленные в операторе `GROUP BY`, гарантированно будут иметь строго одно значение в группе, вне зависимости от того, сколько строк данная группа включает.

Выражение `MAX()` также гарантирует вывод единственного значения для каждой группы: в данном случае наибольшего значения среди всех строк в группе.

Однако когда речь идет о сервере базы данных в целом, нельзя быть абсолютно уверенными в отношении всех остальных столбцов, перечисленных в списке выборки. Нельзя дать гарантию, что одинаковые значения принадлежат каждой строке в других столбцах.

Файл примера: *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 данном примере будет получено много отличных значений столбца `bug_id` от данного столбца `product_id`, поскольку таблица `BugsProducts` включает множество ошибок для данного продукта. В запросе группировки, сокращающем строки до единственной строки на один продукт, исчезает возможность представить все значения столбца `bug_id`.

Поскольку больше нельзя гарантировать, что каждому особому столбцу будет присвоено единственное значение, СУБД это расценивает как нарушение Правила единственного значения. В базах данных большинства производителей появляется сообщение об ошибке, если вы вводите какой-либо запрос, пытающийся вернуть столбец, не принадлежащий числу столбцов, названных в предложении `GROUP BY`, или являющихся аргументами агрегатной функции.

СУБД MySQL и SQLite имеют поведение, отличающееся от СУБД других производителей. Оно будет изучено в разделе 15.4 «Допустимые способы использования антипаттерна».

Запросы «делай то, что я думаю»

Среди программистов распространено заблуждение, что СУБД может предположить, какое именно значение столбца `bug_id` вы хотите получить в отчете, основываясь только на том, что в другом столбце используется функция `MAX()`. Многие считают, что если в запросе выбирается наибольшее значение, то другие названные столбцы возьмут значения из той строки, в которой находится искомое наибольшее значение.

К сожалению, СУБД не может произвести такой вывод в нескольких случаях.

- Если две ошибки имеют одинаковые значения в столбце `date_reported` и они являются наибольшими значениями в группе, какой идентификационный номер из этих двух ошибок столбца `bug_id` должен отобразиться в отчете?
- Если вы сделали запрос для двух различных агрегатных функций, например `MAX()` и `MIN()`, они, вероятно, соответствуют двум различным строкам в группе. Какое тогда значение столбца `bug_id` запрос должен вывести для данной группы?

Файл примера: *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 соответствует значению, возвращенному агрегатной функцией, каково значение `bug_id`? Такое часто случается при использовании функций `AVG()`, `COUNT()`, и `SUM()`.

Файл примера: *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;
```

Подобные примеры Правила единственного значения очень важны. Не каждый запрос, вынуждающий следовать этому правилу, привел бы к неоднозначному результату, но большинство все же приводит. Было бы разумно, если бы СУБД могла отличить неоднозначный запрос от однозначного и оповещать об ошибке только когда данные содержат неоднозначность. Но это отрицательно сказалось бы на надежности приложения, поскольку один и тот же запрос мог бы быть правильным или недопустимым, в зависимости от состояния данных.

15.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

В базах данных большинства производителей в то время, как вы пишете запрос, нарушающий Правило единственного значения, СУБД сообщает об ошибке немедленно. Ниже приведены примеры сообщений об ошибках в некоторых базах данных:

- Firebird 2.1:

```
Invalid expression in the select list (not contained in either
an aggregate function or the GROUP BY clause)1
```

- IBM DB2 9.5:

```
An 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.2
```

---

[1] Недопустимое выражение в списке выборки (не содержит агрегатную функцию либо оператор GROUP BY)

[2] Выражение, начинающееся с "*BUG_ID*", указанное в выражении SELECT, выражении HAVING или выражении ORDER BY, не указано в выражении GROUP BY, или выражение указано в выражении SELECT, выражении HAVING или выражении ORDER BY с функцией столбца и при этом не указано в выражении GROUP BY.

GROUP BY И DISTINCT

В СУБД языка SQL поддерживается модификатор запроса DISTINCT, уменьшающий строки результатов запроса таким образом, чтобы каждая строка была уникальна. В приведенном ниже примере результат запроса будет содержать данные о том, кто и когда сообщил об ошибках, но только на каждую дату и каждого сообщившего будет приходиться только одна строка:

```
SELECT DISTINCT date_reported, reported_by FROM Bugs;
```

Аналогичного результата можно достигнуть в группируемом запросе, опуская любую агрегатную функцию. Результат запроса будет уменьшен до одной строки для каждой различной пары значений в столбце, названном в операторе GROUP BY:

```
SELECT date_reported, reported_by FROM Bugs
GROUP BY date_reported, reported_by;
```

Оба запроса приводят к одному и тому же результату, оптимизируются и выполняются одинаковым образом. Поэтому различия в данном примере — лишь вопрос предпочтения.

- Microsoft SQL Server 2008:

```
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.1
```

- MySQL 5.1 после установки SQL-режима ONLY_FULL_GROUP, запрещающего неоднозначные запросы:

```
'bugs.b.bug_id' isn't in GROUP BY2
```

- Oracle 10.2:

```
not a GROUP BY expression3
```

- PostgreSQL 8.3:

```
column «bp.bug_id" must appear in the GROUP BY clause or be
used in an aggregate function4
```

В СУБД SQLite и MySQL неоднозначные столбцы могут содержать неожиданные и ненадежные значения. В MySQL возвращенное значение принадлежит первой строке в группе, где слово *«первой»* соответствует физической памяти. СУБД SQLite дает противоположный результат: значение принадлежит *последней* строке в группе. В обоих случаях поведение не документи-

---

[1] Столбец 'Bugs.bug_id' не допустим в списке выборки, так как не содержит агрегатную функцию либо выражение GROUP BY.

[2] Столбец 'bugs.b.bug_id' не содержится в выражении GROUP BY.

[3] Не является выражением GROUP BY.

[4] Столбец "bp.bug_id" должен присутствовать в выражении GROUP BY или использоваться агрегатной функцией.

ровано, и данные СУБД не обязаны работать аналогично в будущих версиях. Только вам решать, обратить ли внимание на указанные выше случаи, чтобы проектировать запросы, избегая неоднозначности.

15.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Как видно, СУБД MySQL и SQLite не могут гарантировать надежный результат для столбца, не соответствующего Правилу единственного значения. Однако существуют случаи, когда вы можете воспользоваться фактом менее строго, в отличие от других СУБД, следования Правилу единственного значения двумя данными СУБД.

Файл примера: *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 предыдущем запросе столбец `account_name` технически нарушает Правило единственного значения, так как он не указан ни в операторе `GROUP BY`, ни в агрегатной функции. Однако есть только одно значение, возможное для столбца `account_name` в каждой группе; группы основаны на столбце `Bugs.reported_by`, являющимся внешним ключом к таблице `Accounts`. Поэтому группы один в один совпадают со строками в таблице `Accounts`.

Другими словами, если вы знаете значение столбца `reported_by`, тогда вы однозначно знаете значение столбца `account_name`, так же, как если бы вы делали запрос в первичном ключе таблицы `Accounts`.

Такой вид однозначных связей называется *функциональной зависимостью*. Самый распространенный пример — это зависимость между первичным ключом таблицы и атрибутами таблицы: столбец `account_name` зависит от его первичного ключа `account_id`. Если вы группируете запрос столбцом(ами) первичного ключа таблицы, то каждая группа соответствует отдельной строке данной таблицы, поэтому все остальные столбцы той же таблицы должны иметь единственное значение для каждой группы.

Столбец `Bugs.reported_by` имеет подобные связи с зависящими от него атрибутами таблицы `Accounts`, поскольку он ссылается на первичный ключ таблицы `Accounts`. Когда запрос группируется по столбцу `reported_by`, являющемуся внешним ключом, атрибуты таблицы `Accounts` функционально зависят друг от друга, а результат запроса не содержит неоднозначности.

Однако большинство марок СУБД по-прежнему возвращает ошибку. Во-первых, таков стандарт SQL, а во-вторых, будет не слишком дорого выяснить функциональные зависимости на ходу[1]. Однако если вы используете СУБД MySQL или SQLite и делаете все возможное, чтобы создать запрос только функционально зависящих столбцов, вы можете использовать этот вид группируемого запроса и по-прежнему избегать проблем с неоднозначностью.

15.5. РЕШЕНИЕ: ОДНОЗНАЧНОЕ ИСПОЛЬЗОВАНИЕ СТОЛБЦОВ

Следующие разделы описывают несколько способов, при помощи которых вы можете решить данный антипаттерн и создать однозначные запросы.

Создание запроса только функционально зависящих столбцов

Самое прямое решение состоит в том, чтобы удалить неоднозначные столбцы из запроса.

Файл примера: *Groups/anti/groupbyproduct.sql*

```
SELECT product_id, MAX(date_reported) AS latest
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

Запрос выводит дату последней ошибки в продукте, даже при том, что он не отображает номер последней ошибки из столбца `bug_id`. Иногда этого достаточно, так что не пропускайте простое решение.

Использование коррелированных подзапросов

Корреллированный вложенный запрос содержит адрес внешнего запроса и поэтому выдает различные результаты для каждой строки внешнего запроса. Мы можем использовать это, чтобы найти последнюю ошибку в продукте, выполняя подзапрос ошибки в том же продукте, но с более поздней датой. Если вложенный запрос не нашел ни одну ошибку, следовательно, ошибка во внешнем запросе является последней.

Файл примера: *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
```

[1] Приведенные в этой главе в качестве примера запросы просты. Вычисление функциональных зависимостей для любого произвольного SQL-запроса более трудно.

```
(SELECT * FROM Bugs b2 JOIN BugsProducts bp2 USING (bug_id)
WHERE bp1.product_id = bp2.product_id
AND b1.date_reported < b2.date_reported);
```

Используйте этот способ как простое решение, которое легко прочитать и перевести в код. Однако имейте в виду, что данное решение, вероятно, отрицательно скажется на производительности, поскольку коррелированные вложенные запросы выполняются для каждой строки во внешнем запросе.

Использование производной таблицы

Вы можете использовать подзапрос как *производную таблицу*, получая промежуточный результат, содержащий только идентификационный номер каждого продукта из столбца `product_id` и соответствующую ему наиболее позднюю дату сообщения об ошибке для данного продукта. Затем используйте этот результат, объединив таблицы друг напротив друга, таким образом, чтобы результат запроса содержал только ошибки с наиболее поздней датой для каждого продукта.

Файл примера: *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	01.06.2010	2248
2	16.02.2010	3456
2	16.02.2010	5150
3	01.01.2010	5678

Обратите внимание, что вы можете получить несколько строк для одного продукта, если последняя дата, возвращенная подзапросом, содержится в нескольких строках. Если вы хотите, чтобы каждая строка соответствовала отдельному идентификационному номеру продукта, вы можете использовать другую функцию группировки во внешнем запросе:

Файл примера: *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	latest_bug_id
1	01.06.2010	2248
2	16.02.2010	5150
3	01.01.2010	5678

Используйте решение с производной таблицей как более масштабную альтернативу для коррелированных подзапросов. Производная таблица некоррелирована, поэтому большинство марок СУБД должны выполнять подзапрос однажды. Однако база данных должна сохранять набор промежуточных результатов во временной таблице, что вновь отрицательно сказывается на производительности.

Использование оператора JOIN

Вы можете создать соединение, сопоставляющее между собой набор строк, которые, возможно, не существуют. Такой тип соединения называется *внешнее соединение*. Если совпавшие строки не существуют, СУБД использует значение NULL для всех столбцов, в которых содержится несуществующая строка. Поэтому, когда запрос выводит значение NULL, становится известно, что ни одна строка не была найдена.

Файл примера: *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	01.06.2010	2248
2	16.02.2010	5150
3	01.01.2010	5678

Понадобится всего несколько минут пристального изучения и, возможно, пара черточек в блокноте, чтобы понять, как это работает. Но когда вы начнете работать, данная техника может стать важным инструментом.

Используйте оператор JOIN, когда важна масштабируемость запроса с большими наборами данных. Несмотря на то, что JOIN — инструмент более жесткого контроля, в связи с чем его трудно поддерживать, данный способ масштабирования зачастую лучше, чем решение, основанное на подзапросах. Помните, что лучше проверять производительность при использовании нескольких форм запросов, чем предположить, что один лучше другого.

Использование агрегатных функций для дополнительных столбцов

Вы можете применить к дополнительному столбцу Правило единственного значения, включив в него агрегатную функцию.

Файл примера: *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 когда уверены, что ошибка, чей идентификационный номер стоит последним в столбце bug_id, имеет наиболее позднюю дату сообщения о ней, иными словами, если можно гарантировать, что ошибки расположены в хронологическом порядке сообщения о них.

Конкатенация всех значений в группе

Наконец, вы можете использовать другую агрегатную функцию в столбце bug_id, чтобы избежать нарушения Правила единственного значения. СУБД MySQL и SQLite поддерживают функцию GROUP_CONCAT (), которая объединяет все значения в группе в одно значение. По умолчанию это — обособленная запятыми строка.

Файл примера: *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	01.06.2010	1234,2248
2	16.02.2010	3456,4077,5150
3	01.01.2010	5678,8063

Данный запрос не показывает, какой номер ошибки соответствует последней дате; в столбец bug_id_list включены все значения bug_id в каждой группе.

Другой недостаток этого решения заключается в том, что оно не входит в стандарты SQL и остальные модели СУБД не поддерживают данную функцию. Некоторые модели СУБД поддерживают пользовательские функции и пользовательские агрегатные функции. Ниже приведен пример решения для СУБД PostgreSQL:

Файл примера: *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 другие модели СУБД не поддерживают пользовательские функции, поэтому в качестве решения может потребоваться написание хранимой процедуры, циклично обрабатывающей все несгруппированные результаты запросов, объединяя значения самостоятельно.

Используйте это решение, когда ожидаете, что дополнительный столбец будет иметь единственное значение для каждой группы, но столбец все еще нарушает Правило единственного значения.

ВНИМАНИЕ!

Следуйте Правилу единственного значения, чтобы избежать неоднозначных результатов запроса.

Генерация случайных чисел слишком важна, чтобы оставлять ее на волю случая.

Роберт Р. Кавью

ГЛАВА 16. СЛУЧАЙНЫЙ ВЫБОР

Вы пишете веб-приложение, которое отображает рекламные объявления. Вы рассчитываете на то, что приложение будет выбирать случайную рекламу для показа, чтобы у всех ваших рекламодателей была равная возможность продемонстрировать свою рекламу и чтобы читателям не наскучило постоянно видеть одно и то же рекламное объявление.

Первые несколько дней дела идут хорошо, но приложение постепенно становится вялым. Несколько недель спустя посетители начинают жаловаться, что ваш веб-сайт слишком медленный. Вы понимаете, что это действительно так; при открытии страниц сайта вы чувствуете разницу во времени по сравнению с тем, как это было раньше. Посетители вашего сайта начинают терять к нему интерес и трафик снижается.

По опыту прошлых событий вы сначала пытаетесь найти узкие места производительности, используя инструменты профилей и версию теста вашей базы данных с выборочными значениями. Вы измеряете время загрузки веб-страницы, но, что любопытно, нет никаких проблем с производительностью ни в одном из SQL-запросов, используемых для создания страницы. Тем не менее производительность сайта продолжает снижаться.

Наконец вы понимаете, что база данных на веб-сайте вашей продукции намного больше, чем выборка в ваших тестах. Вы повторяете тестирование для базы данных такого же размера с данными о продукции и получаете в результате запрос о выборе рекламного объявления. С огромным числом рекламных объявлений, из которых должен производиться выбор, производительность такого запроса резко падает. Вы обнаружили запрос, который вы не в состоянии масштабировать, — это первый важный шаг к решению проблемы.

Как вы можете перепроектировать запрос, выбирающий случайные рекламные объявления, прежде чем ваш веб-сайт потеряет свою аудиторию, а следовательно, и ваших спонсоров?

16.1. ЦЕЛЬ: ВЫБОР ТИПОВОЙ СТРОКИ

Удивительно, как часто мы нуждаемся в SQL-запросах, возвращающих случайный результат. Кажется, что это противоречит принципам повторяемости и детерминированного программирования. Однако нет ничего необыч-

ного в том, чтобы запросить выборку из большого объема данных. Ниже приведено несколько примеров:

- отображение подсвечиваемого вращающегося контента, например, рекламного объявления или сообщения с новостью;
- демонстрация вложенного набора записей;
- направление входящих вызовов доступным операторам;
- генерация тестовых данных.

Лучше сделать запрос данной выборки, чем производить выборку всего набора данных в вашем приложении, только так вы можете снять выборку из набора данных.

Цель состоит в том, чтобы написать эффективный SQL-запрос, который возвращает только случайную выборку данных[1].

16.2. АНТИПАТТЕРН: СОРТИРОВКА ДАННЫХ СЛУЧАЙНЫМ ОБРАЗОМ

Самый распространенный трюк в SQL: чтобы выбрать случайную строку из запроса, нужно сортировать его случайным образом и затем выбрать первую строку. Эта методика проста для понимания и легка в исполнении:

Файл примера: *Random/anti/orderby-rand.sql*

```
SELECT * FROM Bugs ORDER BY RAND() LIMIT 1;
```

Хоть это и популярное решение, оно быстро показывает свои слабые места. Чтобы понять, в чем заключается первый проблемный момент, давайте для начала сравним такой способ сортировки с обычной сортировкой, в которой сравниваются значения в столбце и выстраиваются в определенной последовательности строки, в которых значение данного столбца больше или меньше эталонного. Это повторяемая сортировка: она имеет один и тот же результат, когда вы выполняете ее больше одного раза. В этом смысле также эффективны индексы, поскольку они, по сути, являются предварительным сортировочным набором значений данного столбца.

Файл примера: *Random/anti/indexed-sort.sql*

```
SELECT * FROM Bugs ORDER BY date_reported;
```

Если критерий сортировки — функция, возвращающая случайное значение для каждой строки, то каждая строка случайным образом становится больше или меньше другой. Поэтому данная последовательность не имеет ника-

[1] Математики и программисты отделяют понятия действительно случайного и *псевдослучайного*. Фактически компьютер может создать только псевдослучайные значения.

кого отношения к значениям в каждой строке. Порядок строк меняется каждый раз, когда вы производите подобную сортировку. Пока нам такой результат подходит.

В сортировке по недетерменированному выражению (`RAND()`) не могут быть задействованы индексы строк. Индексы строк не содержат значений, случайным образом возвращенных функцией. В этом заключается суть случайности: каждый раз при выборе значения различны и непредсказуемы.

Ситуация с индексами является проблемой, так как использование индексов — это один из лучших способов ускорить сортировку. Как следствие — СУБД вынуждена сортировать набор результатов запроса самостоятельно. Этот процесс называют *сканированием таблицы*, он зачастую включает сохранение всего результата во временной таблице и сортировку этого результата при помощи добавления строк. Сканирование таблицы — это гораздо более медленный способ сортировки, нежели сортировка с помощью индексов, и различие в производительности растет с увеличением размера набора данных.

Другая слабость метода случайной сортировки заключается в том, что при дорогостоящем процессе сортировки всего набора данных бóльшая часть работы проделывается впустую, поскольку в результате все строки, кроме первой, отбрасываются. В таблице с тысячей строк зачем тратить время на сортировку всей тысячи, если нам нужна только одна строка?

Обе этих проблемы не так заметны, когда выполняется запрос для небольшого количества строк, поэтому данное решение может применяться при разработке и тестировании. Однако при увеличении вашей базы данных с течением времени запрос теряет способность хорошо масштабироваться.

16.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Метод, показанный в антипаттерне, является прямым, и многие программисты используют его, узнав о нем из статьи или придя к нему самостоятельно. Некоторые из приведенных ниже цитат являются подсказками, что ваши коллеги используют данный антипаттерн.

- «В SQL возврат случайной строки является действительно медленным».

Запрос на выбор случайной строки работал хорошо для небольшого объема данных во время разработки и тестирования, но стал значительно замедляться по мере роста объема данных. Никакие дополнительные настройки сервера СУБД, индексации или кэширования не смогут улучшить масштабируемость запроса.

- «Как я могу увеличить память моего приложения. Мне необходимо выбрать все строки, а затем выбрать одну произвольную строку».

Не стоит грузить все строки в приложении — это очень расточительно. Кроме того, база данных может увеличиваться до более значительных размеров, чем память вашего приложения может вместить.

> «Вам не кажется, что некоторые записи возникают более часто, чем должны? Этот генератор случайных чисел, похоже, создает не такие уж случайные числа».

Ваши случайные числа не согласованы с интервалами между значениями первичного ключа в базе данных (см. в разделе 16.5 «Выберите следующее наибольшее значение ключа» далее).

16.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Неэффективность решения случайной сортировки терпима, если ваш набор данных будет небольшим.

Например, вы могли бы использовать данный метод, чтобы выбрать программиста, который будет исправлять какую-либо ошибку. Будем считать, что у вас никогда не будет столько программистов, чтобы применять высоко масштабируемый метод ради выбора одного случайного человека.

В качестве другого примера можно было бы выбрать случайный штат США из списка в 50 штатов, это небольшой список, и вряд ли он увеличится во время нашей жизни.

16.5. РЕШЕНИЕ: БЕЗ КАКОГО-ЛИБО ОПРЕДЕЛЕННОГО ПОРЯДКА...

Техника случайной сортировки — это пример запроса, выполняющего сканирование таблицы и затратную сортировку. Когда вы разрабатываете решения в SQL, следует брать во внимание и неэффективные решения, как это. Вместо того чтобы искать способ оптимизировать неоптимизируемый запрос, заново обдумайте свой метод. Можно использовать альтернативные методы запроса случайной строки из набора результатов запроса, рассмотренные в следующих разделах. При различных обстоятельствах каждое из этих решений может привести к одинаковым результатам, но с большей эффективностью.

Выберите случайное значение ключа между единицей и максимальным значением

Существует метод, не сортирующий всю таблицу, а выбирающий случайное значение между единицей и наибольшим значением первичного ключа.

Файл примера: *Random/soln/rand-1-to-max.sql*

```
SELECT b1.*
FROM Bugs AS b1
JOIN (SELECT CEIL(RAND() * (SELECT MAX(bug_id) FROM Bugs)) AS
rand_id) AS b2
   ON (b1.bug_id = b2.rand_id);
```

Данное решение предполагает, что значения первичного ключа начинаются с единицы и увеличиваются непрерывно. То есть нет пропущенных чисел от значения 1 до наибольшего значения. Если есть пропущенные значения, то случайно выбранное значение может отсутствовать в строках таблицы.

Используйте это решение, когда знаете, что в вашем ключе присутствуют все значения от единицы до максимального значения ключа.

Выберите следующее наибольшее значение ключа

Данное решение подобно предыдущему, но если диапазон значений ключа имеет промежутки с неиспользуемыми значениями, то запрос будет сравнивать случайное значение с существующими значениями первичного ключа.

Файл примера: *Random/soln/next-higher.sql*

```
SELECT b1.*
FROM Bugs AS b1
JOIN (SELECT CEIL(RAND() * (SELECT MAX(bug_id) FROM Bugs)) AS
bug_id) AS b2
WHERE b1.bug_id >= b2.bug_id
ORDER BY b1.bug_id
LIMIT 1;
```

Таким образом, проблема случайного числа, не попадающего в диапазон значений ключа, будет решена, но это значит, что значение ключа, следующее за пропуском, будет выбираться чаще. Случайные значения должны выбираться более или менее равномерно по всему диапазону, но в случае со значениями столбца `bug_id` равномерности не будет.

Используйте это решение, когда промежутки в диапазоне значений редки и когда не важно, будут ли значения ключей выбираться с одинаковой частотой.

Создайте список всех значений ключей и выберите одно наугад

Используйте код приложения, чтобы с его помощью выбрать одно значение первичного ключа в наборе результатов. Затем запросите всю строку в базе данных, используя значение выбранного ключа. Данный метод показан ниже в PHP-коде:

Файл примера: *Random/soln/rand-key-from-list.php*

```
<?php
$bug_id_list = $pdo->query("SELECT bug_id FROM Bugs")-
>fetchAll();

$rand = random( count($bug_id_list) );
$rand_bug_id = $bug_id_list[$rand]["bug_id"];
$stmt = $pdo->prepare("SELECT * FROM Bugs WHERE bug_id = ?");
$stmt->execute( array($rand_bug_id) );
$rand_bug = $stmt->fetch();
```

Данное решение помогает избежать сортировки всей таблицы, и шансы выбора каждого ключа приблизительно равны, однако оно имеет свои минусы.

- Вызов всех значений столбца `bug_id` из базы данных может возвратить список слишком большого размера. Это может даже превысить ресурсы памяти приложения и вызвать ошибку, подобную этой:

```
Fatal error: Allowed memory size of 16777216 bytes exhausted1
```

- Запрос должен быть выполнен дважды: в первый раз, чтобы создать список первичных ключей; во второй раз, чтобы выбрать случайную строку. Если запрос является комплексным и на него затрачивается много ресурсов, это становится проблемой.

Используйте это решение, когда случайная строка выбирается в простом запросе со средним размером набора результатов. Оно применимо для выбора из списка значений, состоящего из нескольких прерывистых участков.

Выберите случайную строку, используя смещение

Другой метод, помогающий избежать проблем, свойственных предыдущим альтернативам, состоит в пересчете строк в наборе данных и возврате случайного числа между нулем и количеством строк. Полученное число используется как смещение во время запроса набора данных.

---

[1] Фатальная ошибка: Допустимый объем памяти 16777216 байт исчерпан.

Файл примера: *Random/soln/limit-offset.php*

```
<?php

$rand = "SELECT ROUND(RAND() * (SELECT COUNT(*) FROM Bugs))";
$offset = $pdo->query($rand)->fetch(PDO::FETCH_ASSOC);

$sql = «SELECT * FROM Bugs LIMIT 1 OFFSET :offset»;
$stmt = $pdo->prepare($sql);
$stmt->execute( $offset );
$rand_bug = $stmt->fetch();
```

Данное решение основывается на нестандартном операторе `LIMIT`, поддерживаемом СУБД MySQL, PostgreSQL и SQLite.

Этот оператор имеет альтернативу в виде оконной функции `ROW_NUMBER()`, работающую на СУБД Oracle, Microsoft SQL Server и IBM DB2.

Ниже приведен пример решения для СУБД Oracle:

Файл примера: *Random/soln/row_number.php*

```
<?php
$rand = "SELECT 1 + MOD(ABS(dbms_random.random()),
(SELECT COUNT(*) FROM Bugs)) AS offset FROM dual";
$offset = $pdo->query($rand)->fetch(PDO::FETCH_ASSOC);

$sql = "WITH NumberedBugs AS (
SELECT b.*, ROW_NUMBER() OVER (ORDER BY bug_id) AS RN FROM Bugs
b
) SELECT * FROM NumberedBugs WHERE RN = :offset";
$stmt = $pdo->prepare($sql);
$stmt->execute( $offset );
$rand_bug = $stmt->fetch();
```

Используйте это решение, когда невозможно присвоить ключу непрерывный диапазон значений и когда вы должны быть уверены, что каждая строка имеет равные шансы быть выбранной.

Частные решения

Любой известный производитель СУБД может реализовать свое собственное решение для данного вида задач. Например, производитель СУБД Microsoft SQL Server 2005 ввел оператор `TABLESAMPLE`:

Файл примера: *Random/soln/tablesample-sql2005.sql*

```
SELECT * FROM Bugs TABLESAMPLE (1 ROWS);
```

СУБД Oracle использует несколько отличающийся оператор SAMPLE, например, чтобы возвратить 1% строк в таблице:

Файл примера: *Random/soln/sample-oracle.sql*

```
SELECT * FROM (SELECT * FROM Bugs SAMPLE (1)
ORDER BY dbms_random.value) WHERE ROWNUM = 1;
```

Следует читать документацию частных решений вашей модели СУБД. Часто в них существуют ограничения или другие варианты, о которых вы должны знать.

ВНИМАНИЕ!

Зачастую запрос невозможно оптимизировать; используйте другое решение.

Некоторые люди, сталкиваясь с проблемой, думают: «Понятно, здесь можно использовать регулярные выражения». Теперь у них две проблемы.

Джейми Завински

ГЛАВА 17. СОБСТВЕННАЯ ПОИСКОВАЯ СИСТЕМА

Я работал в технической поддержке в 1995 году, в то время, когда компании только начинали предоставлять информацию клиентам через Интернет. У нас был набор коротких документов, содержащих ответы на общие вопросы о поддержке, и мы хотели разместить их в сети Интернет в базе знаний.

Мы быстро поняли, что в связи с ростом базы знаний необходимо сделать доступным поиск по ней, так как клиенты не захотят просматривать сотни статей в поисках ответа. Одна из стратегий состояла в том, чтобы организовать категории статей, но даже эти группы были слишком большими, к тому же многие статьи принадлежали к нескольким категориям.

Мы хотели, чтобы наши клиенты искали статьи, сужая список до статей, соответствующих определенному критерию. Самый гибкий и прямой интерфейс должен был позволить клиентам вводить любую комбинацию слов и отображать статьи, в которых эти слова присутствуют. Статья должна была отобразиться выше остальных, если более полно соответствовала критериям поиска. Кроме того, мы хотели ввести согласование форм слов. Например, при поиске слова *crash* будут также найдены слова *crashed*, *crashes* и *crashing*. Разумеется, поиск по растущей базе знаний должен был работать достаточно быстро, чтобы быть полезным при работе с веб-приложением.

Если это краткое описание излишне, оно не должно вас удивить. Поиск текста онлайн стал настолько обычным, что мы едва можем вспомнить время, когда он только начинал быть доступным. Однако использование SQL в целях поиска ключевых слов, также делая решение быстрым и аккуратным, является обманчиво трудным.

17.1. ЦЕЛЬ: ПОЛНОТЕКСТОВЫЙ ПОИСК

Любое приложение, содержащее в себе текст, нуждается в поддержке поиска слова или фразы в этом тексте. Мы используем базы данных, чтобы хранить огромное количество текстовых данных, и в то же время требуем, чтобы поиск нужного текста был быстрым. Веб-приложения особенно остро нуждаются в высокоэффективных и масштабируемых возможностях СУБД по поиску текста.

Один из основных принципов языка SQL (и реляционной теории, на основе которой язык SQL был разработан) заключается в том, что значение в столбце является нерасщепимым. Таким образом, сравнивая одно значение с другим, вы всегда сравниваете целые значения. Сравнивание частей строк в SQL является неэффективным и неточным.

Тем не менее мы нуждаемся в способе сравнения короткой строки с более длинной, чтобы найти соответствие, даже если короткая строка является частью другой, более длинной строки. Как мы можем построить мост между пропастью, используя язык SQL?

17.2. АНТИПАТТЕРН: ПРЕДИКАТЫ СОПОСТАВЛЕНИЯ С ШАБЛОНАМИ

SQL поддерживает предикаты сопоставления с шаблонами с целью сравнения строк, и это — первое решение, используемое большинством программистов для поиска ключевых слов. Наиболее широко поддерживаемый различными СУБД предикат — `LIKE`.

Предикат `LIKE` поддерживает подстановочный знак `%`, соответствующий нулю или другим символам. Подстановочный знак вписывается с обеих сторон искомого ключевого слова, соответствующего какой-либо строке, которая содержит это ключевое слово. Первый подстановочный знак сопоставляется с текстом, предшествующим ключевому слову, второй — с текстом, идущим после слова.

Файл примера: *Search/anti/like.sql*

```
SELECT * FROM Bugs WHERE description LIKE '%crash%' ;
```

Регулярные выражения также поддерживаются многими моделями СУБД, хоть и не стандартным способом. Отпадает необходимость в подстановочных знаках, поскольку, условно говоря, регулярные выражения сопоставляют шаблон с подстрокой в любом случае. Ниже приведен пример использования предиката регулярного выражения в СУБД MySQL[1].

Файл примера: *Search/anti/regexp.sql*

```
SELECT * FROM Bugs WHERE description REGEXP 'crash' ;
```

Самый главный недостаток операторов сравнения с шаблоном — низкая производительность. Они не могут использовать индекс, поэтому вынуждены сканировать каждую строку в таблице. Само по себе сопоставление шаблона с каждой ячейкой строки — довольно дорогостоящая операция (например, сравнивание двух целых чисел для равенства), общая цена сканирования таблицы для такого поиска очень высока.

[1] Несмотря на то, что в SQL-99 предикат `SIMILAR TO` определен для сопоставления регулярных выражений, большинство марок СУБД SQL используют нестандартный синтаксис.

Вторая проблема простого сопоставления с шаблоном при использовании предиката LIKE или регулярных выражений состоит в том, что СУБД может найти нежелательные соответствия.

Файл примера: *Search/anti/like-false-match.sql*

```
SELECT * FROM Bugs WHERE description LIKE '%one%' ;
```

В предыдущем примере в тексте ищутся слова *one*, но СУБД также найдет строки, содержащие слова *money*, *prone*, *lonely* и т.п. Поиск шаблона с ключевым словом, разграниченным пробелами, не входит в случаи употребления слов с пунктуацией или слов в начале или конце предложения. Регулярные выражения, поддерживаемые вашей СУБД, могут поддерживать специальный шаблон для *разграничения слов*, решая данную проблему:[1]

Файл примера: *Search/anti/regexp-word.sql*

```
SELECT * FROM Bugs WHERE description REGEXP '[[:<:]]one[[:>:]]'
;
```

Учитывая проблемы с производительностью, масштабируемостью и гибкостью, вы должны принять необходимые меры, чтобы предотвратить неподходящие соответствия, простое сопоставление с шаблоном — плохая методика поиска ключевых слов.

17.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Некоторые вопросы, подобные следующим, как правило, указывают, что используется антипаттерн **Собственная поисковая система**:

- «Как мне поставить переменную между двумя подстановочными знаками в выражении LIKE?»

Такой вопрос может быть задан, когда программист хочет найти что-либо при помощи поиска сопоставления с шаблоном, используя ввод со стороны пользователя.

- «Как мне написать регулярное выражение, чтобы проверить строку, содержащую несколько слов, на *отсутствие* в ней определенного слова или на содержание в ней любой формы искомого слова?»

Если комплексная проблема выглядит слишком сложной, чтобы решить ее регулярными выражениями, то, вероятно, используется антипаттерн **Собственная поисковая система**.

- «Функция поиска на нашем веб-сайте стала работать слишком медленно, когда мы добавили в базу данных больше документов. Что произошло?»

---

[1] Данный пример использует синтаксис СУБД MySQL.

Поскольку объем данных увеличился, решение с данным антипаттерном демонстрирует слабую масштабируемость.

17.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Выражения, показанные в разделе «Антипаттерн», являются правильными SQL-запросами, они имеют прямое и простое использование. Это имеет большое значение.

Производительность очень важна, однако существуют запросы, выполняющиеся настолько редко, что не имеет смысла вкладывать большое количество ресурсов, чтобы оптимизировать их. Содержание индексов, используемых редко осуществляемыми запросами, может быть столь же дорогостоящим, как выполнение этих запросов неэффективным способом. Если это запрос особого характера, то нет никакой гарантии, что соответствующий ему индекс так или иначе принесет ему пользу.

Операторы сопоставления с шаблоном трудно использовать для комплексных запросов, однако, если вы проектируете шаблоны для простых случаев, они могут помочь вам получить необходимые результаты при минимуме суеты.

17.5. РЕШЕНИЕ: ИСПОЛЬЗОВАНИЕ ДЛЯ РАБОТЫ ПОДХОДЯЩЕГО ИНСТРУМЕНТА

Вместо SQL лучше всего использовать специализированную технологию поисковой системы. Данная альтернатива уменьшает стоимость повторных поисков, сохраняя результаты.

Следующие разделы описывают некоторые технологии, представляемые как встроенные расширения, различными моделями СУБД, а так же технологии, предлагаемые независимыми проектами. Кроме того, мы разработаем решение, использующее стандарт SQL, но в среднем более эффективное, чем поиск соответствия с подстроками.

Расширения ведущих производителей

Все крупнейшие производители СУБД изобрели собственные ответы на вопрос о полнотекстовом поиске, но их функции нестандартны или не совместимы в различных моделях СУБД. Если вы используете единственную модель СУБД (или желаете использовать зависящие от производителя функции), эти функции — лучший способ получить высокоэффективный текстовый поиск с наибольшей степенью интеграции с SQL-запросами.

Ниже даны краткие описания функций полнотекстового поиска в некоторых моделях СУБД SQL. Детали подлежат изменению, поэтому убедитесь, что прочли документацию для вашей модели СУБД.

Полнотекстовый индекс в СУБД MySQL

СУБД MySQL представляет простой вид полнотекстового индекса, предназначенный только для системы хранения данных MyISAM. Вы можете установить, что используется полнотекстовый индекс по таким названиям столбцов, как CHAR, VARCHAR или TEXT. Ниже приведен пример, по которому можно определить использование полнотекстового индекса, содержащего контент из столбцов summary и description:

Файл примера: *Search/soln/mysql/alter-table.sql*

```
ALTER TABLE Bugs ADD FULLTEXT INDEX bugfts (summary, description);
```

Используйте функцию MATCH() для поиска ключевого слова в индексированном тексте. Вы должны перечислить столбцы в полнотекстовом индексе (так вы сможете использовать в поиске другой индекс, содержащий различные столбцы в той же таблице).

Файл примера: *Search/soln/mysql/match.sql*

```
SELECT * FROM Bugs WHERE MATCH(summary, description) AGAINST
('crash');
```

Начиная с версии СУБД MySQL 4.1 вы можете использовать упрощенную систему обозначений булевых выражений в шаблоне, чтобы фильтровать результаты более тщательно.

Файл примера: *Search/soln/mysql/match-boolean.sql*

```
SELECT * FROM Bugs WHERE MATCH(summary, description)
   AGAINST ('+crash -save' IN BOOLEAN MODE);
```

Индексация текста в СУБД Oracle

СУБД Oracle поддерживает функции индексация текста, начиная с версии Oracle 8 1997 года, когда данная СУБД была частью кассеты с данными под названием ConText. Технология обновлялась несколько раз, и функция теперь интегрирована в программное обеспечение СУБД. Индексация текста в Oracle комплексна и богата, поэтому ниже приведена значительно сокращенная информация:

- CONTEXT

Создайте индекс этого типа для одного текстового столбца. Используйте оператор CONTAINS(), чтобы найти, где используется данный индекс. Индекс не согласуется с изменениями данных, поэтому вам придется перестроить индекс вручную или через список.

Файл примера: *Search/soln/oracle/create-index.sql*

```
CREATE INDEX BugsText ON Bugs(summary) INDEXTYPE IS CTSSYS.
CONTEXT;

SELECT * FROM Bugs WHERE CONTAINS(summary, 'crash') > 0;
```

- CTXCAT

Данный тип индекса специализируется на коротких текстовых выборках, например, используемых в сетевых каталогах, наряду с другими структурированными столбцами в той же таблице. Индекс с течением времени не становится противоречивым, поскольку транзакции обновляют индексированные данные.

Файл примера: *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 CTSSYS.
CTXCAT
   PARAMETERS('BugsCatalogSet' );
```

Оператор CATSEARCH() использует два аргумента для поиска в текстовых столбцах и структурированных наборах столбцов соответственно.

Файл примера: *Search/soln/oracle/ctxcat-search.sql*

```
SELECT * FROM Bugs
WHERE CATSEARCH(summary, '(crash save)' , 'status = "NEW"' ) >
0;
```

- CTXXPATH

Этот тип индекса специализируется на поиске по XML-документу при помощи оператора existsNode().

Файл примера: *Search/soln/oracle/ctxxpath.sql*

```
CREATE INDEX BugTestXml ON Bugs(testoutput) INDEXTYPE IS
CTSSYS.CTXXPATH;
```

```
SELECT * FROM Bugs
WHERE testoutput.existsNode('/testsuite/test[@status="fail"]'
) > 0;
```

- CTXRULE

Предположим, вы имеете большую коллекцию документов, хранящуюся в базе данных, и вам необходимо классифицировать их на основе их содержания.

При помощи индекса CTXRULE вы можете проектировать правила, чтобы анализировать документы и создавать отчеты по классификации этих документов. В качестве альтернативы вы можете взять выборочный набор документов, имея свою идею по поводу их классификации, и применить спроектированные производителями Oracle правила к остальной части коллекции документов. Вы можете даже полностью автоматизировать процесс, позволяя СУБД Oracle анализировать вашу коллекцию документов и создавать набор правил и классификаций для их идентификации.

Примеры использования индекса CTXRULE выходят за рамки этой книги.

Полнотекстовый поиск в СУБД Microsoft SQL Server

СУБД SQL Server 2000 (а также более поздние ее версии) поддерживает полнотекстовый поиск со сложными вариантами языковых конфигураций, тезаурусом и автоматическим отслеживанием изменений данных. СУБД SQL Server предоставляет серию хранимых процедур для создания полнотекстовых индексов, вы также можете использовать в запросах оператор CONTAINS(), чтобы задействовать полнотекстовый индекс.

Чтобы выполнить уже знакомый запрос с поиском ошибок, включающих слово *crash*, сначала включите функцию полнотекстового поиска и определите каталог вашей базы данных.

Файл примера: *Search/soln/microsoft/catalog.sql*

```
EXEC sp_fulltext_database 'enable'
EXEC sp_fulltext_catalog 'BugsCatalog' , 'create'
```

Затем определите полнотекстовый индекс в таблице Bugs, добавьте столбцы к индексу и активируйте индекс:

Файл примера: *Search/soln/microsoft/create-index.sql*

```
EXEC sp_fulltext_table 'Bugs' , 'create' , 'BugsCatalog'
'bug_id'
EXEC sp_fulltext_column 'Bugs' , 'summary' , 'add' , '2057'
EXEC sp_fulltext_column 'Bugs' , 'description' , 'add'  '2057'
EXEC sp_fulltext_table 'Bugs' , 'activate'
```

Включите функцию автоматического отслеживания изменений для полнотекстового индекса так, чтобы изменения индексированного столбца распространялись в индекс. Затем начните процесс заполнения индекса. Он выполнится в фоновом режиме, поэтому требуется некоторое время для его завершения, прежде чем запросы смогут полноценно использовать индекс.

Файл примера: *Search/soln/microsoft/start.sql*

```
EXEC sp_fulltext_table 'Bugs' , 'start_change_tracking'
EXEC sp_fulltext_table 'Bugs' , 'start_background_updateindex'
EXEC sp_fulltext_table 'Bugs' , 'start_full'
```

Наконец, создайте запрос с использованием оператора `CONTAINS()`:

Файл примера: *Search/soln/microsoft/search.sql*

```
SELECT * FROM Bugs WHERE CONTAINS(summary, '"crash"' );
```

Текстовый поиск в СУБД PostgreSQL

СУБД PostgreSQL 8.3 предлагает продвинутый и высококонфигурируемый способ преобразования текста в доступные для поиска коллекции лексических элементов и поиска по этому тексту при помощи шаблонов.

Чтобы максимально улучшить производительность, необходимо хранить контент в исходной текстовой форме, а также в доступной для поиска форме, используя специальный тип данных `TSVECTOR`.

Файл примера: *Search/soln/postgresql/create-table.sql*

```
CREATE TABLE Bugs (
   bug_id SERIAL PRIMARY KEY,
   summary VARCHAR(80),
   description TEXT,
   ts_bugtext TSVECTOR
   -- другие столбцы
);
```

Убедитесь, что столбец `TSVECTOR` сохранен в синхронизированном с контентом текстовом столбце(ах), который вы хотите сделать доступным для поиска. СУБД PostgreSQL представляет встроенный триггер, упрощающий данную задачу:

Файл примера: *Search/soln/postgresql/trigger.sql*

```
CREATE TRIGGER ts_bugtext BEFORE INSERT OR UPDATE ON Bugs
FOR EACH ROW EXECUTE PROCEDURE
```

```
   tsvector_update_trigger(ts_bugtext, 'pg_catalog.english' ,
summary, description);
```

Также вы должны создать *обобщенный инвертированный индекс* (GIN, generalized inverted index) столбца `TSVECTOR`:

Файл примера: *Search/soln/postgresql/create-index.sql*

```
CREATE INDEX bugs_ts ON Bugs USING GIN(ts_bugtext);
```

После этого вы можете использовать оператор текстового поиска СУБД PostgreSQL — `@@`, эффективность поиска которого повышает полнотекстовый индекс:

Файл примера: *Search/soln/postgresql/search.sql*

```
SELECT * FROM Bugs WHERE ts_bugtext @@ to_tsquery('crash' );
```

Существует множество других вариантов пользовательской настройки доступного для поиска контента, поисковых запросов и результатов поиска.

Полнотекстовый поиск в СУБД SQLite

Стандартные таблицы СУБД SQLite не поддерживают эффективный полнотекстовый поиск (FTS, Full-text search), однако вы можете использовать опциональное расширение для SQLite, чтобы хранить доступный для поиска текст в *виртуальной таблице*, предназначенной для поиска текста. Существует три версии доступного для поиска текстового расширения — FTS1, FTS2 и FTS3.

FTS-расширения не включаются в стандартную сборку СУБД SQLite, поэтому вам придется найти сборку со встроенным FTS-расширением. Например, добавьте следующие настройки в файлы `Makefile.in` и встройте в SQLite.

Файл примера: *Search/soln/sqlite/makefile.in*

```
TCC += -DSQLITE_CORE=1

TCC += -DSQLITE_ENABLE_FTS3=1
```

Как только у вас появляется сборка SQLite с поддержкой FTS-расширений, вы можете создать виртуальную таблицу для доступного для поиска текста. Типы данных, ограничения и другие возможности столбца будут игнорироваться.

Файл примера: *Search/soln/sqlite/create-table.sql*

```
CREATE VIRTUAL TABLE BugsText USING fts3(summary, description);
```

Если вы индексируете текст из другой таблицы (как в примере ниже, где используется таблица Bugs), необходимо скопировать данные в виртуальную таблицу. Виртуальная таблица FTS всегда содержит столбец первичного ключа, называемый docid, поэтому вы можете связывать строки со строками исходной таблицы.

Файл примера: *Search/soln/sqlite/insert.sql*

```
INSERT INTO BugsText (docid, summary, description)
   SELECT bug_id, summary, description FROM Bugs;
```

Теперь вы можете делать FTS-запросы в виртуальной таблице BugsText, используя эффективный предикат полнотекстового поиска MATCH, а также объединять с этим поиск соответствия строк в исходной таблице Bugs. Использование названия FTS-таблицы как псевдостолбца позволяет сопоставить шаблон с любым столбцом.

Файл примера: *Search/soln/sqlite/search.sql*

```
SELECT b.* FROM BugsText t JOIN Bugs b ON (t.docid = b.bug_id)
WHERE BugsText MATCH 'crash' ;
```

Сопоставление с шаблоном также поддерживает некоторые булевы выражения.

Файл примера: *Search/soln/sqlite/search-boolean.sql*

```
SELECT * FROM BugsText WHERE BugsText MATCH 'crash -save' ;
```

Поисковые системы других производителей

Если вам необходимо искать текст таким способом, который работает одинаково вне зависимости от производителя вашей СУБД, вам нужна поисковая система, работающая независимо от базы данных SQL. Данный раздел кратко описывает два таких продукта, как Sphinx и Apache Lucene.

Поисковая система Sphinx

Sphinx (**www.sphinxsearch.com/**) — это поисковая система с открытым исходным кодом, хорошо интегрируемая с СУБД MySQL и PostgreSQL. К моменту написания этих строк уже существовал неофициальный патч для использования Sphinx с СУБД с открытым исходным кодом Firebird. Возможно, в будущем эта поисковая система будет поддерживаться другими СУБД.

Индексация и поиск в поисковой системе Sphinx являются быстродействующими, данная система также поддерживает распределенные запросы. Это

хороший выбор для высокомасштабируемых поисковых приложений, содержащих нечасто обновляемые данные.

Вы можете использовать поисковую систему Sphinx для индексации данных, хранимых в базе данных MySQL. Изменив несколько полей в файле конфигурации `sphinx.conf`, вы можете специализировать базу данных. Вы должны также написать SQL-запрос, чтобы выбрать данные для создания индекса. Первый столбец в этом запросе — целочисленный первичный ключ. Вы можете сделать некоторые столбцы атрибутами ограничения или сортировки результатов. Оставшиеся столбцы будут включены в полнотекстовый индекс. Наконец, другой SQL-запрос выбирает полную строку из базы данных при условии, что значение первичного ключа записано как `$id`.

Файл примера: *Search/soln/sphinx/sphinx.conf*

```
source bugsrc
{
   type  = mysql
   sql_user         = bugsuser
   sql_pass         = xyzzy
   sql_db   = bugsdatabase
   sql_query        = \
      SELECT   bug_id,   status,   date_reported,   summary,
description \
         FROM Bugs
   sql_attr_timestamp      = date_reported
   sql_attr_str2ordinal    = status
   sql_query_info          = SELECT * FROM Bugs WHERE bug_id =
$id
}

index bugs
{
   source    = bugsrc
   path      = /opt/local/var/db/sphinx/bugs
}
```

Как только вы примените данную конфигурацию в файле `sphinx.conf`, вы сможете создать индекс в программной оболочке при помощи команды индексатора:

Файл примера: *Search/soln/sphinx/indexer.sh*

```
indexer -c sphinx.conf bugs
```

Вы можете выполнять поиск по индексу, используя команду `search`:

Файл примера: *Search/soln/sphinx/search.sh*

```
search -b "crash -save"
```

В поисковой системе Sphinx также существуют так называемые демоны (фоновые процессы) и интерфейс прикладного программирования (API, Application Programming Interface), предназначенный для активации поиска в таких популярных языках сценариев, как PHP, Perl и Ruby. Главный недостаток текущего программного обеспечения заключается в том, что индексный алгоритм не поддерживает должным образом инкрементные обновления. Использование системы Sphinx для поиска по источнику данных требует некоторых компромиссов. Например, разбейте доступную для поиска таблицу на две: в первой будет храниться большинство накопленных данных, которые не изменяются, во второй — более узкий набор текущих данных, которые время от времени растут и требуют повторной индексации. В итоге ваше приложение должно пользоваться двумя индексами при поиске с помощью системы Sphinx.

Apache Lucene

Lucene (**lucene.apache.org**) — это зрелая поисковая система для Java-приложений. Работающие подобным образом проекты существуют и для других языков, таких как C++, C#, Perl, Python, Ruby и PHP.

Поисковая система Lucene встраивает индекс в свой собственный формат для коллекции текстовых *документов*. Индекс Lucene не ставится в синхронизацию с источником данных, которые он индексирует. Если вы вставляете, удаляете или обновляете строки в базе данных, вы должны применять соответствующие изменения в индексе Lucene.

Использование поисковой системы Lucene немного напоминает использование автомобильного механизма; вам требуется совсем немного технологий для поддержки его работы, но их вполне достаточно, чтобы сделать его полезным. Система Lucene не берет коллекции данных непосредственно из базы данных SQL, вам необходимо вписать документы в индекс Lucene. Например, вы могли бы выполнить SQL-запрос и для каждой строки результата создать по одному документу Lucene и сохранить их в индексе Lucene. Вы можете использовать систему Lucene через ее Java-API-интерфейс.

К счастью, производитель Apache также предлагает дополнительный проект Solr (**lucene.apache.org/solr/**). Solr — это поисковый сервер, предоставляющий шлюз индексу Lucene. Вы можете создавать документы Solr и передавать поисковые запросы, используя интерфейс, схожий с интерфейсом REST, то есть вы можете использовать Solr с любым языком программирования.

Вы также можете заставить Solr подключаться непосредственно к базе данных, выполнять запросы и индексировать результаты при помощи инструмента `DataImportHandler`.

Попробуйте сами

Предположим, что вы не хотите использовать специально-разработанные функции поиска и устанавливать независимые поисковые системы. Вы нуждаетесь в эффективном, не зависящем от базы данных решении, как сделать текст доступным для поиска. В этом разделе мы будем проектировать *инвертированный индекс*. Как правило, инвертированный индекс — это список слов, которые можно было бы искать. В отношениях типа «множество — множество» индекс связывает слова с введенным текстом, содержащим соответствующее слово. То есть такое слово, как crash, может появиться во многих ошибках, а каждая ошибка может соответствовать множеству других ключевых слов. Данный раздел показывает, как проектировать инвертированный индекс.

Во-первых, задайте таблицу `Keywords`, перечисляющую ключевые слова, которые ищут пользователи, а так же таблицу `BugsKeywords`, устанавливающую связь «множество — множество»:

Файл примера: *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)
);
```

Во-вторых, добавьте в таблицу BugsKeywords по строке для каждого ключевого слова, соответствующего тексту описания данной ошибки. Можно использовать запрос по подстрочному сопоставлению, чтобы обнаружить соответствия при помощи оператора LIKE или регулярных выражений. Это не более дорогостоящий метод, чем наивный метод поиска, описанный в разделе «Антипаттерн», но он более эффективен, поскольку поиск необходимо выполнить только один раз. Если сохранить результат в перекрестной таблице, все последующие поиски того же ключевого слова будут проходить намного быстрее.

Затем напишите хранимую процедуру, чтобы облегчить поиск данного ключевого слова[1]. Если это слово уже искали, запрос будет работать быстрее, так как строки таблицы BugsKeywords представляют собой список документов, содержащих ключевые слова. Если ранее никто не искал данное ключевое слово, поиск будет производиться по коллекции текстовых вводов более трудным способом.

Файл примера: *Search/soln/inverted-index/search-proc.sql*

```
CREATE PROCEDURE BugsSearch(keyword VARCHAR(40))
BEGIN
   SET @keyword = keyword;

① PREPARE s1 FROM 'SELECT MAX(keyword_id) INTO @k FROM Keywords
      WHERE keyword = ?';
   EXECUTE s1 USING @keyword;
   DEALLOCATE PREPARE s1;
   IF (@k IS NULL) THEN

② PREPARE s2 FROM 'INSERT INTO Keywords (keyword) VALUES (?)' ;
   EXECUTE s2 USING @keyword;
   DEALLOCATE PREPARE s2;

③ SELECT LAST_INSERT_ID() INTO @k;

④ PREPARE s3 FROM 'INSERT INTO BugsKeywords (bug_id, keyword_
id)
      SELECT bug_id, ? FROM Bugs
      WHERE summary REGEXP CONCAT('' [[:<:]]'' , ?, '' [[:>:]]''
)
```

[1] Приведенная здесь в качестве примера хранимая процедура использует синтаксис СУБД MySQL.

```
            OR description REGEXP CONCAT('' [[:<:]]'' , ?, ''
[[:>]]'' )';
      EXECUTE s3 USING @k, @keyword, @keyword;
      DEALLOCATE PREPARE s3;
   END IF;

⑤  PREPARE s4 FROM 'SELECT b.* FROM Bugs b
      JOIN BugsKeywords k USING (bug_id)
      WHERE k.keyword_id = ?';
   EXECUTE s4 USING @k;
   DEALLOCATE PREPARE s4;

END
```

① Поиск указанного пользователем ключевого слова. Возвращает или целочисленный первичный ключ из столбца `Keywords.keyword_id` или значение `NULL`, если слово не встречалось ранее.

② Если слово не было найдено, оно будет вставлено в таблицу `Keywords` как новое ключевое слово.

③ Запрос значения первичного ключа будет сгенерирован в таблице `Keywords`.

④ При поиске в таблице `Bugs`, содержащей строки с новым ключевым словом, заполняется перекрестная таблица.

⑤ Наконец, производится запрос полных строк из таблицы `Bugs`, соответствующих идентификационному номеру ключевого слова из столбца `keyword_id`, отображающий было ли найдено ключевое слово или оно должно быть вставлено как новое ключевое слово.

Теперь можно вызвать эту хранимую процедуру и ввести желаемое ключевое слово. Процедура возвращает набор соответствующих ошибок, вне зависимости от того, приходится ли ей, вычислив соответствующие ошибки, заполнить перекрестную таблицу для нового ключевого слова или просто воспользоваться результатом более раннего поиска.

Файл примера: *Search/soln/inverted-index/search-proc.sql*

```
CALL BugsSearch('crash' );
```

У этого решения есть другая сторона: вам придется задать триггер для заполнения перекрестной таблицы, поскольку в нее будет вставляться каждая новая ошибка. Если вы нуждаетесь в поддержке редактирования описаний ошибок, вы также должны будете написать триггер для повторного анализа текста и добавления или удаления строк в таблице `BugsKeywords`.

Файл примера: *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('[[:<:]]' , k.keyword,
'[[:>:]]' )
            OR NEW.summary REGEXP CONCAT('[[:<:]]' , k.keyword,
'[[:>:]]' );
END
```

Список ключевых слов заполняется автоматически, когда пользователи выполняют поиск, поэтому нет необходимости самостоятельно заполнять список каждым словом, найденным в статьях базы знаний. С другой стороны, если можно предугадать вероятный ключевые слова, можно легко выполнить поиск этих слов, тем самым неся расходы времени на первый поиск, а не сваливая их на пользователей.

Я использовал инвертированный индекс для своего приложения базы знаний, описанного в начале этой главы. Я также расширил таблицу `Keywords`, добавив в нее столбец `num_searches`. Я увеличивал этот столбец каждый раз, когда пользователи искали данное ключевое слово, прослеживая тем самым наибольший спрос на ключевые слова.

ВНИМАНИЕ!

Не нужно использовать SQL для решения всех проблем.

Enita non sunt multiplicanda praeter necessitatem.
(лат. — «Не следует множить сущее без необходимости»).

Уильям Оккам

ГЛАВА 18. ЗАПУТАННЫЙ ЗАПРОС

Ваш начальник находится на телефонной линии с его начальником и сигналит вам, чтобы вы пришли. Он закрывает трубку рукой и шепчет вам: «Исполнители находятся на деловой встрече, они собираются сократить наш штат, если мы не предоставим вице-президенту статистику, доказывающую, что наш отдел предоставляет занятость большому количеству людей. Мне нужно знать, над каким количеством продуктов мы работаем, как много разработчиков занимается исправлением ошибок, каково среднее количество ошибок, исправляемых одним разработчиком, и как много было исправлено ошибок, о которых сообщили пользователи. Сейчас же!»

Вы бросаетесь к своему рабочему месту и начинаете писать запрос. Вы хотите получить все ответы сразу, поэтому создаете комплексный запрос в надежде снизить количество повторной работы и получить результат быстрее.

Файл примера: *Spaghetti-Query/intro/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;
```

Возвращенные цифры кажутся неправильными. Как мы получили такое множество продуктов? Как может среднее число исправленных ошибок быть равно 1,0? И нет числа пользователей; только число ошибок, о которых сообщили пользователи. Как могут все эти цифры так не соответствовать друг другу? Этот запрос окажется намного сложнее, чем выдумали.

Ваш начальник кладет трубку. «Не важно», — вздыхает он, — «Слишком поздно. Давайте собирать вещи с наших рабочих мест».

18.1. ЦЕЛЬ: УМЕНЬШЕНИЕ SQL-ЗАПРОСОВ

Один из самых распространенных моментов, на которых застревают SQL-программисты, когда они говорят: «Как я могу выполнить все одним запросом?». Этот вопрос может прозвучать практически для любой задачи. Программисты учились тому, что SQL-запросы являются сложными, комплексными и затратными. По той же логике они рассуждают, что два SQL-запроса — это в два раза хуже. Более двух SQL-запросов для решения проблемы обычно вообще не рассматривается.

Программисты не могут уменьшить комплексность своего задания, но хотят упростить решение. Они описывают свою цель такими эпитетами, как «элегантный» или «эффективный», и полагают, что достигли этой цели, решая задачу при помощи единственного запроса.

18.2. АНТИПАТТЕРН: РЕШЕНИЕ СЛОЖНОЙ ПРОБЛЕМЫ ЗА ОДИН ШАГ

Язык SQL очень выразителен — вы можете выполнить множество действий в единственном запросе или операторе. Но это не значит, что так делать обязательно или что идея решать все задачи одной строкой кода является хорошей. Вы имеете такую привычку с любым другим языком программирования, который вы используете? Вероятно, нет.

Неверные результаты

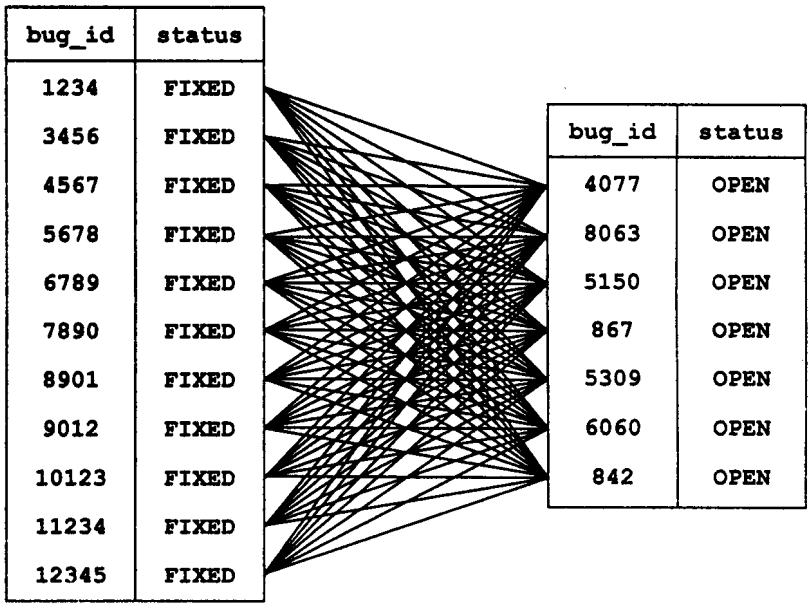
Одним общим последствием вывода всех ваших результатов в одном запросе является *декартово (прямое) произведение*. Такое случается, когда две таблицы в запросе не имеют условия, ограничивающего их связь. Без такого ограничения при объединении этих таблиц каждая строка в первой таблице объединяется с *каждой* строкой во второй таблице. Каждое такое объединение создает строку набора результатов, и вы получаете тот же итог для множества других строк, вместо того, что ожидали.

Посмотрите на пример. Предположим, нужно сделать запрос в базе данных, чтобы подсчитать число исправленных и неисправленных ошибок для данного продукта. Многие программисты попытались бы использовать запрос, подобный следующему, чтобы получить искомые цифры:

Файл примера: *Spaghetti-Query/anti/cartesian.sql*

```
SELECT p.product_id,
   COUNT(f.bug_id) AS count_fixed,
   COUNT(o.bug_id) AS count_open
FROM BugsProducts p
LEFT OUTER JOIN Bugs f ON (p.bug_id = f.bug_id AND f.status =
```

```
'FIXED' )
LEFT OUTER JOIN Bugs o ON (p.bug_id = o.bug_id AND o.status =
'OPEN' )
WHERE p.product_id = 1
GROUP BY p.product_id;
```



bug_id	status
1234	FIXED
3456	FIXED
4567	FIXED
5678	FIXED
6789	FIXED
7890	FIXED
8901	FIXED
9012	FIXED
10123	FIXED
11234	FIXED
12345	FIXED

bug_id	status
4077	OPEN
8063	OPEN
5150	OPEN
867	OPEN
5309	OPEN
6060	OPEN
842	OPEN

Рис. 18.1. Декартово произведение между значениями исправленных и неисправленных ошибок

Так получилось, что вы знаете, что в действительности число исправленных ошибок для данного продукта соответствует 12, а неисправленных — 7. Поэтому результат запроса озадачивает:

product_id	count_fixed	count_open
1	84	84

Почему результат настолько неточен? В данном случае не имеет значения, что 84 — это произведение 12 и 7. В данном примере значения таблицы `Products` объединяются с двумя различными подмножествами значений таблицы `Bugs`, и это заканчивается декартовым произведением между вторыми двумя наборами значений. Каждая из 12 строк исправленных (*FIXED*) ошибок объединяется с каждой из 7 строк неисправленных (*OPEN*) ошибок.

Декартово произведение представлено схематично на рис. 18.1. Каждая линия, соединяющая исправленную ошибку с неисправленной, становится строкой в промежуточном наборе результатов (прежде чем будет произведена группировка). Модно увидеть промежуточный набор результатов, убрав оператор `GROUP BY` из агрегатных функций.

Файл примера: *Spaghetti-Query/anti/cartesian-no-group.sql*

```
SELECT p.product_id, f.bug_id AS fixed, o.bug_id AS open
FROM BugsProducts p
JOIN Bugs f ON (p.bug_id = f.bug_id AND f.status = 'FIXED' )
JOIN Bugs o ON (p.bug_id = o.bug_id AND o.status = 'OPEN' )
WHERE p.product_id = 1;
```

В данном запросе выражены только связи между таблицами `Bugs-Products` и каждым подмножеством таблицы `Bugs`. Нет условий, ограничивающих сопоставление исправленных ошибок с неисправленными, поэтому запрос производится по умолчанию. В результате мы получаем 12 раз по 7 строк.

Случайно получить декартово произведение при попытке создать запрос «два в одном» очень просто. Если вы попытаетесь выполнить больше несвязанных задач в одном запросе, результат может дать большее количество декартовых произведений.

Как если бы этого было мало...

Помимо факта, что вы можете получить неверные результаты, важно осознавать, что эти запросы попросту трудно написать, трудно изменить и трудно отладить. Вы должны ожидать, что получите обычные требования для инкрементных расширений приложений вашей базы данных. Менеджеры требуют более сложных отчетов и больше полей в пользовательском интерфейсе. Если вы проектируете запутанные, объединенные в один SQL-запросы, то их расширение будет более дорогостоящим и затратным по времени. Ваше время чего-то стоит, как для вас, так и для вашего проекта.

Время выполнения запроса тоже имеет свою цену. Оптимизация и выполнение сложного SQL-запроса, использующего много объединений, коррелированных подзапросов и других операций, более трудны для движка SQL, чем работа с простым запросом. Программисты имеют мнение, что выполнение меньшего числа запросов лучше сказывается на производительности. С другой стороны, цена выполнения одного запроса-монстра может расти в геометрической прогрессии, в отличие от использования нескольких более простых запросов.

18.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Если вы слышите следующие утверждения от членов вашего проекта, вы сможете распознать случай использования запутанного запроса:

- «Почему полученные мной суммы и подсчеты невероятно большие?»

Произошло незапланированное вычисление СУБД прямого произведения двух различных объединенных набора данных.

- «Я работал над этим гигантским SQL-запросом целый день!»

Язык SQL в действительности не так труден. Если вы боролись с одним запросом слишком долго, вам необходимо пересмотреть свой подход.

- «Мы не можем ничего добавить к отчету базы данных, потому что уйдет слишком много времени на повторное написание кода для SQL-запроса».

Программист, написавший код запроса, должен будет постоянно поддерживать этот код, даже если запросы переводятся в другой проект. Этим программистом могли быть вы, так что не пишите чрезмерно сложный SQL-код, который никто не сможет поддерживать.

- «Попробуй вставить в запрос еще один оператор `DISTINCT`».

Компенсируя взрыв строк при декартовом произведении, программисты уменьшают частоту копирования, используя ключевое слово `DISTINCT`, как модификатор запроса и модификатор агрегатной функции. Это скрывает уродливость запроса, но принуждает РСУБД к дополнительной работе: генерировать промежуточный набор результатов, только чтобы отсортировать и удалить копии.

Другой сигнал, подсказывающий, что запрос, возможно, является запутанным, — чрезмерно долгое время выполнения. Низкая производительность также является симптоматикой и других причин, но поскольку вы исследуете такой вид запросов, вам следует предположить, что вы попытались выполнить слишком много действий в одном SQL-операторе.

18.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Наиболее частая причина, почему приходится выполнять комплексные задания в одном запросе, заключается в том, что программисты используют программный каркас (framework) или библиотеку визуальных компонентов, которые подключаются к источнику данных и представляют данные в приложении. Простая система анализа деловых данных и инструменты составления отчетов также относятся к этой категории, несмотря на то, что программное обеспечение системы анализа деловых данных может объединять результаты нескольких источников данных.

Компонент или инструмент составления отчетов, предполагающие, что источник данных в одном SQL-запросе может иметь простое использование, поощряют вас к проектированию слитных запросов, чтобы объединить все данные в одном отчете. Если вы используете одно из этих приложений составления отчетов, вам, возможно, придется писать более комплексный SQL-запрос, чем вы можете себе позволить, чтобы сохранить возможность обработки набора результатов.

Если требуются слишком сложные отчеты, чтобы удовлетворить их одним SQL-запросом, то было бы лучше создать несколько отсчетов. Если вашему начальнику это не понравится, напомните ему или ей о взаимосвязи между комплексностью отчета и временем, которое потребуется на его создание.

Иногда может возникнуть необходимость в комплексном результате в одном запросе, потому что вам нужно, чтобы все результаты были объединены в сортированном порядке. В SQL-запросе порядок сортировки устанавливается легко. Вероятно, будет более эффективно, если это сделает СУБД, чем если вам придется писать пользовательский код в вашем приложении, чтобы рассортировать результаты нескольких запросов.

18.5. РЕШЕНИЕ: РАЗДЕЛЯЙ И ВЛАСТВУЙ

Цитата Уильяма Оккама, приведенная в начале этой главы, также известна, как *закон экономии*:

ЗАКОН ЭКОНОМИИ

Когда имеется две конкурирующие между собой теории, приводящие к одинаковому результату, более простая является наилучшей.

Для SQL это значит: когда существует выбор между двумя запросами, производящими один и тот же набор результатов, выберите самый простой. Необходимо помнить об этом при использовании экземпляров данного антипаттерна.

Один шаг за раз

Итак, вы не можете увидеть логическое условие объединения между таблицами, включенными в декартово произведение, которое могло было быть получено просто потому, что такового условия не было. Чтобы избежать прямого произведения, необходимо разделить запутанный запрос на несколько более простых запросов. В простом примере, показанном выше, нам нужно только два запроса:

Файл примера: *Spaghetti-Query/soln/split-query.sql*

```
SELECT p.product_id, COUNT(f.bug_id) AS count_fixed
FROM BugsProducts p
LEFT OUTER JOIN Bugs f ON (p.bug_id = f.bug_id AND f.status =
'FIXED' )
WHERE p.product_id = 1
GROUP BY p.product_id;

SELECT p.product_id, COUNT(o.bug_id) AS count_open
FROM BugsProducts p
LEFT OUTER JOIN Bugs o ON (p.bug_id = o.bug_id AND o.status =
'OPEN' )
WHERE p.product_id = 1
GROUP BY p.product_id;
```

Результаты двух этих запросов — 12 и 7, как и ожидалось.

Возможно, вы будете чувствовать небольшое сожаление, обращаясь к «неэлегантному» решению, разделяя запрос на несколько меньших запросов, но это чувство быстро сменится облегчением, так как вы поймете, что такое решение имеет положительный эффект для разработки, обслуживания и производительности.

- Запрос не создает нежелательное декартово произведение, как происходило в предыдущих примерах, таким образом, легче убедиться, что запрос вернул точные результаты.
- Когда появляются новые требования к отчету, легче добавить другой простой запрос, чем интегрировать большое количество вычислений в и без того сложном запросе.
- Движок SQL обычно оптимизирует и выполняет простые запросы более легко и надежно, чем комплексные. Даже если кажется, что работа дублируется при разделении запроса, все же это может привести к маленькой победе.
- При анализе кода или во время сеанса обучения товарищей по команде легче объяснить, как работают прямые запросы, чем растолковывать принцип работы одного сложного запроса.

Ищите метку UNION

Можно объединить результаты нескольких запросов в один набор результатов при помощи операции `UNION`. Это может быть полезно, если вы хотите использовать один запрос и получить один набор результатов, например, потому что результат должен быть отсортирован.

Файл примера: *Spaghetti-Query/soln/union.sql*

```
(SELECT p.product_id, f.status, COUNT(f.bug_id) AS bug_count
   FROM BugsProducts p
   LEFT OUTER JOIN Bugs f ON (p.bug_id = f.bug_id AND f.status
= 'FIXED' )
   WHERE p.product_id = 1
   GROUP BY p.product_id, f.status)

UNION ALL

(SELECT p.product_id, o.status, COUNT(o.bug_id) AS bug_count
   FROM BugsProducts p
   LEFT OUTER JOIN Bugs o ON (p.bug_id = o.bug_id AND o.status
= 'OPEN' )
   WHERE p.product_id = 1
   GROUP BY p.product_id, o.status)

ORDER BY bug_count;
```

В результате получены результаты каждого подзапроса, конкатенированные в один. Этот пример содержит две строки — по одной для каждого подзапроса. Не забудьте создать столбец, отличающий результаты подзапросов друг от друга, в данном примере это столбец `status`.

Используйте операцию `UNION` только когда столбцы в обоих подзапросах совместимы. Вы не сможете изменить номер, название или тип данных столбцов на полпути в наборе результатов, поэтому убедитесь, что столбцы применяются ко всем строкам последовательно и адекватно. Если вы замечаете, что столбцы имеют псевдонимы, похожие на этот — `bugcount_or_customerid_or_null`, значит, вы, вероятно, применяете операцию `UNION` к результатам запроса, являющимся несовместимыми.

Решение проблемы вашего начальника

Как выполнить настоятельную просьбу о статистике вашего проекта? Ваш начальник сказал: «Мне нужно знать, над каким количеством продуктов мы работаем, как много разработчиков занимается исправлением ошибок, каково среднее количество ошибок, исправляемых одним разработчиком, и как много было исправлено ошибок, о которых сообщили пользователи».

Лучшим решением будет разделение работы.

- Количество продуктов:

Файл примера: *Spaghetti-Query/soln/count-products.sql*

```
SELECT COUNT(*) AS how_many_products
FROM Products;
```

- Количество разработчиков, исправляющих ошибки:

Файл примера: *Spaghetti-Query/soln/count-developers.sql*

```
SELECT COUNT(DISTINCT assigned_to) AS how_many_developers
FROM Bugs
WHERE status = 'FIXED' ;
```

- Среднее число ошибок, исправляемых одним разработчиком:

Файл примера: *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 нами ошибок сообщили пользователи:

Файл примера: *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 довольно хитры. Попытка объединить их в один запрос стала бы кошмаром.

Автоматическое написание SQL-кода: с помощью SQL

Когда вы разбиваете комплексный SQL-запрос, вы можете получить множество очень похожих запросов, различающихся разве что значениями данных. Написание этих запросов — тяжелая работа, являющаяся хорошим подспорьем генерации кода.

Генерация кода — процесс написания кода, результатом которого является новый код, который можно скомпилировать или выполнить. Это может быть ценным, если написание кода вручную является трудоемким. Генератор кода может избавить вас от однообразной работы.

МУЛЬТИТАБЛИЧНЫЕ ОБНОВЛЕНИЯ

Во время консультационной работы меня попросили решить срочную проблему, связанную с SQL, для менеджера в другом отделе.

Я отправился в офис менеджера и обнаружил измотанного коллегу, находящегося в явно безысходном положении. Мы только успели обменяться приветствиями, как он начал делиться со мной своей бедой. «Я очень надеюсь, что вы сможете решить эту проблему быстро; наша система учета запасов была отключена *весь день*». Он не был поклонником SQL, но сказал, что работал в течение нескольких часов над оператором, который должен был обновить большой набор строк.

Его проблема состояла в том, что он не мог использовать согласованное SQL-выражение в операторе `UPDATE` для всех значений строк. Фактически значение, которое он должен был установить, в каждой строке было разным. В его базе данных содержалась информация о материально-технических ресурсах для компьютерной лаборатории и об использовании каждого компьютера. Он хотел установить столбец `last_used`, в котором отображалась бы дата каждого использования компьютера.

Он был слишком сосредоточен на решении этой комплексной задачи при помощи единственного SQL-оператора, что является еще одним примером антипаттерна **Запутанный запрос**. За все эти часы он изо всех сил пытался написать идеальное предложение `UPDATE`, возможно, производя изменения вручную.

Вместо того чтобы написать один SQL-оператор, чтобы создать комплексное обновление, я написал сценарий, который генерирует набор более простых операторов SQL, имеющих требуемый эффект:

Файл примера: *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;
```

Данный запрос выводит серию полных, с точками и запятыми, операторов `UPDATE`, готовых работать как SQL-сценарии:

```
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;
    ...
```

Таким методом я за минуты решил проблему, с которой боролся менеджер долгие часы.

Выполнение большого количества SQL-запросов или операторов, возможно, не самый эффективный способ решения задачи. Однако необходимо совмещать стремление к эффективности с целью решения задачи.

ВНИМАНИЕ!

Несмотря на то что при работе с SQL создается ощущение, что можно решать комплексные проблемы одной строкой кода, не поддавайтесь искушению построить карточный домик.

Как я могу сказать, что я думаю, до того как понял, что сказал?

Э.М. Форстер

ГЛАВА 19. СКРЫТЫЕ СТОЛБЦЫ

PHP-программист попросил помочь ему разобраться в запутанном результате казалось бы простого SQL-запроса базы данных его библиотеки:

Файл примера: *Implicit-Columns/intro/join-wildcard.sql*

```
SELECT * FROM Books b JOIN Authors a ON (b.author_id = a.
author_id);
```

Этот запрос вернул все книжные заголовки со значением NULL. Даже посторонний, выполняющий тот или иной запрос без слияния какой-либо таблицы с таблицей Authors, получил бы результат, включающий действительные книжные заголовки, как и ожидалось.

Я помог ему найти причину его проблемы: расширение базы данных PHP, которую он использовал, возвратило каждую строку результата SQL-запроса как ассоциативный массив. Например, он мог обратиться к столбцу Books.isbn через элемент массива $row[«isbn»]. В обеих таблицах Books и Authors есть столбец title (последними заголовками были такие, как Dr. или Rev.). Элемент массива одного результата $row[«title»] может хранить только одно значение; в такой ситуации столбец Authors.title занял этот элемент массива. У большинства авторов в базе данных отсутствовало значение в столбце title, поэтому в результате элемент массива $row[«title»] получает значение NULL. Если же в запросе отсутствует объединение таблицы Books с таблицей Authors, то конфликт между названиями столбцов не возникает, и заголовки книг занимают ячейки массива согласно ожиданиям.

Я сказал программисту, что решение состоит в том, чтобы присвоить столбцам псевдонимы, чтобы одинаковые столбцы имели разные названия, а следовательно, и различные записи в массиве.

Файл примера: *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);
```

Тогда он задал второй вопрос: «Как мне присвоить одному столбцу псевдоним и одновременно запрашивать другие столбцы?» Он хотел продолжить использование подстановочных знаков (SELECT *) и присваивать псевдоним столбцу, используемому в шаблоне.

19.1. ЦЕЛЬ: УМЕНЬШЕНИЕ ОБЪЕМА КЛАВИАТУРНОГО ВВОДА

Создается ощущение, что разработчикам программного обеспечения не нравится печатать, что делает выбор их карьеры забавным, как поворот событий в концовке рассказа О. Генри.

Ниже приведен пример, на который программисты ссылаются при необходимости в чрезмерном клавиатурном наборе при перечислении всех столбцов в SQL-запросе:

Файл примера: *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 программного обеспечения с удовольствием используют функцию *подстановочных знаков* в SQL. Символ * означает *каждый столбец*, поэтому список столбцов становится скрытым. Это помогает сделать запросы более краткими.

Файл примера: *Implicit-Columns/obj/select-implicit.sql*

```
SELECT * FROM Bugs;
```

Аналогично, кажется разумным воспользоваться преимуществами оператора INSERT: значения по умолчанию присваиваются ко всем столбцам в порядке их определения в таблице.

Файл примера: *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' ), NULL, 49.95);
```

Вместо внесения столбцов в список будет короче написать оператор.

Файл примера: *Implicit-Columns/obj/insert-implicit.sql*

```
INSERT INTO Accounts VALUES (DEFAULT,
   'bkarwin' , 'Bill' , 'Karwin' , 'bill@example.com' ,
SHA2('xyzzy' ), NULL, 49.95);
```

19.2. АНТИПАТТЕРН: ЗАПУТЫВАЮЩИЕ КОМБИНАЦИЯ КЛАВИШ

Несмотря на то что использование подстановочных знаков и скрытых столбцов удовлетворяет цели сокращения клавиатурного набора, эта привычка все же имеет некоторые риски.

Срыв рефакторинга

Предположим, вам необходимо добавить столбец в таблицу `Bugs`, например, столбец `date_due` для внесения целей.

Файл примера: *Implicit-Columns/anti/add-column.sql*

```
ALTER TABLE Bugs ADD COLUMN date_due DATE;
```

Теперь ваш оператор `INSERT` заканчивается ошибкой, поскольку вы перечислили 11 значений вместо 12 значений ожидаемых таблицей.

Файл примера: *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: Column count doesn't match value count at
row 1
```

В операторе `INSERT`, содержащем скрытые столбцы, вы должны дать значения всех столбцов в том же порядке, в котором находятся столбцы в таблице. Если столбцы меняются местами, оператор выдает ошибку или даже присваивает значения неправильным столбцам.

Допустим, вы выполняете запрос типа `SELECT *` и, так как вы не знаете названий столбцов, ссылаетесь на столбцы, основываясь на их порядковой позиции:

Файл примера: *Implicit-Columns/anti/ordinal.php*

```
<?php
$stmt = $pdo->query("SELECT * FROM Bugs WHERE bug_id = 1234");
$row = $stmt->fetch();
$hours = $row[10];
```

Но вы не знаете, что другой член команды удалил столбец:

Файл примера: *Implicit-Columns/anti/drop-column.sql*

```
ALTER TABLE Bugs DROP COLUMN verified_by;
```

Столбец `hours` больше не в позиции 10. Ваше приложение по ошибке использует значения из другого столбца. По мере того как столбцы будут переименовываться, добавляться или удаляться, результат вашего запроса будет меняться таким образом, который не поддерживает ваш код. Вы не сможете предугадать, сколько столбцов будет возвращено в запросе, если будете использовать подстановочные знаки.

Эти ошибки могут распространиться через весь код, и к тому времени, когда вы заметите проблему в выводе приложения, будет уже трудно проследить по цепочке, где произошла ошибка.

Скрытые стоимости

Удобство использования подстановочных знаков в запросах может вредить производительности и масштабируемости. Чем больше столбцов обрабатывает ваш запрос, тем больше данных должно пройти по сети между вашим приложением и сервером базы данных.

Вы, вероятно, имеете много одновременно выполняющихся запросов в среде вашего приложения. Они конкурируют за пропускную способность сети. Даже гигабитная сеть может переполняться сотней клиентов приложений, выполняющих запросы тысяч строк одновременно.

В объектно-реляционном отображении (ORM, Object-relational mapping) так же, как и в Активной записи (Active Record) часто используются запросы типа `SELECT *` по умолчанию, чтобы заполнить поля объекта, представленного в виде строки в базе данных. Даже если ORM предлагает отменить это действие, большинство программистов не беспокоится.

Вы просили — вы получили

Один из самых распространенных вопросов, задаваемых мне программистами, использующими подстановочные знаки SQL: «Существует ли сокращение, с помощью которого я могу запросить все столбцы, кроме нескольких, указанных мной?» Возможно, эти программисты пытаются избежать затрат ресурсов на выборку громоздких текстовых столбцов, в которых они не нуждаются, но желают при этом комфортных условий использования подстановочных знаков.

Ответом будет «нет». Язык SQL не поддерживает синтаксис, означающий «получить все столбцы, которые я хочу, но исключить те столбцы, которых я не хочу». Либо вы используете подстановочный знак, чтобы выделить все столбцы в таблице, либо вы сами перечисляете столбцы, которые вам нужны.

19.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Следующие сценарии могут подсказать, что в вашем проекте неуместно используются скрытые столбцы, что создает проблему:

- «Приложение прерывает работу, потому что продолжает указывать старые имена столбцов в наборе результатов базы данных. Мы попытались обновить весь код, но, я полагаю, мы что-то упустили».

Вы изменили таблицу в базе данных, добавив, удалив, переименовав столбцы или изменив их порядок, но вам не удалось изменить прикладной код, ссылающийся на таблицу. Отслеживать все эти ссылки — кропотливая работа.

- «Нам потребовались дни, чтобы отыскать узкое место нашей сети, и мы наконец свели его к чрезмерному трафику сервера базы данных. Согласно нашей статистике средний запрос требует более 2 Мб данных, но выводит лишь десятую часть от этого объема».

Вы берете в выборку больше данных, чем вам нужно.

19.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Оправданным использованием подстановочных знаков является особый случай в SQL, когда вы пишете быстрые запросы, чтобы проверить решение или провести диагностическую проверку текущих данных. Единожды используемый запрос не принесет особой выгоды от удобства управления.

В примерах этой книги подстановочные знаки используются, чтобы не занимать много места и не отвлекать читателя от более интересных частей примеров запросов. Я редко использую подстановочные знаки SQL в производственном прикладном коде.

Если для вашего приложения существует необходимость выполнять запросы, адаптирующиеся к добавлению, удалению, переименованию или перестановке столбцов, вы можете обнаружить, что лучше всего для этого подойдет использование подстановочных знаков. Убедитесь, что запланировали дополнительную работу, которая потребуется для отслеживания ловушек, описанных выше.

Вы можете использовать подстановочные знаки для каждой таблицы индивидуально в запросе с соединением. Поставьте подстановочный знак перед названием таблицы или псевдонимом. Это позволит вам указать короткий список выбранных вами из одной таблицы столбцов, в то время как с помощью подстановочного знака будут выбраны все столбцы из другой таблицы. Например:

Файл примера: *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);
```

Клавиатурный набор длинного списка названий столбцов может отнимать много времени. Для некоторых программистов эффективность разработки более важна, чем эффективность времени выполнения. Аналогично, вы могли бы поставить приоритет на написании запросов, что сделает их более краткими и улучшит читаемость. Использование подстановочных знаков действительно снижает объем клавиатурного ввода и позволяет сделать запрос короче — если таков ваш приоритет, то пользуйтесь подстановочными знаками.

Однажды я услышал, как разработчик утверждал, что длинный SQL-запрос при передаче от приложения до сервера базы данных требует слишком много сетевых издержек. Теоретически длина запроса в некоторых случаях может иметь значение. Однако более распространена ситуация, при которой возвращаемые запросом строки данных используют гораздо больше пропускной способности сети, чем ваша строка SQL-запроса. Используйте свое суждение для исключительных случаев, но не потейте над маленьким объемом материала.

19.5. РЕШЕНИЕ: ИМЕНОВАНИЕ СТОЛБЦОВ В ЯВНОМ ВИДЕ

Всегда перечисляйте все столбцы, в которых вы нуждаетесь, вместо того чтобы полагаться на подстановочные знаки или скрытые списки столбцов.

Файл примера: *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;
```

Файл примера: *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);
```

Весь этот клавиатурный ввод кажется обременительным, но он того стоит по нескольким причинам.

Защита от ошибок

Помните о так называемой защите от дурака (пока-екэ)?[1] Вы должны сделать SQL-запросы более устойчивыми к ошибкам и путанице, описанным ранее при указании столбцов в списке выборки запроса.

- Если столбец был переставлен в таблице, он не изменит позицию в результате запроса.
- Если столбец был добавлен в таблицу, он не появится в результате запроса.
- Если столбец был удален из таблицы, то запрос спровоцирует ошибку, но это хорошая ошибка, поскольку будет указан непосредственно код, который вам необходимо будет исправить, вместо того чтобы отправляться на поиски первопричины.

Вы получите аналогичные преимущества, когда будете указывать столбцы в операторе `INSERT`. Порядок столбцов, которые вы указываете, заменяет порядок в определении таблицы, и значения присваиваются назначенным вами столбцам. Только что добавленные столбцы, еще не названные вами в операторе, получают значения по умолчанию или значение `NULL`. Если вы ссылаетесь на столбец, который был удален, вы получите ошибку, но ее устранение является более простым.

Это пример принципа *ошибись раньше* (*fail early*).

То, что вам не понадобится

Если вы обеспокоены масштабируемостью и пропускной способностью вашего программного обеспечения, вам необходимо искать возможные причины расточительного использования пропускной способности сети. Во время разработки и тестирования программного обеспечения пропускная способность SQL-запроса может казаться безвредной, но вы можете столкнуться с проблемой, когда ваша производственная среда выполняет тысячи SQL-запросов в секунду.

Как только вы отказываетесь от использования подстановочного знака SQL, вы естественным образом получаете мотивацию не учитывать ненужные столбцы, поскольку это означает меньше клавиатурного набора, что, в свою очередь, повышает эффективность использования пропускной способности.

Файл примера: *Implicit-Columns/soln/yagni.sql*

```
SELECT date_reported, summary, description, resolution, status,
priority
FROM Bugs;
```

[1] Практика японской индустрии проектирования систем защиты от ошибок. См. главу 5.

В любом случае необходимо прекратить использование подстановочных знаков

Когда вы покупаете пакет конфет M&M's в торговом автомате, обертка — это удобство, помогающее в целости донести пакет с конфетами к вашему столу. Однако когда вы открываете пакет, вам необходимо обработать каждую конфету M&M's индивидуально. Они катаются, скользят и отскакивают по всему пакету. Если вы не будете осторожны, некоторые из них упадут со стола и создадут ошибки. Но нельзя съесть хотя бы одну конфету, пока вы не раскроете пакет.

В SQL-запросе, когда вы хотите применить выражение к столбцу, использовать псевдоним столбца или исключить столбцы из запроса ради повышения эффективности, вы должны раскрыть «контейнер» в виде подстановочного знака. Вы теряете удобство обработки коллекции столбцов как одного пакета, но вы получаете доступ ко всему их содержимому.

Вам неизбежно придется обрабатывать некоторые столбцы в запросе индивидуально, используя псевдоним столбца, функцию или удаление столбца из списка. Если вы пропустите использование подстановочных знаков с самого начала, то в будущем ваш запрос будет легче изменять.

ВНИМАНИЕ!

Берите все, что хотите, но съедайте все, что берете.

ЧАСТЬ IV. АНТИПАТТЕРНЫ РАЗРАБОТКИ ПРИЛОЖЕНИЙ

Враг знает систему.

Принцип Керкгоффса
в формулировке Шеннона

ГЛАВА 20. СЧИТЫВАЕМЫЕ ПАРОЛИ

Предположим, вы получили телефонный звонок от человека, использующего одно из поддерживаемых вами приложений. У позвонившего проблема со входом в приложение.

«Это Пэт Джонсон из отдела продаж. Я, по всей видимости, забыл свой пароль. Не могли бы вы найти его и сообщить мне?» — голос Пэта был робким, но и подозрительно спешащим.

«Сожалею, но я не могу этого сделать, — отвечаете вы. — Я могу сбросить вашу учетную запись, и пароль придет вам по электронной почте, на которую она зарегистрирована. Вы сможете использовать команды в пришедшем письме, чтобы установить новый пароль».

Позвонивший становится более нетерпеливым и агрессивным. «Это смешно, — говорит он. — В последней компании, в которой я работал, штат поддержки мог найти мой пароль. Неужели вы не можете выполнить свою работу? Вы хотите, чтобы я вызвал вашего руководителя?»

Естественно, вы хотите сохранить гладкие отношения с вашими пользователями, поэтому вы запускаете SQL-запрос, чтобы найти текстовый пароль от учетной записи Пэта Джонсона и прочитать его ему по телефону.

Человек кладет трубку. Вы говорите своему сотруднику: «Я был в опасном положении. У меня только что чуть не произошел конфликт с Пэтом Джонсоном. Я надеюсь, он не пожалуется».

Ваш коллега выглядит озадаченным: «*Он*? Пэт Джонсон из отдела продаж — женщина. Мне кажется, вы только что дали ее пароль мошеннику».

20.1. ЦЕЛЬ: ВОССТАНОВЛЕНИЕ ИЛИ СБРОС ПАРОЛЕЙ

Готов поставить на то, что в любом приложении, имеющем пароли, кто-то из пользователей обязательно забудет свой пароль. Самые современные приложения решают эту проблему, предоставляя пользователю шанс вос-

становить или сбросить пароль через почтовый механизм обратной связи. Это решение зависит от того, имеет ли пользователь доступ к адресу электронной почты, связанному с пользовательским профилем в приложении.

20.2. АНТИПАТТЕРН: ХРАНЕНИЕ ПАРОЛЕЙ В ОТКРЫТОЙ ТЕКСТОВОЙ ФОРМЕ

Частая ошибка в подобных способах восстановления пароля заключается в том, что положение позволяет пользователю запрашивать электронное письмо, содержащее пароль в текстовом виде. Эта огромная трещина в безопасности связана с проектированием базы данных, она подвергает определенному риску безопасность приложения, позволяя неавторизованным пользователям получать привилегированный доступ к приложению.

Хранение паролей

Пароль, как правило, сохраняется в таблице `Accounts` в столбце атрибутов строки:

Файл примера: *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
);
```

Вы можете создать учетную запись, просто добавив одну строку и определив пароль как литерал строки:

Файл примера: *Passwords/anti/insert-plaintext.sql*

```
INSERT INTO Accounts (account_id, account_name, email, password)
VALUES (123, 'billkarwin' , 'bill@example.com' , 'xyzzy' );
```

Небезопасно хранить пароль в открытом текстовом виде, равно как и передавать его по сети. Если недоброжелатель может прочесть SQL-оператор, который вы используете, чтобы вставлять пароль, он может ясно увидеть пароль. Это также относится к SQL-операторам, с помощью которых производится смена пароля или проверка соответствия ввода пользователя с сохраненным паролем. У взломщиков есть несколько возможностей украсть пароли, включая следующие.

- Прерывание отправки пакетов, когда SQL-оператор отправляет их от клиента приложения на сервер базы данных. Это более просто, чем звучит; существуют бесплатные программные инструменты, например Wireshark[1].
- Поиск журнала SQL-запросов на сервере базы данных. Взломщик должен войти на хост сервера базы данных, после того, как он это сделает, он может получить доступ к журналам событий, которые могут содержать записи об SQL-операторах, выполняемых данным сервером базы данных.
- Чтение данных из резервных копий базы данных на сервере или на носителе резервных копий. Ваши носители резервных копий хранятся в безопасности? Вы уничтожаете безвозвратно данные с ваших носителей перед переработкой или перенесением для других целей.

Аутентификация паролей

Затем, когда пользователь пытается войти под своей учетной записью, ваше приложение сравнивает введенный пользователем пароль и пароль в строке, хранящейся в базе данных. Это сопоставление производится нешифрованным текстом, поскольку пароль также хранится в виде открытого текста. Например, вы можете использовать следующий запрос, чтобы вернуть 0 (ложь) или 1 (истина), что показывает соответствует ли ввод пользователя паролю в базе данных:

Файл примера: *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 данном примере пароль (*opensesame*), введенный пользователем, является неверным, и запрос возвращает значение 0.

Как показано в разделе выше, изменение введенной пользователем строки в SQL-запрос открытым текстом предоставляет взломщикам возможность увидеть эти данные.

НЕ СМЕШИВАЙТЕ ДВА РАЗЛИЧНЫХ УСЛОВИЯ

Почти всегда я сталкиваюсь с условиями аутентификационных запросов для столбцов `account_id` и `password` в операторе `WHERE`:

[1] Программа Wireshark (ранее известная как Ethereal) доступна на сайте http://www.wireshark.org/.

Файл примера: *Passwords/anti/auth-lumping.sql*

```
SELECT * FROM Accounts
WHERE account_name = 'bill' AND password = 'opensesame' ;
```

Этот запрос возвращает пустой набор результатов, если учетная запись не существует или пользователь ввел неверный пароль. Ваше приложение не может отделить друг от друга две причины неудавшейся аутентификации. Лучше использовать запрос, который сможет обрабатывать два этих случая отдельно. Затем вы сможете исправить произошедшую ошибку.

Например, представим, что вы хотите временно блокировать учетную запись, после того как обнаружили множество неудавшихся попыток входа, поскольку это указывает на вероятную попытку вторжения. Однако вы не можете найти нужный шаблон, поскольку не находите отличий между неверным именем учетной записи и неверным паролем.

Отправка паролей по электронной почте

Поскольку пароль сохранен незашифрованным текстом в базе данных, отыскать его в вашем приложении просто:

Файл примера: *Passwords/anti/select-plaintext.sql*

```
SELECT account_name, email, password
FROM Accounts
WHERE account_id = 123;
```

Тогда ваше приложение может выслать электронное письмо по запросу пользователя. Вы, вероятно, видели подобные письма, являющиеся частью функции напоминания пароля любого веб-сайта, которым вы пользовались. Пример такого вида писем приведен ниже:

ПРИМЕР ЭЛЕКТРОННОГО ПИСЬМА «ВОССТАНОВЛЕНИЕ ПАРОЛЯ»

```
От кого: daemon
Кому: bill@example.com
Тема: Восстановление пароля

Вы сделали запрос на напоминание пароля для учетной записи
"bill".
Ваш пароль: "xyzzy".

Перейдите по ссылке, чтобы войти в свою учетную запись:

http://www.example.com/login
```

Отправка по электронной почте пароля в открытом текстовом виде несет в себе серьезную угрозу безопасности. Электронное письмо может быть перехвачено, взломано или сохранено взломщиками множеством способов. Недостаточно того, что вы используете безопасный протокол для просмотра почты или что серверы получения/отправки писем управляются ответствен-

ными системными администраторами. Поскольку электронное письмо отправляется через Интернет, оно может быть перехвачено на других сайтах. Безопасные протоколы электронной почты распространены не повсеместно, и не все они находятся под вашим контролем.

20.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Любое приложение, которое может восстановить ваш пароль и отправить его вам, должно хранить пароли в открытом текстовом виде или, по крайней мере, с некоторым количеством обратимого кода. Это — антипаттерн. Если ваше приложение может читать пароль правомерным образом, то есть вероятность, что взломщик сможет прочитать этот пароль неправомерно.

20.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Вашему приложению, возможно, необходимо использовать пароль, чтобы обратиться к другому сервису, относящемуся к третьей стороне, то есть ваше приложение может быть клиентом. В таком случае вы должны хранить этот пароль в читаемом формате. Предпочтителен вариант, при котором вы использовали бы шифрование, которое ваше приложение могло бы раскодировать, чтобы использовать открытый текст в базе данных.

ЭТИКА РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Если вы разрабатываете приложение, поддерживающее пароли, и вас просят спроектировать функцию восстановления пароля, вы должны вежливо отклонить эту просьбу, предупредив работающих над проектом о рисках, и предложить альтернативное решение, предоставляющее схожие возможности.

Так же как электрик должен распознавать и корректировать проектирование электропроводки, имеющей риск пожароопасности, ваш долг, как инженера программного обеспечения, — знать о проблемах безопасности и разрабатывать безопасное программное обеспечение.

Хорошая книга, которую вы должны прочитать, — «19 Deadly Sins of Software Security» [10]. Также массу полезной информации вы можете найти на ресурсе «Open Web Application Security Project» (**owasp.org**).

Вы можете найти различия между понятиями *идентификация* и *аутентификация*. Пользователь может идентифицировать себя кем угодно, но аутентификация подтвердит, тот ли он, за кого себя выдает. Пароли — самый распространенный способ сделать это.

Если вы не в состоянии обеспечить достаточно высокую безопасность, чтобы победить квалифицированных и решительных взломщиков, то вам эффективнее иметь механизм идентификации, чем ненадежный механизм аутентификации. Но это не будет нарушением сделки.

Не каждое приложение находится под угрозой атаки, и не каждое приложение содержит важную информацию, которая должна быть защищена. Например, к веб-приложению может присоединиться только небольшое число проверенных людей, являющихся честными и отзывчивыми. В этом случае идентификационного механизма будет достаточно для работы с приложением, и в этой неофициальной обстановке более простой вход в систему может быть вполне адекватным.

Дополнительная работа, необходимая для создания мощной системы аутентификации, может быть неоправданной.

Все же будьте внимательны — приложения имеют тенденцию развиваться за пределы их исходной среды или роли. Прежде чем вы сделаете свое оригинальное небольшое веб-приложение доступным за пределами сетевой защиты вашей компании, вы должны отдать его на проверку квалифицированному эксперту по безопасности.

20.5. РЕШЕНИЕ: ХРАНЕНИЕ ПАРОЛЯ В ВИДЕ БЕСПОРЯДОЧНОГО НАБОРА СИМВОЛОВ

Главная проблема этого антипаттерна состоит в том, что исходный вид пароля читаем. Но вы можете аутентифицировать пользователя по введенному паролю, не читая его. Данный раздел описывает, как реализовать такой вид безопасного хранения паролей в базе данных SQL.

Понимание хеш-функций

Закодируйте пароль, используя одностороннюю *криптографическую хеш-функцию*. Эта функция преобразует свою входную строку в новую, называемую *хеш-кодом*, по которой невозможно прочесть исходную. Даже длину оригинальной строки определить невозможно, поскольку хеш-код, возвращаемый хеш-функцией, является строкой фиксированной длины. Например, алгоритм `SHA-256` преобразует приведенный в качестве примера пароль *xyzzy* в 256-разрядную строку, обычно представляемую как 64-символьная строка с символами шестнадцатеричной системы счисления:

```
SHA2('xyzzy') = '184858a00fd7971f810848266ebcecee5e8b69972c5f
faed622f5ee078671aed'
```

Другой характеристикой хеш-кода является его необратимость. Вы не сможете восстановить входную строку по ее хеш-коду, так как алгоритм хеширования спроектирован таким образом, чтобы «потерять» некоторую часть информации о вводе. Хороший алгоритм хеширования должен занимать столько работы для взлома, сколько он бы взял для простого предположения ввода подбором.

В прошлом популярным алгоритмом был алгоритм SHA-1, но исследователи недавно доказали, что этот 160-разрядный алгоритм хеширования имеет недостаточно высокую криптостойкость; существует техника вывода из хеш-строки. Эта техника отнимает много времени, однако не так много, как подбор паролей. Национальный Институт стандартов и технологий (NIST, National Institute of Standards and Technology) объявил о плане прекращения поддержки алгоритма SHA-1, как безопасного алгоритма хеширования, после 2010 года пользу следующих более стойких вариантов: SHA-224,

SHA-256, SHA-384 и SHA-512[1]. Нуждаетесь вы в применении стандартов NIST или нет, было бы хорошо использовать, по крайней мере, функцию SHA-256 для хеширования паролей.

MD5() — другая популярная хеш-функция, создающая 128-разрядные строки хеш-кода. Данная функция также была названа криптографически слабой, поэтому вы не должны использовать ее для кодирования паролей. Более слабые алгоритмы по-прежнему используются, но не для такой важной информации, как пароли.

Использование хеширования в SQL

Ниже представлено преобразование таблицы Accounts. Хеш-код алгоритма SHA-256 пароля всегда имеет длину 64 символа, поэтому определите столбец как столбец CHAR фиксированной длины в 64 символа.

Файл примера: *Passwords/soln/create-table.sql*

```
CREATE TABLE Accounts (
   account_id      SERIAL PRIMARY KEY,
   account_name    VARCHAR(20),
   email           VARCHAR(100) NOT NULL,
   password_hash   CHAR(64) NOT NULL
);
```

Хеш-функции не являются частью стандартного языка SQL, поэтому вам, возможно, придется положиться на модель вашей СУБД, поддерживает ли она хеширование как расширение. Например, СУБД MySQL 6.0.5, поддерживающая протокол SSL, содержит функцию SHA2(), возвращающую 256-разрядный хеш-код по умолчанию.

---

[1] csrc.nist.gov/groups/ST/toolkit/secure_hashing.html.

Файл примера: *Passwords/soln/insert-hash.sql*

```
INSERT INTO Accounts (account_id, account_name, email, password_
hash)
   VALUES (123, 'billkarwin' , 'bill@example.com' , SHA2('xyzzy'
));
```

Вы можете проверить корректность ввода пользователя, применяя эту же хеш-функцию к нему и сравнивая результат со значением, сохраненным в базе данных.

Файл примера: *Passwords/soln/auth-hash.sql*

```
SELECT CASE WHEN password_hash = SHA2('xyzzy' ) THEN 1 ELSE 0
END
   AS password_matches
FROM Accounts
WHERE account_id = 123;
```

Вы можете легко блокировать учетную запись, изменяя значение в строке хеш-кода пароля на то, которое хеш-функция не может возвратить. Например, строка *noaccess* содержит символы, которые не являются символами шестнадцатеричной системы счисления.

Добавление соли в хеш-код

Если вы храните пароли в виде хеш-кода и злоумышленник получил доступ к вашей базе данных (например, найдя в вашем мусоре диск CD-ROM с резервными копиями), он может попытаться подобрать пароли. Подбор каждого пароля может занять много времени, однако злоумышленник может создать свою собственную базу данных хеш-кода вероятных паролей, чтобы сравнивать с ним строки хеш-кода, найденные им в вашей базе данных. Если хотя бы один пользователь выбрал пароль, являющийся словом из словаря злоумышленника, ему будет легко это определить, ища в вашей базе данных паролей хеш-код, соответствующий коду в его специальной таблице. Он даже может сделать это с помощью SQL:

Файл примера: *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;
```

Существует способ избежать такого вида «словарной атаки» — включение соли в ваше выражение кодирования пароля. *Соль* — это строка бессмысленных байт, которые вы конкатенируете с паролем пользователя, прежде чем ввести результирующую строку в хеш-функцию. Даже если пользователь выбрал слово из словаря в качестве пароля, хеш-код, созданный из пароля с солью, не будет соответствовать хеш-коду в базе данных злоумышленника. Например, если пароль — *password*, вы можете увидеть, что его хеш-код отличается от хеш-кода с добавлением нескольких случайных байт:

```
SHA2('password' )
   = '5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef7
21d1542d8'
SHA2('password-0xT!sp9' )
   ='7256d8d7741f740ee83ba7a9b30e7ac11fcd9dbd7a0147f4cc83c62dd
6e0c45b'
```

Каждый пароль должен использовать различные значения соли, чтобы заставить злоумышленника генерировать новую таблицу словаря хеш-кода для каждого пароля. Таким образом, взломщик попадет в ловушку, поскольку взлом паролей в вашей базе данных занимает столько же времени, как их подбор[1].

Файл примера: *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('xyzzy' || '-0xT!sp9' ), '-0xT!sp9' );
```

[1] csrc.nist.gov/groups/ST/toolkit/secure_hashing.html.

```
SELECT (password_hash = SHA2('xyzzy' || salt)) AS password_
matches
FROM Accounts
WHERE account_id = 123;
```

Нормальная длина соли составляет 8 байт. Необходимо генерировать соль произвольным образом для каждого пароля. В предыдущих примерах показано, что строки соли содержат печатные символы, но помните, что вы можете сделать так, чтобы в соли использовались любые случайные непечатные байты.

Сокрытие пароля от SQL

Теперь, когда вы используете мощную хеш-функцию для того, чтобы закодировать пароль, прежде чем сохранить его, а также соль, чтобы помешать взлому пароля по словарю, вы думаете, что этого достаточно, чтобы гарантировать безопасность. Однако пароли все еще всплывают в открытом текстовом виде в SQL-выражениях, это означает, что они читаемы, если взломщик может перехватить сетевые пакеты или если SQL-запросы регистрируются, а журналы записей попадают не в те руки.

Вы можете защититься от подобного рода угроз, если не будете помещать пароли в SQL-запрос в читаемом виде. Вместо этого вычислите хеш-код в вашем прикладном коде и используйте только его в SQL-запросах. Таким образом, если взломщику удастся перехватить хеш-код, он не сможет раскодировать его, чтобы получить пароль.

Вам действительно необходимо добавление соли перед вычислением хеш-кода.

Ниже приведен PHP-пример, в котором используется расширение PDO для получения соли, вычисления хеш-кода и выполнения запроса, чтобы проверить правильность пароля по хеш-коду с солью, хранящегося в базе данных:

Файл примера: *Passwords/soln/auth-salt.php*

```
<?php

$password = 'xyzzy' ;

$stmt = $pdo->query(
      "SELECT salt
      FROM Accounts
      WHERE account_name = 'bill' ");
```

```
$row = $stmt->fetch();
$salt = $row[0];

$hash = hash('sha256' , $password . $salt);

$stmt = $pdo->query("
      SELECT (password_hash = '$hash' ) AS password_matches;
      FROM Accounts AS a
      WHERE a.acct_name = 'bill' ");

$row = $stmt->fetch();
if ($row === false) {
      // account 'bill' does not exist
} else {
   $password_matches = $row[0];
   if (!$password_matches) {
      // password given was incorrect
   }
}
```

Функция `hash()` гарантированно возвращает только символы шестнадцатеричной системы счисления, таким образом, нет рисков при внедрении SQL-кода (см. главу 21).

Когда речь идет о веб-приложениях, у взломщиков существует другая возможность перехватить данные в сети: между браузером пользователя и сервером веб-приложения. Когда пользователь вводит форму регистрации, браузер посылает его пароль в открытом текстовом виде на сервер, где вычисляется хеш-код, как описано ранее. Вы можете защититься от подобной ситуации, переводя пароль в хеш-код еще в браузере пользователя перед отправлением данных. Но это затруднительно, поскольку вам придется отыскать соль, связанную с введенным паролем, прежде, чем вы сможете вычислить корректный хеш-код. Хорошим компромиссом будет являться использование безопасного HTTP-соединения при отправке пароля от браузера к приложению.

Сброс пароля вместо его восстановления

Теперь, когда пароль сохранен более безопасным способом, вам по-прежнему остается решить начальную цель: помочь пользователям, забывшим пароль. Вы не можете восстанавливать пароли, поскольку теперь ваша база данных хранит хеш-код вместо пароля. Вы не можете восстановить

пароль по хеш-коду более легким образом, чем взломщик. Но вы можете предоставить пользователю доступ иным путем. Здесь приведены два основных способа.

Первая альтернатива заключается в том, что когда пользователь, забывший пароль, просит о помощи, вместо того чтобы высылать ему его пароль, приложение может выслать электронное письмо с временным паролем, сгенерированным приложением. Для дополнительной безопасности срок действия такого пароля может быть коротким, поэтому если письмо было перехвачено, скорее всего, несанкционированный доступ не удастся. Кроме того, приложение должны быть спроектировано таким образом, чтобы пользователь был вынужден сменить пароль при первом заходе в приложение.

ПРИМЕР ЭЛЕКТРОННОГО ПИСЬМА СО СГЕНЕРИРОВАННЫМ СИСТЕМОЙ ВРЕМЕННЫМ ПАРОЛЕМ

```
От кого: daemon
Кому: bill@example.com
Тема: Сброс пароля

Вы сделали запрос на сброс пароля для вашей учетной записи.

Ваш временный пароль: "p0trz3b1e".
Этот пароль перестанет быть действительным по истечении одного
часа.

Перейдите по ссылке, чтобы войти в свою учетную запись и уста-
новить новый пароль:

http://www.example.com/login
```

Во второй альтернативе, вместо того чтобы выслать новый пароль по электронной почте, запрос регистрируется в таблице базы данных и ему назначается уникальный маркер в качестве идентификатора:

Файл примера: *Passwords/soln/reset-request.sql*

```
CREATE TABLE PasswordResetRequest (
   token    CHAR(32) PRIMARY KEY,
   account_id      BIGINT UNSIGNED NOT NULL,
   expiration      TIMESTAMP NOT NULL,
   FOREIGN KEY     (account_id)  REFERENCES  Accounts(account_
id)
);
SET @token = MD5('billkarwin' || CURRENT_TIMESTAMP);
```

```
INSERT INTO PasswordResetRequest (token, account_id, expiration)
   VALUES (@token, 123, CURRENT_TIMESTAMP + INTERVAL 1 HOUR);
```

Затем вы высылаете маркер по электронной почте. Вы можете также послать маркер другими способами, например SMS-сообщением, если телефонный номер привязан к учетной записи, нуждающейся в сбросе пароля. Таким образом, если посторонний запрашивает сброс пароля несанкционированно, сообщение придет только фактическому владельцу учетной записи.

ПРИМЕР ЭЛЕКТРОННОГО ПИСЬМА С ВРЕМЕННОЙ ССЫЛКОЙ НА СТРАНИЦУ СБРОСА ПАРОЛЯ

```
От кого: daemon
Кому: bill@example.com
Тема: Сброс пароля

Вы сделали запрос на сброс пароля для вашей учетной записи.

Перейдите по ссылке в течение одного часа, чтобы сменить пароль.
По истечении одного часа ссылка будет недействительна, и ваш па-
роль не будет изменен.

http://www.example.com/reset_password?token=f5cabff22532bd0025
118905bdea50da
```

Когда приложение получает запрос со специальным экраном `reset_password`, значение в параметре маркера должно соответствовать строке в таблице `PasswordResetRequest`, а время истечения в этой строке должно быть предстоящим, а не прошедшим. Идентификационный номер `account_id` в этой строке должен ссылаться на таблицу `Accounts`, поэтому маркер будет ограничен сбросом пароля только одной учетной записи.

Разумеется, если бы посторонние могли получить возможность войти на эту страницу — это могло бы нанести вред. Простые ограничения уменьшают этот риск, например, предоставление специальному экрану короткого периода действия и удостоверение в том, что экран не показывает учетную запись тому, чей пароль устанавливается.

Не создавайте маркер сброса пароля, используя только имя учетной записи и метку текущего времени в файле `reset-request.sql`. Злоумышленник, запускающий механизм сброса пароля для данного пользователя на сайте, будет знать имя пользователя (оно будет предоставлено), а также может предположить, когда будет сделана временная метка. (Согласно протоколу NTP часы на компьютере взломщика будут идти в пределах одной-двух секунд в сравнении с часами сервера базы данных. Даже если в базе данных не используется NTP-протокол, то временной интервал может быть не-

сколько сотен секунд.) Угадывание маркера только с именем пользователя и временной меткой является слишком простым; необходимо использовать гораздо более разнообразные источники энтропии. Пожалуйста, посетите ресурс **cwe.mitre.org/**, чтобы изучить дефект 330 с массой приведенных примеров реальных проблем, вызванных недостаточной энтропией в протоколах.

Криптография постоянно совершенствуется, борясь с технологиями атак. Методы, описанные в этой главе улучшают огромное число типичных приложений, однако если вы должны разработать более защищенные системы, вам нужно перейти к использованию более расширенных техник, таких как следующие:

PBKDF2 (**tools.ietf.org/html/rfc2898**) — широко используемый стандарт *усиления защиты ключей.*

Bcrypt (**bcrypt.sourceforge.net/**) реализует *адаптивные хеш-функции.*

ВНИМАНИЕ!

Если вы можете прочитать пароли — сможет и взломщик.

Процитируйте меня, когда я говорю, что меня неверно процитировали.

Граучо Маркс

ГЛАВА 21. ИНЪЕКЦИЯ SQL-КОДА

В марте 2010 года серийный компьютерный хакер Альберт Гонсалес (Albert Gonzalez) был признан виновным в участии в самом большом хищении конфиденциальной информации в истории. Он украл приблизительно 130 миллионов номеров кредитных и платежных карт, взломав банкоматы и платежные системы нескольких крупнейших сетей розничной продажи, а также компании, обрабатывающие эти кредитные карты.

Гонсалес побил предыдущий рекорд, установленный им же, хищения 45,6 миллионов номеров кредитных и платежных карт в 2006 году. Он совершил это, более раннее преступление, воспользовавшись уязвимостью беспроводных сетей.

Как Гонсалес практически утроил свой предыдущий рекорд? Мы представляем себе отважный сюжет фильма о Джеймсе Бонде, где одетые в черное агенты спускаются по веревке вниз шахты лифта; используя суперкомпьютеры, взламывают зашифрованные при помощи новейших технологий пароли или отключают электроэнергию во всем городе.

Официальное обвинение описывают более приземленную действительность. Гонсалес использовал уязвимое место, являющееся одной из самых распространенных слабостей безопасности в сети Интернет. Он использовал технологию атаки, называемую «инъекция SQL-кода», чтобы получить доступ, дающий привилегию для загрузки файлов на сервера компаний-жертв. После того как Гонсалес и его подельники получили этот доступ, в обвинительном акте констатировалось[1]:

ПРОВЕДЕНИЕ АТАК: ВРЕДОНОСНОЕ ПО

...ими были установлены программы-«снифферы», которые фиксируют номера кредитных и платежных карт, соответствующие им данные о картах и другую информацию, базирующуюся на режиме реального времени, в то время как информация проходит через сети, обрабатывающие кредитные и платежные карты компаний-жертв, а затем периодически передают эту информацию подельникам.

---

[1] Связанная с этим, но более сложная методика восстановления паролей по их хеш-коду называется *радужная таблица*. Использование соли защищает также и от этой техники.

Розничные продавцы, чьи сайты взломал Гонсалес, уверяли, что произвели изменения, корректирующие эти бреши в защите. Однако они залатали только одну щель, в то время как новые веб-приложения создаются каждый день и содержат в себе другие дыры. Атаки с внедрением SQL-кода остаются простой целью для хакеров, потому что разработчики программного обеспечения не понимают суть уязвимого места или как написать код, чтобы предотвратить эту уязвимость.

21.1. ЦЕЛЬ: СОЗДАНИЕ ДИНАМИЧЕСКИХ SQL-ЗАПРОСОВ

Язык SQL предназначен для использования совместно с прикладным кодом. Когда вы встраиваете SQL-запросы как строки и объединяете переменные приложения в одну строку, это обычно называется *динамическим SQL-кодом*[1].

Файл примера: *SQL-Injection/obj/dynamic-sql.php*

```
<?php
$sql = "SELECT * FROM Bugs WHERE bug_id = $bug_id";
$stmt = $pdo->query($sql);
```

Этот простой пример показывает интерполяцию PHP-переменных в строку. Мы описываем значение `$bug_id` как целочисленное так, чтобы к тому времени, когда база данных получит запрос, значение `$bug_id` являлось частью запроса.

Динамические SQL-запросы — это естественный способ получить большее от базы данных. Когда вы используете данные приложения, чтобы обозначить, каким образом вы хотите совершить запрос в базе данных, вы используете SQL как двусторонний язык. Ваше приложение ведет своего рода диалог с СУБД.

Тем не менее не так трудно заставить ваше программное обеспечение выполнять задачи, которые вы хотите; более сложной целью будет сделать его защищенным, чтобы оно не позволяло выполнять действия, которые вы не хотите. Программные дефекты, к которым приведет внедрение SQL-кода, будут исправляться в последнюю очередь.

---

[1] Технически любой запрос, анализируемый во время выполнения, является динамическим SQL-кодом, однако при совместном использовании SQL-код включает данные о переменных.

21.2. АНТИПАТТЕРН: ВЫПОЛНЕНИЕ НЕПРОВЕРЕННЫХ ВХОДНЫХ ДАННЫХ В КАЧЕСТВЕ КОДА

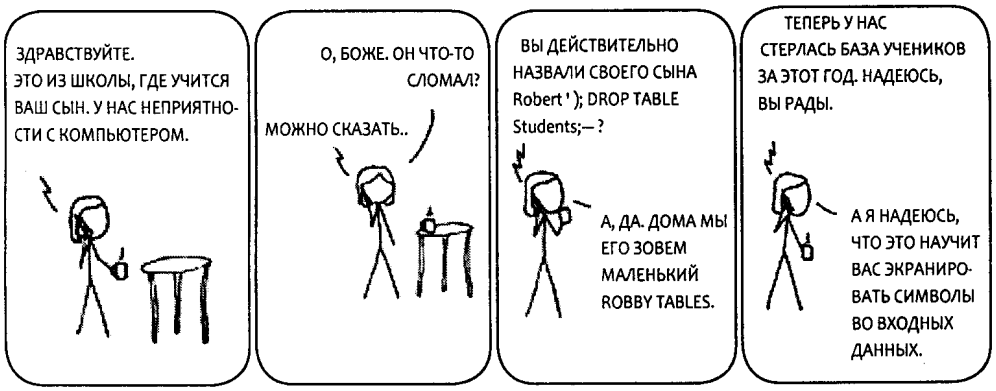


Рис. 21.1: Мамины эксплойты

Инъекция SQL-кода происходит, когда вы интерполируете какой-либо контент в строку SQL-запроса, а он изменяет синтаксис вашего запроса таким образом, который вы не предусмотрели. В классическом примере внедрения SQL-кода значение, которые вы интерполируете в строку, завершает SQL-оператор и выполняет второй законченный оператор. Например, если значением переменной `$bug_id` будет `1234; DELETE FROM Bugs`, то результат SQL-запроса, показанного ранее, выглядел бы следующим образом:

Файл примера: *SQL-Injection/anti/delete.sql*

```
SELECT * FROM Bugs WHERE bug_id = 1234; DELETE FROM Bugs
```

Данный тип внедрения SQL-кода может быть эффектным, как показано на рис. 21.1[1]. Как правило, такие дефекты более тонкие, но по-прежнему опасные.

Несчастные случаи, которые могут произойти

Предположим, вы пишете веб-интерфейс для просмотра базы данных ошибок и на одной из страниц вы можете просмотреть проект с данным ему именем:

Файл примера: *SQL-Injection/anti/ohare.php*

```
<?php
$project_name = $_REQUEST["name"];
```

[1] Комикс Рэндела Манро (Randall Munroe), использован с разрешения автора (**xkcd.ru/327/**).

```
$sql = "SELECT * FROM Projects WHERE project_name = '$proj-
ect_name' ";
```

Неприятности начинаются, когда вы нанимаете группу для разработки программного обеспечения для Международного аэропорта О'Хара в Чикаго. Вы, естественно, даете проекту имя «О'Хара». Каким вы представляете себе запрос для просмотра проекта в вашем веб-приложении?

```
http://bugs.example.com/project/view.php?name=O'Hare
```

Ваш PHP-код берет значение из соответствующего параметра запроса и интерполирует его в SQL-запрос, однако он создает запрос, который не ожидали ни вы, ни пользователь:

Файл примера: *SQL-Injection/anti/ohare.sql*

```
SELECT * FROM Projects WHERE project_name = 'O'Hare'
```

Поскольку строка завершается первой одинарной кавычкой, конечное выражение содержит короткую строку `'O'`, сопровождаемую дополнительными символами `Hare'`, не имеющими в данном контексте никакого смысла. СУБД может только сообщить о синтаксической ошибке. Это простой несчастный случай. Риск пагубных последствий в этом случае мал, поскольку оператор с синтаксической ошибкой не может быть выполнен. Большую угрозу несет ситуация, при которой оператор выполняется без ошибки, но делает что-то непредвиденное.

Главная угроза веб-безопасности

Инъекция SQL-кода становится большой угрозой, когда злоумышленник может использовать данный вид атаки, чтобы управлять вашими SQL-операторами. Например, ваше приложение может позволить пользователю менять его или ее пароль:

Файл примера: *SQL-Injection/anti/set-password.php*

```
<?php
$password = $_REQUEST["password"];
$userid = $_REQUEST["userid"];
$sql = "UPDATE Accounts SET password_hash = SHA2('$password' )
   WHERE account_id = $userid";
```

Умелый взломщик, способный предположить, как параметры запроса используются в вашем SQL-операторе, может отправить тщательно подобранную строку, чтобы воспользоваться ими:

```
http://bugs.example.com/setpass?password=xyzzy&userid=123  OR
TRUE
```

После интерполирования строки из параметра `userid` в ваше SQL-выражение строка меняет синтаксис оператора. Теперь оператор изменяет пароль *каждой* учетной записи в базе данных, а не одной определенной:

Файл примера: *SQL-Injection/anti/set-password.sql*

```
UPDATE Accounts SET password_hash = SHA2('xyzzy' )
WHERE account_id = 123 OR TRUE;
```

Это ключ к пониманию инъекции SQL-кода и к тому, как с ней бороться: внедрение SQL-кода работает, изменяя синтаксис SQL-оператора, прежде чем тот будет синтаксически проанализирован. Пока вы вставляете динамические блоки в оператор перед синтаксическим анализом, у вас есть риск инъекции SQL-кода.

Существует бесчисленное множество способов, с помощью которых злонамеренно выбранная строка может изменить поведение ваших SQL-операторов. Это ограничено только воображением злоумышленника и вашей возможностью защитить SQL-операторы.

Поиск излечения

Теперь, когда мы знаем, чем грозит инъекция SQL-кода, следующим закономерным вопросом будет: что нам необходимо сделать, чтобы защитить код от взлома? Вам, возможно, приходилось читать некую статью в блоге или на сайте, описывающую некоторый единственный метод и утверждающую, что он является универсальным средством против инъекции SQL-кода. В действительности ни один из методов не прошел испытание всеми формами внедрения SQL-кода, поэтому вы должны пользоваться всеми методами в зависимости от ситуации.

Escape-значения

Самый старый способ защиты SQL-запросов от случайных несогласованных кавычек — это использование escape-значений, препятствующих тому, чтобы кавычки становились заключительными, заканчивающими строку. В стандартном языке SQL предусмотрено использование двух видов кавычек, один из которых является литеральным:

Файл примера: *SQL-Injection/anti/ohare-escape.sql*

```
SELECT * FROM Projects WHERE project_name = 'O''Hare'
```

Большинство марок СУБД поддерживает обратную наклонную черту в качестве escape-значения, так же как в большинстве других языков программирования:

Файл примера: *SQL-Injection/anti/ohare-escape.sql*

```
SELECT * FROM Projects WHERE project_name = 'O\'Hare'
```

Идея состоит в том, что вы преобразуете данные, прежде чем интерполировать их в строки SQL. Большинство интерфейсов SQL-программирования обеспечивает функцию удобства. Например, в PHP-расширении PDO используйте функцию `quote()`, чтобы разграничить строку и символы кавычек и тем самым избежать любых литеральных кавычек в пределах строки.

Файл примера: *SQL-Injection/anti/ohare-escape.php*

```
<?php
$project_name = $pdo->quote($_REQUEST["name"]);
$sql = "SELECT * FROM Projects WHERE project_name = $project_
name";
```

Данный метод может уменьшить риск инъекции SQL-кода, появляющийся из-за несогласованных кавычек с динамическим контентом. Но он не работает так же хорошо для бесстрочного контента.

Файл примера: *SQL-Injection/anti/set-password-escape.php*

```
<?php
$password = $pdo->quote($_REQUEST["password"]);
$userid = $pdo->quote($_REQUEST["userid"]);
$sql = "UPDATE Accounts SET password_hash = SHA2($password)
   WHERE account_id = $userid";
```

Файл примера: *SQL-Injection/anti/set-password-escape.sql*

```
UPDATE Accounts SET password_hash = SHA2('xyzzy' )
WHERE account_id = '123 OR TRUE'
```

Невозможно сравнить числовой столбец непосредственно со строкой, содержащей цифры, ни в одной модели СУБД. Некоторые СУБД могут скрытым образом привести строку к целесообразному числовому эквиваленту,

однако в стандартном языке SQL необходимо преднамеренно использовать функцию `CAST()`, чтобы преобразовать строку в числовой тип данных.

Существуют также случаи, заводящие в тупик, в которых строки в наборах символов, не относящихся к ASCII-кодировке, могут пройти через escape-функцию, но оставить кавычки нетронутыми[1].

Параметры запроса

Решение, наиболее часто цитируемое как панацея от инъекций SQL-кода, состоит в использовании *параметров запроса*. Вместо того чтобы интерполировать динамические значения в SQL-строку, оставьте указатель места заполнения параметра в строке во время подготовки запроса. Затем введите значение параметра, когда будете выполнять готовый запрос.

Файл примера: *SQL-Injection/anti/parameter.php*

```
<?php
$stmt = $pdo->prepare("SELECT * FROM Projects WHERE project_
name = ?");
$params = array($_REQUEST["name"]);
$stmt->execute($params);
```

Большинство программистов рекомендуют это решение, поскольку вам не придется использовать escape-значения для динамического контента или волноваться о дефектных escape-функциях. В действительности, параметры запроса — очень сильная защита против инъекции SQL-кода. Но параметры — не универсальное решение, поскольку значение параметра запроса всегда интерпретируется как единственное литеральное значение.

- Ни один список значений не может быть единственным параметром:

Файл примера: *SQL-Injection/anti/parameter.php*

```
<?php
$stmt = $pdo->prepare("SELECT * FROM Bugs WHERE bug_id IN ( ?
)");
$stmt->execute(array("1234,3456,5678"));
```

Это утверждение действует так же, как если бы вы создали единственное строковое значение, состоящее из цифр и запятых, которое не работает по той же причине, что и последовательность целых чисел:

---

[1] Посетите страницу **bugs.mysql.com/8378**, чтобы изучить пример.

Файл примера: *SQL-Injection/anti/parameter.sql*

```
SELECT * FROM Bugs WHERE bug_id IN ( '1234,3456,5678' )
```

- Ни один идентификатор таблицы не может быть параметром:

Файл примера: *SQL-Injection/anti/parameter.php*

```
<?php
$stmt = $pdo->prepare("SELECT * FROM ? WHERE bug_id = 1234");
$stmt->execute(array("Bugs"));
```

Это работает, как если бы вы ввели строковый литерал вместо имени таблицы, что просто является синтаксической ошибкой:

Файл примера: *SQL-Injection/anti/parameter.sql*

```
SELECT * FROM 'Bugs' WHERE bug_id = 1234
```

- Ни один идентификатор столбца не может быть параметром.

Файл примера: *SQL-Injection/anti/parameter.php*

```
<?php
$stmt = $pdo->prepare("SELECT * FROM Bugs ORDER BY ?");
$stmt->execute(array("date_reported"));
```

В данном примере сортировка является холостой командой, потому что выражение является постоянной строкой, аналогично это происходит в каждой строке:

Файл примера: *SQL-Injection/anti/parameter.sql*

```
SELECT * FROM Bugs ORDER BY 'date_reported' ;
```

- Ни одно ключевое слово SQL не может быть параметром.

Файл примера: *SQL-Injection/anti/parameter.php*

```
<?php
$stmt = $pdo->prepare("SELECT * FROM Bugs ORDER BY date_reported
?");
$stmt->execute(array("DESC"));
```

Параметр интерпретируется как литеральная строка, а не ключевое слово SQL. В данном примере результатом является синтаксическая ошибка:

Файл примера: *SQL-Injection/anti/parameter.sql*

```
SELECT * FROM Bugs ORDER BY date_reported 'DESC'
```

КАКОВ БЫЛ МОЙ ПОЛНЫЙ ЗАПРОС?

Многие полагают, что использование параметров SQL-запроса является способом автоматического заключения значений SQL-оператора в кавычки. Это утверждение не является точным, и размышление о параметрах запроса приводит к непониманию того, как они работают.

Сервер РСУБД анализирует ваш запрос в то время, как вы *готовите* запрос. После этого ничто не сможет изменить синтаксис этого SQL-запроса.

Вы задаете значения во время *выполнения* готового запроса. Каждое значение, которое вы задаете, используется для каждого указателя места заполнения один к одному.

Можно выполнить готовый запрос снова, заменяя новыми значениями параметра старые значения. Поэтому РСУБД должна отслеживать запрос и значения параметра отдельно друг от друга. Это хорошо сказывается на сохранении безопасности.

Это значит, что когда вы получаете готовую строку SQL-запроса, она не содержит значений параметра. Было бы удобно видеть SQL-оператор, включающий значения параметра, если вы отлаживаете или регистрируете запросы, но эти значения никогда не объединяются с запросом в его удобочитаемой форме SQL.

Лучший способ отладить динамические SQL-операторы состоит в том, чтобы зарегистрировать как оператор с указателями места заполнения параметра во время подготовки, так и значения параметра во время выполнения.

Хранимые процедуры

Использование хранимых процедур — еще один метод, доказывающий утверждение многих разработчиков программного обеспечения о том, что он защищает от уязвимости инъекции SQL-кода. Как правило, хранимые процедуры содержат фиксированные SQL-операторы, анализируемые, когда вы определяете процедуру.

Однако использовать динамический SQL-код в хранимых процедурах может быть опасно. В следующем примере параметр `input_userid` интерполируется в SQL-запрос дословно, что является небезопасным.

Файл примера: *SQL-Injection/anti/procedure.sql*

```
CREATE PROCEDURE UpdatePassword(input_password VARCHAR(20),
      input_userid VARCHAR(20))
BEGIN
   SET @sql = CONCAT('UPDATE Accounts
      SET password_hash = SHA2(', QUOTE(input_password), ' )
      WHERE account_id = ', input_userid);
   PREPARE stmt FROM @sql;
   EXECUTE stmt;
END
```

Использование динамического SQL-кода в хранимой процедуре безопасно не больше и не меньше, чем использование динамического SQL-кода в прикладном коде. Аргумент `input_userid` может содержать вредоносный контент и создать опасный SQL-оператор:

Файл примера: *SQL-Injection/anti/set-password.sql*

```
UPDATE Accounts SET password_hash = SHA2('xyzzy' )
WHERE account_id = 123 OR TRUE;
```

Каркасы доступа к данным

Вы, возможно, видели защитников каркасов доступа к данным, утверждающих, что их библиотека защищает ваш код от риска инъекции SQL-кода. Это ложное требование к любому каркасу, позволяющему писать SQL-операторы в виде строк.

ПРАКТИКА ХОРОШЕЙ ГИГИЕНЫ

После того как я провел презентацию каркаса доступа к данным, который я разработал, член аудитории подошел ко мне и задал вопрос: «Предотвращает ли ваш каркас инъекцию SQL-кода?». Я ответил, что она предоставляет функции для заключения строк в кавычки и использования параметров запроса.

Молодой человек выглядел озадаченным. «Но может ли она предотвратить инъекцию SQL-кода?» — повторил он. Он искал автоматический способ перестраховать себя, не делает ли он ошибку, которую не знает как распознать.

Я сказал ему, что каркас предотвращает инъекцию SQL-кода так же, как зубная щетка предотвращает дырки в зубах. Необходимо использовать ее соответствующим образом, чтобы извлечь пользу.

Никакой каркас не заставит вас писать безопасный SQL-код. Каркас может обеспечить функции удобства, чтобы помочь вам, но обойти эти функции и вместо них использовать обычный способ манипуляций со строками, чтобы создать небезопасный SQL-оператор, довольно легко.

21.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Фактически каждое приложение СУБД создает SQL-операторы динамическим образом. Если вы создаете какую-либо часть SQL-оператора, конкатенируя строки или интерполируя переменные в строки, то оператор потенциально подвергает ваше приложение риску атак, называемых инъекцией SQL-кода.

ПРАВИЛО № 31: ПРОВЕРЬТЕ ЗАДНЕЕ СИДЕНЬЕ

Если вам нравится смотреть фильмы про монстров, то вы знаете, что существа любят прятаться за сиденьем водителя и набрасываться на него после того, как он

сядет в машину. Урок заключается в том, что нельзя предположить, что нет никакой опасности в таком знакомом вам пространстве, как ваша машина.
Инъекция SQL-кода может принимать косвенную форму. Даже если вы вводите предоставленные пользователем данные, используя параметры запроса безопасным образом, вы можете использовать эти данные позже, во время формирования динамических SQL-запросов.

```
<?php
$sql1 = "SELECT last_name FROM Accounts WHERE account_id
= 123" ;
$row = $pdo->query($sql1)->fetch();
$sql2 = "SELECT * FROM Bugs WHERE MATCH(description) AGAINST
('" . $row["last_name" ] . "')" ;
```

Что произошло бы в предыдущем запросе, если бы пользователь вписал такое имя, как *O'Hara* или если бы было умышленно введено свое имя, чтобы сохранить синтаксис SQL?

Уязвимости перед инъекцией SQL-кода настолько распространены, что разумно предположить, что некоторые ваши приложения, использующие SQL-код, уязвимы, если вы еще не завершили анализ кода, проводимый специально для поиска и исправления этих проблем.

21.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Данный антипаттерн отличается от большинства других в этой книге, в нем нет никаких оправданных причин, чтобы разрешить вашему приложению иметь уязвимость безопасности от инъекции SQL-кода. Наоборот, это ваша обязанность как разработчика программного обеспечения — написать защищенный код и помочь коллегам сделать то же самое. Программное обеспечение защищено настолько, насколько защищено его самое слабое звено — убедитесь, что не вы ответственны за это слабое звено!

21.5. РЕШЕНИЕ: НИКОМУ НЕЛЬЗЯ ДОВЕРЯТЬ

Не существует одного единственного способа защиты SQL-кода. Следует изучить все описанные ниже методы и использовать их в соответствующих случаях.

Фильтруйте входные данные

Вместо того чтобы задаваться вопросом, содержат ли некоторые введенные данные вредоносный контент, необходимо убрать из входных данных все символы, недопустимые для ввода. То есть если вы нуждаетесь в целом числе, используйте только ту часть контента, которая включает целое число. Наилучший способ сделать это зависит от используемого вами языка программирования; например, в PHP используются расширения фильтров:

Файл примера: *SQL-Injection/soln/filter.php*

```
<?php
$bugid = filter_input(INPUT_GET, "bugid", FILTER_SANITIZE_
NUMBER_INT);
$sql = "SELECT * FROM Bugs WHERE bug_id = {$bugid}";
$stmt = $pdo->query($sql);
```

Вы можете использовать функции преобразования типов для таких простых случаев, как числа:

Файл примера: *SQL-Injection/soln/casting.php*

```
<?php
$bugid = intval($_GET["bugid"]);
$sql = "SELECT * FROM Bugs WHERE bug_id = {$bugid}";
$stmt = $pdo->query($sql);
```

Можно также использовать регулярные выражения, чтобы сопоставлять безопасные подстроки, отфильтровывая недопустимый контент:

Файл примера: *SQL-Injection/soln/regexp.php*

```
<?php
$sortorder = "date_reported"; // default

if (preg_match("/[_[:alnum:]]+/", $_GET["order"], $matches)) {
$sortorder = $matches[1];
}

$sql = "SELECT * FROM Bugs ORDER BY {$sortorder}";
$stmt = $pdo->query($sql);
```

Параметризуйте динамические значения

Если динамические компоненты вашего запроса являются простыми значениями, следует использовать параметры запроса, чтобы отделить их от SQL-выражений.

Файл примера: *SQL-Injection/soln/parameter.php*

```
<?php
$sql = "UPDATE Accounts SET password_hash = SHA2(?) WHERE
account_id = ?";
$stmt = $pdo->prepare($sql);
```

```
$params = array($_REQUEST["password"], $_REQUEST["userid"]);
$stmt->execute($params);
```

Вы видели в разделе «Антипаттерн» примеры того, что параметр может замещать только одно значение. Если Вы добавляете значения параметра после того, как РСУБД проанализирует SQL-оператор, то ни одна атака внедрения SQL-кода не сможет изменить синтаксис параметризованного запроса. Даже если взломщик пытается использовать злоумышленное значение параметра такое, как `123 OR TRUE`, РСУБД интерпретирует параметр как значение. В худшем случае запрос не может быть применен к любой строке; он вряд ли применится к неправильным строкам. Злоумышленное значение привело бы к относительно безопасному SQL-оператору, эквивалентному следующему:

Файл примера: *SQL-Injection/soln/parameter.sql*

```
UPDATE Accounts SET password_hash = SHA2('xyzzy' )
WHERE account_id = '123 OR TRUE'
```

Следует использовать параметры запроса, когда необходимо объединить переменные приложения как литеральные значения в SQL-выражениях.

Заключение в кавычки динамических значений

Применение параметров запроса обычно является наилучшим решением, но в редких случаях запрос с указателями места заполнения параметров заставляет оптимизатор запроса принимать лишние решения относительно того, который индекс использовать.

Например, пусть имеется столбец `is_active` в таблице `Accounts`. Он хранит истинные значения для 99% строк, давая неравномерное распределение значений. Запрос `is_active = false` воспользуется индексом, но чтение индекса для запроса `is_active = true` будет затратным. Однако если вы использовали параметр в выражении `is_active = ?`, оптимизатор не может знать, какое значение вы предоставите, когда будете выполнять готовый запрос, следовательно вы выбрали неверный план оптимизации.

В экзотических случаях, подобных этому, было бы лучше интерполировать значения непосредственно в SQL-оператор, несмотря на общую рекомендацию использовать параметры запроса. Если вы делаете это, следует заключить строки в кавычки более аккуратно.

Файл примера: *SQL-Injection/soln/interpolate.php*

```
<?php
$quoted_active = $pdo->quote($_REQUEST["active"]);
```

```
$sql = "SELECT * FROM Accounts WHERE is_active = {$quoted_
active}";
$stmt = $pdo->query($sql);
```

Убедитесь, что используете полностью сформировавшуюся и хорошо протестированную в вопросах безопасности SQL функцию. Большинство библиотек доступа к данным содержат такие функции, заключающие строки в кавычки. Например, в PHP используйте функцию `PDO::quote()`. Не пытайтесь реализовать свою собственную функцию заключения в кавычки, если полностью не изучили угрозы безопасности.

ПАРАМЕТРИЗАЦИЯ В ПРЕДИКАТЕ IN()

Мы уже видели, что невозможно ввести разделенную запятыми строку в единственный параметр. Необходимо использовать столько параметров, каково число элементов в вашем списке.

Например, скажем, вам необходимо запросить шесть ошибок при помощи их первичных ключей, которые находятся в переменной `$bug_list` массива:

```
<?php
$sql = "SELECT * FROM Bugs WHERE bug_id IN (?, ?, ?, ?, ?, ?)");
$stmt = $pdo->prepare($sql);
$stmt->execute($bug_list);
```

Это работает, только если у вас есть точно шесть элементов в переменной `$bug_list`, соответствующих числу указателей места заполнения параметра. Необходимо создать SQL-предикат `IN()` динамическим образом, используя число указателей места заполнения, равное числу элементов переменной `$bug_list`.

Пример, показанный ниже, использует в PHP некоторые встроенные функции массивов для создания указателей места заполнения массивов той же длины, что и переменная `$bug_list`, а затем для присоединения к этому массиву строк при помощи разделения запятыми перед тем, как интерполировать все это в SQL-выражение.

```
<?php
$sql = "SELECT * FROM Bugs WHERE bug_id IN ("
. join(",", array_fill(0, count($bug_list), "?")) . ")";
$stmt = $pdo->prepare($sql);
$stmt->execute($bug_list);
```

Используйте этот метод для параметризации списков значений.

Изолируйте вводимые пользователем данные от кода

Параметры запроса и escape-методы помогают объединять литеральные значения в SQL-выражения, но они не помогут с другими частями оператора, такими как идентификаторы таблиц или столбцов или ключевые слова SQL. Вам необходимо другое решение, чтобы сделать эти компоненты запроса динамичными.

Представьте, что ваши пользователи хотят выбрать как сортировать списки ошибок, например по состоянию или по времени создания. Они также хотят выбрать направление сортировки.

Файл примера: *SQL-Injection/soln/orderby.sql*

```
SELECT * FROM Bugs ORDER BY status ASC
SELECT * FROM Bugs ORDER BY date_reported DESC
```

В следующем примере PHP-сценарий принимает запрошенные параметры `order` и `dir`, а ваш код интерполирует выбор пользователя в SQL-запрос с названием столбца и ключевым словом.

Файл примера: *SQL-Injection/soln/mapping.php*

```
<?php
$sortorder = $_REQUEST["order"];
$direction = $_REQUEST["dir"];
$sql = "SELECT * FROM Bugs ORDER BY $sortorder $direction";
$stmt = $pdo->query($sql);
```

Сценарий предполагает, что порядок содержит название столбца и что параметр `dir` содержит значения *ASC* (ASCending, возрастание) или *DESC* (DESCending, убывание). Это небезопасное предположение, поскольку пользователь может ввести любые значения параметра в веб-запросе.

Вместо этого вы можете использовать параметры запроса для поиска предопределенных значений и затем использовать эти значения в SQL-запросе.

1. Задайте массив `$sortorders`, отображающий варианты выбора пользователей как ключи, а названия SQL-столбцов — как значения. Задайте массив `$directions`, отображающий варианты выбора пользователей как ключи, а ключевые слова SQL — как значения `ASC` и `DESC`.

Файл примера: *SQL-Injection/soln/mapping.php*

```
$sortorders = array( "status" => "status", "date" => "date_
reported" );
$directions = array( "up" => "ASC", "down" => "DESC" );
```

2. Установите в переменных `$sortorder` и `$dir` значение по умолчанию в случае, если варианты выбора пользователей не входят в массивы.

Файл примера: *SQL-Injection/soln/mapping.php*

```
$sortorder = "bug_id";
$direction = "ASC";
```

3. Если варианты выбора пользователей соответствуют ключам, которые вы задали в массивах `$sortorders` и `$directions`, используйте соответствующие значения.

Файл примера: *SQL-Injection/soln/mapping.php*

```
if (array_key_exists($_REQUEST["order"], $sortorders)) {
   $sortorder = $sortorders[ $_REQUEST["order"] ];
}

if (array_key_exists($_REQUEST["dir"], $directions)) {
   $direction = $directions[ $_REQUEST["dir"] ];
}
```

4. Теперь безопасно использовать переменные `$sortorders` и `$directions` в SQL-запросе, так как они могут содержать только значения, которые вы описали в коде.

Файл примера: *SQL-Injection/soln/mapping.php*

```
$sql = "SELECT * FROM Bugs ORDER BY {$sortorder} {$direction}";
$stmt = $pdo->query($sql);
```

Использование этой техники имеет несколько преимуществ.

- Вам не придется объединять ввод пользователя с SQL-запросом, таким образом, уменьшается риск инъекции SQL-кода.
- Вы можете сделать любой компонент SQL-оператора динамическим, включая идентификаторы, ключевые слова SQL и даже любые выражения.
- Вы получаете простой и эффективный способ проверить правильность варианты выбора пользователей.
- Вы отсоединяете внутренние детали запросов в вашей базе данных от пользовательского интерфейса.

Варианты выбора пользователей имеют жесткую кодировку в вашем приложении, но они соответствуют названиям таблиц, столбцов, а также ключевым словам SQL. Варианты выбора пользователей по всему диапазону строк или чисел типичны для значений данных, но не для идентификаторов или синтаксиса.

Попросите друга просмотреть ваш код

Лучший способ отыскать дефекты — это найти вторую пару глаз для поиска. Попросите товарища по команде, знакомого с рисками инъекции SQL-кода, помочь вам проверить код. Не позволяйте своей гордости или самолюбию препятствовать вам делать правильные вещи — лучше смутиться сейчас пропущенной ошибке в коде, чем позже нести ответственность за трещину в безопасности, позволившую злоумышленникам взломать ваш веб-сайт.

Во время проверки на риск внедрения SQL-кода используйте следующие рекомендации.

1. Найдите SQL-операторы, сформированные с использованием переменных приложения, конкатенацией или заменой строк.
2. Проследите начало всего динамического контента, используемого в SQL-операторах. Найдите все данные, которые исходят из внешнего источника, например пользовательский ввод, файлы, среды, веб-сервисы, сторонний код или даже строки, выбранные из базы данных.
3. Предположите, что весь внешний контент несет потенциальную опасность. Используйте фильтры, контрольные устройства и отображение массивов, чтобы преобразовать ненадежный контент.
4. Объедините внешние данные в свои SQL-операторы, используя параметры запроса или устойчивые escape-функции.
5. Не забывайте просматривать хранимые процедуры и другие места, где могут находиться динамические SQL-операторы.

Проверка кода — самый точный и экономичный способ найти дефекты защиты от инъекций SQL-кода. Вы должны включать проверку кода в свой план работы и она должна являться обязательной частью работы. Вы можете также благодарить за содействие по проверке кода своих товарищей, помогая в проверке им.

ВНИМАНИЕ!

Позволяйте пользователям вводить значения, но не позволяйте им вводить код.

Тот, кто имеет значение, — не возразит, а тот, кто возразит, — не имеет значения.

Бернард Барух
(во время расположения мест для гостей
для его званого обеда)

ГЛАВА 22. ПСЕВДОКЛЮЧ АККУРАТНОСТИ

Ваш руководитель приближается к вам, держа в руках две распечатки отчетов. «Бухгалтеры сказали, что у нас несоответствия между отчетами этого квартала и прошлого. Я посмотрел и понял, что они абсолютно правы. Большинство последних активов исчезло. Что произошло?»

Вы изучаете отчеты, и шаблон несоответствия напоминает вам кое-что. «Нет, ничего не пропало. Вы просили меня очистить строки в базе данных, поэтому здесь нет никаких недостающих строк. Вы сказали, что бухгалтеры задавали Вам вопросы из-за промежутков в нумерации.

Дело в том, что я перенумеровал некоторые строки, чтобы они вписались на места отсутствующих строк, которые были там прежде. Поэтому теперь нет недостающих строк — каждый номер от 1 до, приблизительно, 12340 используется. Они по-прежнему там, но некоторые только изменили номер и поднялись выше. Вы сами сказали мне это сделать».

Ваш руководитель качает головой. «Но это не то, чего я хотел. Бухгалтеры должны проследить амортизацию по номерам активов. Номер каждой единицы оборудования должен оставаться одинаковым в каждом ежеквартальном отчете. Кроме того, все идентификационные номера активов напечатаны на ярлыках каждой единицы. Потребовались бы недели, чтобы повторно маркировать все оборудование в компании. Прошу вас, измените все идентификационные номера обратно на их исходные значения».

Вы хотите быть отзывчивым, поэтому возвращаетесь на свое рабочее место, чтобы начать работать, но в голову неожиданно приходит мысль о новой проблеме. «А как же новые активы, которые мы приобрели в этом месяце уже после того, как я объединил идентификационные номера активов? Новым активам были присвоены значения идентификационных номеров, находившихся в использовании до ренумерации. Если я верну идентификационные номера назад, что мне делать с повторяющимися номерами?»

22.1. ЦЕЛЬ: ПРИВЕДЕНИЕ ДАННЫХ В ПОРЯДОК

Существует определенный тип людей, которые расстраиваются при виде промежутков в последовательности чисел.

bug_id	status	product_name
1	OPEN	Open RoundFile
2	FIXED	ReConsider
4	OPEN	ReConsider

С одной стороны, можно понять заинтересованность руководства, поскольку неясно, что случилось со строкой `bug_id` 3. Почему запрос не возвращал ту ошибку? СУБД потеряла ее? Что было в той ошибке? Об этой ошибке сообщил один из наших важных клиентов? Собираюсь ли я нести ответственность за потерянные данные?

Цель того, кто использует антипаттерн **Псевдоключ Аккуратности**, заключается в том, чтобы решить эти беспокоящие вопросы. Этот человек несет ответственность за проблемы целостности данных, но, как правило, не обнаруживает достаточного понимания или уверенности в технологии СУБД, чтобы чувствовать себя уверенным в сгенерированных результатах отчета.

22.2. АНТИПАТТЕРН: ЗАПОЛНЕНИЕ УГЛОВ

Первая реакция большинства людей при виде пробела — естественно, заполнить этот пробел. Существует два способа сделать это.

Присвоение номеров не из имеющейся последовательности

Вместо того чтобы распределить новое значение первичного ключа, используя автоматический псевдоключевой механизм, вы можете присваивать каждой новой строке первое неиспользованное значение первичного ключа. Таким образом, при вставке данных вы естественным образом заполняете пробелы.

bug_id	status	product_name
1	OPEN	Open RoundFile
2	FIXED	ReConsider
4	OPEN	ReConsider
3	NEW	Visual TurboBuilder

Однако вам придется выполнить лишний запрос самообъединения, чтобы найти наименьшее неиспользованное значение.

Файл примера: *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;
```

Ранее в книге мы рассматривали проблему параллелизма, когда пытались распределить уникальное значение первичного ключа, выполняя такие запросы, как: `SELECT MAX(bug_id)+1 FROM Bugs`[1]. Данная ситуация имеет тот же дефект, когда два приложения пытаются найти наименьшее неиспользованное значение одновременно. Когда оба они пытаются использовать одно и то же значение в качестве первичного ключа, одному это удается, а другое выводит ошибку. Этот метод является неэффективным и приводящим к ошибкам.

Ренумерация существующих строк

Вы можете найти, что более важно сделать значения первичного ключа более последовательными, и ждать новых строк, чтобы заполнить промежутки, не решая проблему достаточно быстро. Вы можете решить использовать стратегию обновления значений ключа для существующих строк, чтобы устранить промежутки и сделать все значения непрерывными. Это обычно означает, что необходимо найти строку с наибольшим значением первичного ключа и обновить его до наименьшего неиспользованного значения. Например, вам необходимо обновить значение от 4 до 3:

Файл примера: *Neat-Freak/anti/renumber.sql*

```
UPDATE Bugs SET bug_id = 3 WHERE bug_id = 4;
```

bug_id	status	product_name
1	OPEN	Open RoundFile
2	FIXED	ReConsider
3	DUPLICATE	ReConsider

Чтобы выполнить это, необходимо найти неиспользованное значение ключа при помощи метода, подобного предыдущему, со вставкой новых строк. Вы также должны выполнить оператор `UPDATE`, чтобы присвоить новое зна-

[1] См. врезку на стр. 60.

чение первичному ключу. Каждый из этих шагов уязвим перед проблемой параллелизма. Вам придется повторить шаги множество раз, прежде чем заполнить широкий промежуток в числах.

Вы также должны распространить измененное значение во все дочерние записи, ссылающиеся на строки, которые вы перенумеровали. Это будет происходить проще всего, если вы выбрали для внешних ключей вариант `ON UPDATE CASCADE`, однако если вы этого не сделали, то вам придется отключить ограничения, обновить все дочерние записи вручную и восстановить ограничения. Это трудоемкий, подверженный ошибкам процесс, который может прервать обслуживание вашей базы данных, так что если вы решите, что лучше его избежать, то будете правы.

Даже если вы и завершите эту очистку до конца, то ненадолго. Когда псевдоключ генерирует новое значение, оно будет больше, чем предыдущее им сгенерированное (даже если строка с предыдущим значением была удалена или изменена), создав тем самым *не* наибольшее значение в таблице, как будут полагать некоторые программисты. Допустим, вы обновляете строку с наибольшим значением `4` столбца `bug_id`, приводя идентификационные номера к наименьшему значению, чтобы заполнить промежутки в нумерации. Следующая строка, которую вы вставите, используя генерацию псевдоключей по умолчанию, получит номер `5`, создавая новый промежуток — `4`.

Создание несоответствия данных

Митч Рэтклифф (Mitch Ratcliffe) сказал: «Компьютер позволяет делать больше ошибок, чем любое другое изобретение человека за всю историю... разве что за исключением пистолетов и текилы[1]».

Ситуация в начале этой главы описывает некоторые риски ренумерации значений первичного ключа. Если другая система, внешняя по отношению к вашей базе данных, зависит от идентификации строк при помощи первичных ключей, то ваши обновления лишают эту систему ссылок на данные.

Долгое время использовать первичный ключ строки — плохая идея, поскольку промежутки в нумерации могут быть результатом удаления или перенесения строк по определенной причине. Например, допустим, что пользователю с номером учетной записи `account_id` 789 закрыт доступ в систему за рассылку оскорбительных электронных писем. Согласно вашей политике необходимо удалить учетную запись нарушителя, но если вы повторно используете первичные ключи, то впоследствии вы присвоили бы номер 789 другому пользователю. Поскольку некоторые оскорбительные письма все еще не прочитаны своими получателями, вы получите новые

[1] Обзор технологий Массачусетского технологического института, апрель 1992.

жалобы о пользователе 789. Однако новый владелец этого номера, непричастный к рассылке, получит обвинение в ней.

Не присваивайте вторично значения ключа только потому, что они кажутся неиспользованными.

22.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Приведенные ниже цитаты могут означать, что кто-либо в вашей организации собирается использовать антипаттерн **Псевдоключ Аккуратности**.

- «Как я могу снова использовать автоматически сгенерированное значение идентификационных данных после того, как произвел откат введенных данных?»

Распределение псевдоключа не откатывается назад; если бы это произошло, РСУБД должна была бы распределить значения псевдоключа в рамках транзакции. Это вызвало бы либо состояние конкуренции, либо блокирование, когда большое количество клиентов вводят данные одновременно.

- «Что произошло с 4-й строкой `bug_id`?»

В данном случае беспокойство по неиспользованным числам в последовательности первичных ключей неуместно.

- «Как мне создать запрос, чтобы найти первый неиспользованный идентификационный номер?»

Причина этого поиска почти наверняка состоит в том, чтобы использовать идентификационный номер повторно.

- «Что, если я исчерпаю все номера?»

Это используется для выравнивания, чтобы перераспределить неиспользованные значения идентификационных номеров.

22.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Нет причин менять значения псевдоключа, так как значение не имеет смысла. Если значения в столбце первичного ключа несут в себе некоторый смысл, то это столбец *натурального ключа*, а не псевдоключа. Нет ничего необычного в том, чтобы менять значения натурального ключа.

22.5. РЕШЕНИЕ: ПРЕОДОЛЕНИЕ ПРОБЛЕМЫ

Значения любого первичного ключа должны быть уникальными и не являться значением `NULL`, поэтому вы можете использовать их, чтобы сослаться на отдельные строки, но существует одно правило: значения не должны быть последовательными номерами идентификации строк.

Нумерация строк

Большинство генераторов псевдоключей возвращает числа, больше похожие на номера строк, так как они *монотонно возрастают* (то есть каждое последующее значение больше предыдущего), однако это только совпадение их реализации. Генерация значений таким образом — это удобный способ гарантировать уникальность значений.

Не путайте номера строк с первичными ключами. Первичный ключ идентифицирует одну строку в одной таблице, тогда как номера строк идентифицируют строки в наборе результатов. Номера строк в наборе результатов запроса не соответствуют значениям первичного ключа в таблице чаще всего тогда, когда вы используете такие операторы запроса, как: `JOIN`, `GROUP BY` или `ORDER BY`.

Существуют серьезные основания, чтобы использовать номера строк, например, чтобы вернуть подмножество строк в результате запроса. Такую операцию, обычно, называют *пагинацией*, как на странице Интернет-поиска. Чтобы выбрать подмножество таким образом, необходимо использовать истинные номера строк, которые увеличиваются и являются непрерывно последовательными, независимо от формы запроса.

Стандарт SQL:2003 определяет *оконные функции*, включающие функцию `ROW_NUMBER()`, возвращающую непрерывно последовательные числа, определенные для набора результатов запроса. Как правило, использование нумерации строк производится для того, чтобы ограничить результаты запроса диапазоном строк:

Файл примера: *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;
```

Данные функции в настоящий момент поддерживаются моделями большинства ведущих производителей СУБД, включая Oracle, Microsoft SQL Server 2005, IBM DB2, PostgreSQL

8.4 и Apache Derby.

СУБД MySQL, SQLite, Firebird и Informix не поддерживают оконные функции стандарта SQL:2003, однако они имеют собственный синтаксис, который можно использовать в сценарии, представленном в данном разделе.

СУБД MySQL и SQLite поддерживают оператор LIMIT, а СУБД Firebird и Informix поддерживают вариант запроса с ключевыми словами `FIRST` и `SKIP`.

Использование GUID

Вы также можете генерировать случайные значения псевдоключа, если не используете числа больше одного раза. Некоторые СУБД поддерживают *глобальный уникальный идентификатор* (*GUID, Globally Unique Identifier*) для этой цели.

GUID — это псевдослучайное 128-разрядное число (обычно представленное 32-мя цифрами шестнадцатеричной системы счисления). На практике, число GUID является уникальным, поэтому его можно использовать для генерации псевдоключа.

ДЕЙСТВИТЕЛЬНО ЛИ ЦЕЛЫЕ ЧИСЛА — ДЕФИЦИТНЫЙ РЕСУРС?

Другое неверное представление, связанное с антипаттерном **Псевдоключ Аккуратности**, — это мысль о том, что генератор монотонно возрастающих значений псевдоключа в конечном счете исчерпает все возможные целые числа, в связи с чем необходимо принять меры предосторожность, чтобы не тратить значения впустую.

На первый взгляд, это кажется разумным. В математике счет целых чисел можно вести бесконечно, но в СУБД любой тип данных имеет конечное число значений. 32-разрядное целое число может предоставить максимум 2^{32} различных значений. Это правда, что каждый раз, когда вы выделяете значение для первичного ключа, вы на один шаг приближаетесь к конечному.

Однако если совершить несколько математических вычислений, можно посчитать, что если при генерации уникального значения первичного ключа вы будете добавлять по 1000 строк в секунду 24 часа в день, то пройдет лишь 136 лет, прежде чем вы исчерпаете все возможные целые значения 32-разрядного числа.

Если это вас не устраивает, вы можете использовать 64-разрядное целое число. Теперь вы сможете вводить **1 миллион** целых чисел в секунду непрерывно в течение 584 542 лет.

Очень маловероятно, что вы исчерпаете целые числа!

Приведенный ниже пример использует синтаксис СУБД Microsoft SQL Server 2005:

Файл примера: *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' );
```

Таким образом, создается строка, подобная этой:

bug_id	summary
0xff19966f868b11d0b42d00c04fc964ff	Crashes when I save

Вы получаете по крайней мере два преимущества перед традиционными генераторами значений псевдоключа, когда используете GUID:

- Можно генерировать значения псевдоключей на нескольких серверах баз данных одновременно, не используя одинаковые значения.
- Никто не будет жаловаться на промежутки в значениях — все будут слишком заняты, жалуясь на ввод 32-значных цифр шестнадцатеричной системы счисления для значений первичного ключа.

Последний пункт имеет несколько недостатков:

- Значения будут длинными и сложными для печатного ввода.
- Значения будут случайными, поэтому нельзя создать определенный шаблон или положиться на вывод о том, что большое значение указывает на недавно созданную строку.
- Хранение каждого числа GUID требует 16 байт. Это отнимает много места в памяти, и СУБД работает более медленно, чем при использовании типичного 4-разрядного целого псевдоключа.

Самая важная проблема

Теперь, когда вы знаете о проблемах, вызванных ренумерацией псевдоключей, а также о некоторых альтернативных решениях для связанных с этими проблемами целей, у вас остается главная проблема: как вы вернете утерянную последовательность нумерации для начальника, который хочет, чтобы вы навели порядок в базе данных, закрывая промежутки в последовательности значений псевдоключа? Это проблема коммуникации, не технологии. Все-таки вам, возможно, нужно было *руководить своим руководителем*, чтобы защитить целостность вашей базы данных.

- *Объясните технологию.* Честность, как правило, — лучшая политика. Будьте уважительны и, прежде чем выполнить требование, убедитесь в его необходимости. Например, скажите вашему руководителю следующее:

«Промежутки действительно выглядят непривычно, но они безопасны. Это нормально, когда строки пропускаются, переносятся назад или удаляются, время от времени. Мы выделяем новое число для каждой новой строки в базе данных, вместо того чтобы писать код для выяснения, какие старые

числа мы сможем снова безопасно использовать. Это делает наш код дешевым для разработки и более быстрым для работы и исправления ошибок».

- *Согласуйте затраты.* Изменение значений первичного ключа похоже на обычную задачу, но следует дать реалистичную оценку работы, которую займет вычисление новых значений, запись и тестирование кода, обрабатывающего повторные значения, вытекающие из этого изменения по всей базе данных, исследование воздействий на другие системы, а также обучение пользователей и администраторов управлению новыми процедурами.

Большинство руководителей имеют приоритетными задачи, основанные на стоимости, в связи с чем они должны отступить от легкомысленных требований и поиска мелкой выгоды, когда сталкиваются с реальными затратами.

- *Используйте натуральные ключи.* Если ваш руководитель или другие пользователи базы данных настаивают на интерпретации значений первичного ключа, то позвольте этим значениям иметь смысл. Не используйте псевдоключи — используйте строку или число, кодирующее определенное значение идентификации. Тогда будет легче объяснить промежутки в контексте значений этих натуральных ключей.

Вы также можете вместе использовать псевдоключ и другой столбец атрибутов, который вы используете в качестве натурального идентификатора. Скройте псевдоключ в отчетах, чтобы промежутки в значениях не заставляли пользователей беспокоиться.

Строить предположения, не зная всех обстоятельств дела, — крупнейшая ошибка.

Шерлок Холмс

ГЛАВА 23. НЕЗАМЕЧАЕМЫЕ НЕДОСТАТКИ

«Я нашел другую ошибку в вашем продукте», — раздалось в телефонной трубке.

Я получил этот звонок, когда работал инженером технической поддержки для SQL РСУБД в 1990-х. У нас был один клиент, который был известен за создание ложных отчетов нашей РСУБД. Почти все его отчеты, как оказалось, были простыми просчетами на его стороне, а не ошибками нашей системы.

«Доброе утро, мистер Девис. Конечно, мы хотели бы исправить любую проблему, которую вы обнаружили», — ответил я. — «Расскажите мне, что произошло».

«Я осуществил запрос в вашей РСУБД и не получил результат», — ответил он резко. — «Но я знаю, что данные находятся в базе данных, — я могу проверить это в сценарии тестирования».

«Была ли какая-нибудь проблема с запросом?» — спросил я. — «API-интерфейс выводил ошибку?»

На это Девис ответил: «Почему я должен смотреть на возвращаемое значение API-функции? Функция должна только выполнить мой SQL-запрос. Если она возвращает ошибку, это указывает на то, что Ваш продукт имеет дефект. Если бы у вашего продукта не было дефектов, не было бы никаких ошибок. Я не обязан работать с вашими ошибками».

Я был ошеломлен, но я должен был позволить фактам говорить самим за себя. «Ладно, давайте попробуем провести тестирование. Скопируйте и вставьте *точный* SQL-запрос из своего кода в инструмент создания запросов и запустите его. Что он выводит?» — Я ждал его ответа.

«Синтаксическая ошибка в предложении `SELECT`». После паузы он добавил, — «Можете закрыть этот вопрос», — и резко бросил трубку.

Мистер Девис был единственным разработчиком в компании авиадиспетчерской службы, пишущим программное обеспечение, регистрирующее данные о международных рейсах. Мы каждую неделю получали известия от него.

23.1. ЦЕЛЬ: НАПИСАНИЕ МЕНЬШЕГО КОЛИЧЕСТВА КОДА

Все хотят писать *элегантный код*. То есть все хотят проделывать отличную работу с использованием малого количества кода. Чем выше класс работы и чем меньше кода на это ушло, тем выше коэффициент элегантности. Если не удается сделать более высококлассную работу, то хочется повысить коэффициент элегантности за счет уменьшения количества кода для проделанной работы.

Это поверхностная причина, однако существует более рациональная причина для написания краткого кода:

- мы закончим работу над кодом приложения в более короткие сроки;
- мы будем иметь меньше кода, который нужно тестировать, документировать или предоставлять для оценки экспертов;
- мы получим меньше ошибок, если будем иметь меньше строк кода.

Отсюда и инстинктивное желание программистов устранить любой код, который возможно, особенно если он не делает работу более высококлассной.

23.2. АНТИПАТТЕРН: ТРУДНОВЫПОЛНИМОЕ ДЕЛО

Разработчики обычно практикуют антипаттерн **Незамечаемые недостатки** в двух формах: первая — игнорирование возвращаемых значений API-интерфейса СУБД; и вторая — чтение фрагментов кода SQL, рассеянных по коду приложения. В обоих случаях разработчикам не удается использовать информацию, которая им легкодоступна.

Диагнозы без диагностики

Файл примера: *See-No-Evil/anti/no-check.php*

```
   <?php
1. $pdo = new PDO("mysql:dbname=test;host=db.example.com",
   "dbuser", "dbpassword");
   $sql = "SELECT bug_id, summary, date_reported FROM Bugs
   WHERE assigned_to = ? AND status = ?";
2. $stmt = $dbh->prepare($sql);
3. $stmt->execute(array(1, "OPEN"));
4. $bug = $stmt->fetch();
```

Этот код краток, но в нем есть несколько участков, где значения состояния, возвращенные из функций, могут указывать на проблему, но вы никогда

не узнаете об этом, если будете игнорировать возвращаемые значения.

Наверное, наиболее распространенная ошибка API-интерфейса СУБД происходит, когда вы пытаетесь создать соединение с СУБД, как показано в пункте 1 примера. Вы могли случайно ввести название базы данных или узла сервера с опечаткой, или неправильно понять имя учетной записи или пароль, или сервер базы данных мог быть недоступен. Ошибка при обработке PDO-соединения выдает исключение, прекращающее выполнение сценария из примера, показанного выше.

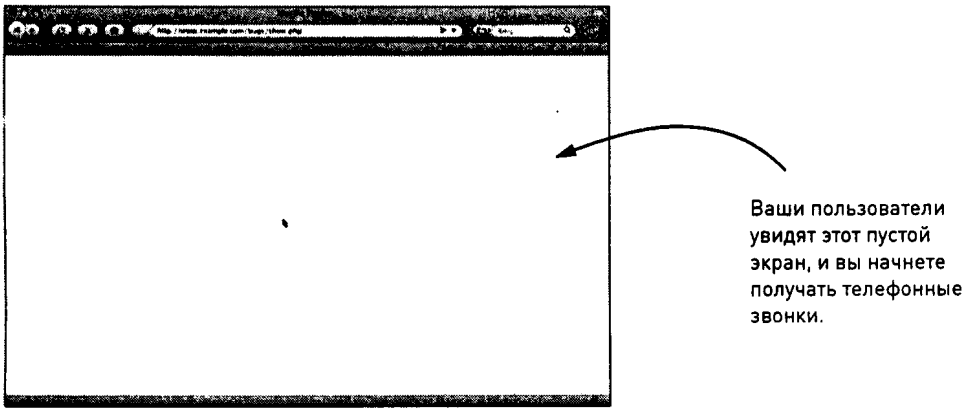


Рис. 23.1. Фатальная ошибка в PHP приводит к появлению пустого экрана.

Вызов функции `prepare()` в пункте 2 мог вернуть `false`, если вы имеете простую синтаксическую ошибку, созданную опечаткой, дисбалансом круглых скобок или орфографической ошибкой в названии столбца. Если это произошло, попытка вызвать функцию `prepare()` как метод для переменной `$stmt` в пункте 3 выдаст фатальную ошибку, поскольку значение `false` не является объектом.

```
PHP Fatal error: Call to a member function execute() on a non-
object
```

Вызов функции `execute()` может также не сработать, например, потому что оператор нарушает ограничение или превышает права доступа. Метод также возвращает `false` с ошибкой.

Вызов функции `fetch()` в пункте 4 мог вернуть `false`, если бы происходила ошибка, схожая с ошибкой при сбое РСУБД.

Программисты с отношением, как у мистера Девиса, являются не такими уж редкими. Им может казаться, что проверка возвращаемых значений и

исключений ничего не прибавляет их коду, потому что не предполагают, что что-то может произойти. Кроме того, дополнительный код является повторяющимся, что делает его уродливым и трудным для чтения. Это определенно не прибавляет высококлассности.

Однако пользователи не видят код, они видят только вывод. Когда фатальная ошибка появляется необработанной, пользователь может увидеть лишь пустой белый экран, как показано на рис. 23.1, или непонятное сообщение об исключительной ситуации. Когда такое происходит, то опрятность и краткость кода приложения являются плохим утешением.

Чтение между строк

Другая распространенная плохая привычка, свойственная использующим антипаттерн **Незамечаемые недостатки**, заключается в отладке кода путем пристального взгляда на прикладной код, в который встроен SQL-запрос в виде строки. Это трудно, поскольку тяжело представить результирующую SQL-строку после встраивания ее в логику приложения, конкатенацию строк или дополнительный контент из переменных приложения. Попытка отладить код таким образом похожа на попытку собрать паззлы, не смотря на картинку на коробке.

Для простого примера, взгляните на тип вопроса, который часто задают разработчики. Указанный ниже код встраивает запрос в зависимости от конкатенации оператора `WHERE`, если сценарий должен искать определенную ошибку вместо коллекции ошибок.

Файл примера: *See-No-Evil/anti/white-space.php*

```
<?php
$sql = "SELECT * FROM Bugs";
if ($bug_id) {
   $sql .= "WHERE bug_id = " . intval($bug_id);
}
$stmt = $pdo->prepare($sql);
```

Почему запрос в этом примере выдает ошибку? Ответ будет абсолютно ясен, если посмотреть на полную строку `$sql`, являющуюся результатом конкатенации:

Файл примера: *See-No-Evil/anti/white-space.sql*

```
SELECT * FROM BugsWHERE bug_id = 1234
```

Между словами `Bugs` и `WHERE` нет пробела, что дает запросу недопустимый синтаксис, буквосочетание `BugsWHERE` распознается как название таблицы,

сопровождаемое SQL-выражением в неверном контексте. Код соединил строки, не оставив между ними пробела.

Разработчики тратят впустую невероятное количество времени и энергии, пытаясь решить проблемы, похожие на эту, просто смотря на код, в который встроен код SQL, вместо того чтобы смотреть непосредственно на код SQL.

23.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Даже если бы вы подумали о том, что отсутствие кода само по себе трудно определить, многие современные интегрированные среды разработки (IDE, Integrated Development Environment) подсвечивают экземпляры класса в вашем коде, в которых вы игнорируете единственное возвращаемое функцией значение или в которых ваш код вызывает функцию, но не обрабатывает проверяемое исключение[1]. Вы также можете определить, что столкнулись со случаем использования антипаттерна **Незамеченные недостатки**, если услышите похожие фразы:

- «Моя программа дает сбой после того, как я произвожу запрос в базе данных».

Зачастую сбои происходят из-за того, что запрос терпит неудачу или вы пытаетесь использовать результат недопустимым образом, к примеру, вызывая метод без объекта или производя разыменование указателя `NULL`.

- «Не могли бы Вы помочь мне найти ошибку в SQL? Вот мой код...»

Начните с просмотра SQL-кода, а не кода, в который он встроен.

- «Я не стал беспокоиться о загромождении моего кода обработкой ошибок».

Некоторые программисты оценили, что до 50% строк кода в надежном приложении разработаны для обработки ситуаций, вызванных ошибками. Может показаться, что это много, если не думать обо всех шагах, которые могут быть включены при обработке ошибок: обнаружение, классификация, вывод сообщения и корректировка. Любое программное обеспечение должны быть готово выполнять все эти шаги.

23.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Вы можете опустить проверку ошибок, когда действительно не можете сделать ничего в ответ на ошибку. Например, функция `close()` выводит статус подключения к базе данных, но если ваше приложение закрывается или

[1] Проверяемое исключение — это такое исключение, которое присваивает себе сигнатура функции, благодаря чему вы знаете, что данная функция способна генерировать такой тип исключения.

прерывает соединение каким-либо образом, то, скорее всего, ресурсы для данного подключения будут очищены в любом случае.

Исключения в объектноориентированных языках программирования позволяют запускать исключения, не отвечая за их обработку. Ваш код полагает, что, что бы он ни вызывал, вашим кодом будет являться код, отвечающий за обработку исключений. Поэтому ваш код может позволить исключению выполнять резервное копирование стека вызывающей программы.

23.5. РЕШЕНИЕ: ИЗЯЩНОЕ ВОССТАНОВЛЕНИЕ ПОСЛЕ ВОЗНИКНОВЕНИЯ ОШИБОК

Любой, кто любит танцевать, знает, что оплошности неизбежны. Секрет к сохранению изящества состоит в том, чтобы знать, как оправиться от ошибки. Дайте себе возможность увидеть причину ошибки. Затем вы сможете реагировать быстро и плавно, вернувшись в ритм прежде, чем кто-либо заметит вашу оплошность.

Поддерживайте ритм

Проверка возвращенного состояния и исключений API-вызовов базы данных — лучший способ гарантировать, что вы не пропустили шаг. В следующем примере показан код, который производит проверку состояния после каждого вызова, в связи с которым могла произойти ошибка:

Файл примера: *See-No-Evil/soln/check.php*

```
<?php
try {
$pdo = new PDO("mysql:dbname=test;host=localhost",
"dbuser", "dbpassword");

① } catch (PDOException $e) {
report_error($e->getMessage());
return;
}
$sql = "SELECT bug_id, summary, date_reported FROM Bugs
WHERE assigned_to = ? AND status = ?";

② if (($stmt = $pdo->prepare($sql)) === false) {
$error = $pdo->errorInfo();
report_error($error[2]);
return;
}
```

```
③ if ($stmt->execute(array(1, "OPEN")) === false) {
  $error = $stmt->errorInfo();
  report_error($error[2]);
  return;
  }

④ if (($bug = $stmt->fetch()) === false) {
  $error = $stmt->errorInfo();
  report_error($error[2]);
  return;
  }
```

Код в пункте 1 получает исключение, если соединение с базой данных прерывается. Другие функции возвращают ложь, когда появляется данная проблема. После проверки на наличие ошибок в пунктах 2, 3 и 4 можно получить подробную информацию от объекта подключения к базе данных или объекта оператора.

Отслеживайте свои шаги повторно

Также важно использовать фактический SQL-запрос для отладки проблемы, вместо кода, создающего SQL-запрос. Многие простые ошибки, вроде орфографических ошибок или дисбаланса круглых скобок, становятся видны немедленно, даже несмотря на то, что они не являются отчетливыми и приводят в замешательство.

- Встраивайте SQL-запрос в переменную, а не в специальные аргументы метода API-интерфейса, чтобы подготовить запрос к выполнению. Это даст вам возможность проверить переменную прежде, чем приступить к ее использованию.
- Выберите место для SQL-вывода, не являющегося частью вывода вашего приложения, например, для журнала регистрации событий, консоли отладчика IDE-среды или расширения браузера для показа вывода диагностики[1].
- Не печатайте SQL-запрос в пределах комментариев HTML-вывода или вывода веб-приложения. Любой пользователь сможет взглянуть на исходный код вашей страницы. Чтение SQL-запросов дает взломщикам много полезных знаний о структуре вашей базы данных.

[1] Расширение Firebug (**getfirebug.com/**) — хороший пример.

Использование каркаса объектно-реляционного отображения, который встраивает и выполняет SQL-запросы, очевидно, может усложнить отладку. Если у вас нет доступа к содержимому SQL-запроса, как вы можете изучить его, чтобы отладить? Некоторые ORM-каркасы решают эту проблему, отправляя сгенерированный SQL-код в журнал регистрации.

Наконец, большинство марок СУБД предоставляют свои собственные механизмы регистрации событий на серверах баз данных вместо регистрации событий в прикладном коде клиента. Если вы не можете включить регистрацию событий SQL в приложении, вы по-прежнему можете контролировать запросы, поскольку сервер базы данных выполняет их.

ВНИМАНИЕ!

Диагностика кода является и без того достаточно трудной. Не препятствуйте себе, выполняя ее вслепую.

У людей аллергия на перемены. Они любят говорить: «Мы всегда делали именно так». Я пытаюсь бороться с этим. Именно поэтому у меня на стене висят часы, идущие против часовой стрелки.

Контр-адмирал
Грейс Мюррей Хоппер

ГЛАВА 24. ДИПЛОМАТИЧЕСКАЯ НЕПРИКОСНОВЕННОСТЬ

Одна из моих первых должностей дала мне урок о том, как важно использовать лучшие методы разработки программного обеспечения, после того как трагический инцидент заставил меня нести ответственность за важное приложение базы данных.

Я был на собеседовании по поводу контрактной работы в компании Hewlett-Packard, заключающейся в разработке и обслуживании приложения на ОС UNIX, написанном на языке C для сервера базы данных HP ALLBASE/SQL. Руководитель и сотрудники, проводившие мое собеседование, с печалью сообщили мне, что их программист, работавший над этим приложением, погиб в автокатастрофе. В их отделе больше никто не умел работать с ОС UNIX и всем остальным, что связано с этим приложением.

После того как я начал работать, я обнаружил, что разработчик никогда не писал документацию или тесты для приложения, он никогда не пользовался ни системой управления исходным кодом, ни даже комментариями к коду. Весь его код постоянно находился в единственном каталоге, в том числе код, который являлся частью оперативной системы, код, находящийся в разработке, и код, который больше не использовался.

Этот проект имел важный *технический долг* — последствия использования сокращений вместо использования наилучших методов[1]. Технический долг несет в себе риски и дополнительную работу в проекте, пока вы не расплатитесь по нему рефакторингом, тестированием и документированием.

Я работал в течение шести месяцев на организацию и документацию кода, пока приложение не стало действительно довольно скромным, поскольку я обязан был тратить бóльшую часть своего времени на обслуживание пользователей и продолжение разработки.

Само собой, у меня не было возможности попросить моего предшественника о помощи, чтобы ускорить работу. Данный опыт действительно демонстрировал, к чему может привести выход технического долга из-под контроля.

---

[1] Уорд Каннингем (Ward Cunningham) придумал эту метафору на конференции OOPSLA в 1992 году (**c2.com/doc/oopsla92.html**).

24.1. ЦЕЛЬ: ИСПОЛЬЗОВАНИЕ ПЕРЕДОВЫХ МЕТОДОВ РАБОТЫ

Профессиональные программисты стремятся использовать хорошие привычки разработки программного обеспечения в своих проектах такие, как следующее.

- Хранение исходного кода приложения под контролем изменений при помощи таких инструментальных средств, как Subversion или Git.
- Разработка и выполнение автоматизированных тестов модуля или проверок работоспособности приложений.
- Написание документации, обозначений и комментариев к коду, чтобы описывать требования и стратегии реализации приложения.

Время, которое вы потратите на разработку программного обеспечения, используя лучшие методы, будет являться чистым выигрышем, поскольку таким образом будет сокращена бóльшая часть бесполезной или повторяющейся работы. Большинство опытных разработчиков знает, что принесение в жертву этих методов ради рационализации — это верный путь к неудаче.

24.2. АНТИПАТТЕРН: ИСПОЛЬЗОВАНИЕ SQL КАК «ВСПОМОГАТЕЛЬНОГО» ЯЗЫКА

Даже среди разработчиков, использующих лучшие методы при разработке прикладного кода, существует тенденция думать, что к коду базы данных это не относится. Я назвал этот антипаттерн **Дипломатическая Неприкосновенность**, поскольку он предполагает, что правила разработки приложений не применяются к разработке базы данных.

Откуда у разработчиков появилась эта мысль? Ниже представлены некоторые возможные причины.

- Роли специалиста по программному обеспечению и администратора базы данных являются раздельными в некоторых компаниях. Администратор базы данных (DBA, Database administrator) обычно работает с несколькими группами программистов, поэтому появляется впечатление, что он не постоянный член каждой из этих команд. С ним обращаются как с посетителем, и он не подчинен тем же обязанностям, что и специалисты по программному обеспечению.
- Программирование на языке SQL, используемого для РСУБД, отличается от обычного программирования. Даже способ вызова SQL-операторов этот специализированный, встраиваемый в код приложения, язык предлагает статус, похожий на статус гостя.
- Расширенные инструментальные средства интегрированной среды разработки популярны для прикладных кодовых языков, они помогают

производить редактирование, тестирование, а система управления исходным кодом работает быстро и безболезненно. Однако инструментальные средства для разработки СУБД не столь продвинуты или, по крайней мере, не так широко используется. Разработчики могут легко писать код для приложений, используя лучшие методы, но, применяя эти методы к SQL, они чувствуют себя неловко. Они склоняются к тому, чтобы заниматься другими вещами.

- В информационных технологиях знания и операции с базой данных обычно фокусируются на одном человеке — администраторе базы данных. Поскольку администратор базы данных — единственный, у кого есть доступ к серверу базы данных, он служит живой базой знаний системы управления исходным кодом.

База данных — основа приложения и показатель качества. Вы знаете, как разработать прикладной код высокого качества, но вы можете встроить свое приложение поверх базы данных, которая не смогла удовлетворить потребности проекта или является неудачной по причинам, которые никто не может понять. Риск заключается в том, что вы можете разработать приложение только, чтобы понять, что вам придется удалить его.

24.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Вы могли подумать, что тяжело привести доказательство невыполнения чего-либо, но это не всегда так. Ниже приведены некоторые устные признаки срезания углов.

- «Мы принимаем новый технический процесс — вот его облегченная версия».

Облегченная в данном контексте означает, что команда пропустит выполнение некоторых задач, которого требует процесс разработки. Некоторые действия действительно может быть разумным пропустить, но некоторые также будет глупо пропускать, поскольку не будет соблюдаться принцип выполнения лучших методов.

- «Мы не нуждаемся в том, чтобы штат администраторов базы данных обучался нашей новой системе управления исходным кодом, так как они все равно не используют ее.»

Недопущение некоторых членов технических групп до обучения дает гарантию того, что они не будут пользоваться инструментальными средствами, обращению с которыми обучают на этих курсах.

- «Как мне проследить использование таблиц и столбцов в базе данных? В ней есть некоторые элементы, назначения которых мы не знаем, и мы хотели бы удалить их, если они устарели».

Вы не используете проектную документацию схемы базы данных. Документ может быть устаревшим, недоступным или, возможно, никогда не существовавшим. Даже если вы не знаете назначение некоторых таблиц или столбцов, они могут быть важными для кого-то, и вы не можете удалить их.

- «Существует ли инструментальное средство, чтобы сравнить две схемы баз данных, сообщить о различиях и создать сценарий, изменяющий одну таблицу, чтобы она соответствовала другой?»

Если вы не следуете процессу размещения изменений в схеме базы данных, они могут выйти из синхронизации, после чего будет трудно вернуть их порядок.

24.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

Я пишу документацию и тесты и использую систему управления исходным кодом, а также имею другие хорошие привычки для любого кода, который я хочу использовать больше одного раза. Но я также пишу код, который является действительно специализированным, например, одноразовый тест функции API-интерфейса, чтобы напомнить себе, как пользоваться им или SQL-запросом, я пишу ответ на вопрос пользователя.

Хорошим ли руководством является удаление временного кода немедленно после использования. Если вы не можете заставить себя сделать это, вероятно, его стоит сохранить. В этом нет ничего страшного, но это означает, что его лучше хранить в системе управления исходным кодом и написать, по крайней мере, несколько коротких заметок о том, для чего этот код и как его использовать.

24.5. РЕШЕНИЕ: ФОРМИРОВАНИЕ СОБИРАТЕЛЬНОЙ КУЛЬТУРЫ КАЧЕСТВА

Качество просто тестируется большинством разработчиков программного обеспечения, но это только часть, называемая *контроль качества*, одной большой истории. Полный жизненный цикл разработки программного обеспечения включает в себя *обеспечение качества*, которое состоит из трех пунктов.

1. В письменной форме ясно определите требования к проекту.
2. Проектируйте и разрабатывайте решение согласно вашим требованиям.
3. Проверьте правильность и протестируйте ваше решение на соответствие требованиям.

Вы должны сделать все это, чтобы правильно выполнить обеспечение качества, хотя некоторые методологии программного обеспечения не приписывают выполнять эти пункты строго в данном порядке.

Вы можете выполнить контроль качества в разработке базы данных следующими наилучшими методами: *документация*, *управление исходным кодом* и *тестирование*.

Пункт A: документация

Не существует такой вещи, как самодокументация кода. Хоть это и правда, что квалифицированные программисты могут расшифровывать бóльшую часть кода при помощи комбинаций осторожного анализа и экспериментирования, но это очень кропотливое занятие[1]. Кроме того, код не может сказать вам о недостающих возможностях или нерешенных проблемах.

Вы должны документировать требования и применение базы данных во время написания прикладного кода. Вне зависимости от того являетесь ли вы исходным дизайнером базы данных или приняли ее, разработанную кем-то другим, используйте следующий контрольный список, чтобы документировать ее.

Диаграмма сущностей и связей. Единственная самая важная часть документации базы данных — это ER-диаграмма (Entity-relationship diagram, диаграмма сущностей и связей), отображающая таблицы и их связи. В некоторых главах этой книги используется упрощенная форма ER-диаграмм. Более сложные ER-диаграммы содержат записи о столбцах, ключах, индексах и других объектах базы данных.

Некоторые пакеты программного обеспечения, связанные с диаграммами, включают элементы системы обозначения ER-диаграмм. Некоторые инструменты могут даже изучить принципы работы сценария SQL или оперативной базы данных и создать ER-диаграмму.

Существует одно предупреждение, которое заключается в том, что базы данных могут быть комплексными и содержать так много таблиц, что было бы непрактично использовать для них единственную диаграмму. В таком случае вы должны разбить ее на несколько диаграмм. Обычно можно выбрать натуральные подгруппы таблиц, чтобы каждая диаграмма была достаточно легко читаема, чтобы быть полезной и не затруднять пользователя.

Таблицы, столбцы, и представления. Вы также нуждаетесь в написании документации для своей базы данных, поскольку ER-диаграмма имеет неподходящий формат для описания целей и способов использования каждой таблицы, столбца и других объектов.

---

[1] Если бы код был читаем, почему бы тогда его назвали *кодом*?

Таблица должна иметь описание о том, какой вид ввода данная таблица моделирует. Например, назначение таблиц `Bugs`, `Products` и `Accounts` вполне ясно, но как насчет таких таблиц, как `BugStatus`, или перекрестной таблицы `BugsProducts`, или зависимой таблицы `Comments`? Кроме того, сколько строк, по вашим ожиданиям, будет иметь каждая таблица? Какие индексы будут существовать в них?

Каждый столбец имеет название и тип данных, но это не говорит пользователю, что означают значения столбца. Какие значения понятны в данном столбце (полный спектр типов данных встречается редко)? В столбцах, хранящих количественные значения, каковы единицы измерения этих значений? Позволяет ли столбец ячейкам иметь значение `NULL`? Имеет ли он уникальное ограничение, и если да — почему?

Представления хранят часто используемые запросы одной или нескольких таблиц. Что явилось причиной для создания данного представления? Какое приложение или какие пользователи будут использовать данное представление? Было ли представление предназначено, чтобы абстрагировать комплексные отношения таблиц? Оно существует, как способ предоставления возможности непривилегированным пользователям осуществлять запросы подмножеств строк или столбцов в привилегированной таблице? Обновляемо ли представление?

Связи. Обеспечение целостности данных, содержащих ссылки, ограничивает взаимосвязи между таблицами, но это еще не значит, что все, чему вы даете обозначение, ограничивает модель. Например, столбец `Bugs.reported_by` не может содержать значение `NULL`, однако столбец `Bugs.assigned_to` — может. Значит, ошибка может быть исправлена прежде, чем о ней будет сообщено? Если нет, какие следует устанавливать бизнес-правила для сообщенных ошибок?

В некоторых случаях, могут существовать скрытые связи, но для них не может быть ограничений. Без документации трудно будет узнать, где существуют эти связи.

Триггеры. Проверка корректности данных, преобразование данных и регистрация изменений в базе данных — таковы примеры задач триггера. Какие бизнес-правила вы реализуете в триггерах?

Хранимые процедуры. Документируйте хранимые процедуры как в API-интерфейсе. Какую проблему решает данная процедура? Изменяет ли данная процедура какие-либо данные? Каковы типы данных и значения параметров ввода и вывода для данной процедуры? Назначаете ли вы данную процедуру для замены определенного типа запроса, чтобы устранить узкое место производительности? Или вы используете про-

цедуру для предоставления непривилегированного пользовательского доступа к привилегированным таблицам?

Безопасность SQL. Как вы определяете, какие пользователи базы данных будут пользоваться приложениями? Какие привилегии доступа будет иметь каждый из этих пользователей? Какие SQL-роли вы предоставляете, и какие пользователи смогут иметь их? Будут ли объявлены какие-либо пользователи для выполнения определенных задач, например, для выполнения резервного копирования или отсылки отчетов? Какие условия безопасности на системном уровне вы используете, например, когда клиент должен обращаться к РСУБД через протокол SSL (Secure Sockets Layer, протокол безопасных соединений)? Какие меры вы предпринимаете для обнаружения и блокирования попыток противоправной аутентификации, как при полном переборе паролей? Выполняете ли вы тщательный обзор кода с целью поиска уязвимых для внедрения SQL-кода мест?

Инфраструктура базы данных. Эта информация в основном используется ИТ-персоналом и администраторами базы данных, однако разработчики также обязаны знать кое-что из этой информации. С какой моделью и версией РСУБД вы работаете? Каково имя главного узла сервера базы данных? Используете ли вы несколько серверов баз данных, пользуетесь ли вы репликацией, кластерами, модулями доступа и т.п.? Какова организация вашей сети и номер порта, используемый сервером базы данных? Какие варианты подключения необходимы клиентским приложениям? Каковы пароли пользователей базы данных? Какова ваша политика резервного копирования базы данных?

Объектно-реляционное отображение. Ваш проект может реализовать некоторую обрабатывающую логическую схему для базы данных в прикладном коде, как это реализуется в уровне классов кода, базирующегося на ORM. Какие бизнес-правила устанавливаются в таком случае? Проверка корректности данных, регистрация событий, кэширование или профилирование?

Разработчики не любят обслуживать техническую документацию. Ее тяжело написать, трудно поддерживать на современном уровне, и это удручает, когда очень мало людей читает то, что вы с таким трудом написали. Но закаленные, экстремальные программисты знают, что они обязаны документировать базу данных, даже если они не пишут документацию ни к какому другому программному продукту[1].

---

[1] Например, Джефф Этвуд (Jeff Atwood) и Джоэл Спольски (Joel Spolsky) не видят особой необходимости в документировании кода, *за исключением* кода базы данных. Подкаст № 80 на сайте StackOverflow: **blog.stackoverflow.com/2010/01/podcast-80/**.

След доказательства: управление исходным кодом

Если бы сервер вашей базы данных полностью утратил работоспособность, как бы вы воссоздали базу данных? Каков лучший способ проследить комплексное обновление для проектирования вашей базы данных? Как бы вы отменили примененное обновление?

Известно, что нужно использовать систему управления исходным кодом, чтобы контролировать прикладной код, решая подобные проблемы разработки программного обеспечения. Проект, контролируемый системой управления исходным кодом, должен включить все, что вам необходимо для перестройки и перемещения проекта, если текущее перемещение не удается. Система управления исходным кодом служит журналом изменений, а также производит инкрементное резервное копирование, благодаря чему можно полностью отменить любое из произведенных изменений.

Вы можете использовать систему управления исходным кодом в коде своей базы данных и получать те же преимущества при разработке.

Вы должны зарегистрировать в системе управления исходным кодом файлы, связанные с разработкой вашей базы данных. Они должны включать:

Сценарии определения данных. Все модели СУБД предоставляют способы выполнения сценариев SQL, содержащих оператор `CREATE TABLE` и другие операторы, определяющие объекты базы данных.

Триггеры и процедуры. Многие проекты дополняют прикладной код подпрограммами, сохраненными в базе данных. Ваше приложение, скорее всего, не будет работать без этих подпрограмм, поэтому они берутся в расчет, как часть кода вашего проекта.

ИНСТРУМЕНТЫ РАЗВИТИЯ СХЕМЫ

Ваш код находится под системой управления исходным кодом, но база данных — нет. При помощи программного каркаса Ruby on Rails была популяризирована методика, называемая миграцией, помогающая управлять модернизацией экземпляров класса базы данных системой управления исходным кодом. Давайте увидим это на коротком примере модернизации:

Напишите сценарий модернизации базы данных за один шаг, базируясь на абстрактных классах инструмента Rails, чтобы произвести изменения. Также напишите понижающую функцию, аннулирующую изменения функции модернизации:

```
class AddHoursToBugs < ActiveRecord::Migration
  def self.up
        add_column :bugs, :hours, :decimal
  end

  def self.down
        remove_column :bugs, :hours
  end
end
```

Инструмент Rails, производящий миграцию, автоматически создаст таблицу записей исправления или исправлений, применяемых к текущему экземпляру класса вашей базы данных. В версии Rails 2.1 были введены изменения, благодаря которым эта система стала более гибкой, последующие версии Rails также могут изменить способ работы миграции.
Создайте новый сценарий миграции для каждого преобразования схемы в базе данных. Вы создаете набор сценариев этих миграций, каждый из которых может улучшить или ухудшить схему базы данных за один шаг. Если вам необходимо изменить базу данных, к примеру, до версии 5, укажите этот аргумент в инструменте миграции:

```
$ rake db:migrate VERSION=5
```

Чтобы узнать о миграциях больше, читайте книгу Agile Web Development with Rails, Third Edition [13] или посетите сайт guides.rubyonrails.org/migrations.html.

Большинство других каркасов веб-разработки, включая Doctrine для PHP, Django для Python, а также Microsoft ASP.NET, поддерживают функции, схожие с миграциями Rails, либо включенные в каркас, либо доступные как проект сообщества. Миграции автоматизируют бóльшую часть утомительной работы по синхронизации экземпляра класса базы данных со структурой, ожидаемой в данном проекте, под управлением исходным кодом. Но они не совершенны. Перемещения обрабатывают лишь малую часть простых типов изменений схемы, и они в основном реализуют систему исправлений поверх вашей обычной системы управления исходным кодом.

Данные начальной загрузки. Поисковые таблицы могут содержать некоторый набор данных, которые представляют начальное состояние вашей базы данных, прежде чем пользователи введут новые данные. Вы должны сохранить данные начальной загрузки, это поможет при необходимости воссоздать базу данных из своего проектного источника. Также такие данные называют *данными начального числа.*

ER-диаграммы и документация. Эти файлы — не код, но они тесно связаны с кодом, они описывают требования базы данных, применение и степень интеграции с приложением. Поскольку проектное развитие заканчивается изменениями и базы данных и приложения, вы должны хранить эти файлы на современном уровне. Удостоверьтесь, что документация описывает текущие дизайны.

Сценарии администратора базы данных. Многие проекты имеют список работы по обработке данных, выполняющейся вне приложения. Эти задачи включают: импорт/экспорт, синхронизацию, создание отчетов, создание резервных копий, проверку корректности, тестирование и т. д. Они могут быть написаны в виде сценариев SQL, а не в виде части обычного языка прикладного программирования.

Удостоверьтесь, что файлы кода базы данных связаны с кодом приложения, использующего базу данных. Одно из преимуществ использования системы

управления исходным кодом заключается в том, что когда вы проверяете свой проект, системы управления исходным кодом присваивает проверке номер, дату или промежуточный отчет, поэтому файлы должны использоваться совместно. Используйте один репозиторий системы управления исходным кодом и для прикладного кода, и для кода базы данных.

Бремя доказательства: тестирование

Финальная часть проверки качества — это проверка корректности, то есть действительно ли ваше приложение делает то, что должно. Многие профессиональные разработчики знакомы с методиками написания автоматизированных тестов проверки корректности поведения прикладного кода. Один из важнейших принципов тестирования — *изолированность*, то есть тестируется только одна часть системы за раз, поэтому, если дефект существует, вы сможете сузить диапазон поиска, насколько это возможно.

Можно применить практику изолированного тестирования ко всей базе данных для проверки корректности ее структуры и поведения, независимо от прикладного кода.

Следующий пример показывает сценарий тестирования элементов с использованием каркаса тестирования PHPUnit[1]:

Файл примера: *Diplomatic_immunity/DatabaseTest.php*

```
<?php
require_once "PHPUnit/Framework/TestCase.php";

class DatabaseTest extends PHPUnit_Framework_TestCase
{
   protected $pdo;

   public function setUp()
   {
       $this->pdo = new PDO("mysql:dbname=bugs", "testuser",
"xxxxxx");
   }

   public function testTableFooExists()
   {
       $stmt = $this->pdo->query("SELECT COUNT(*) FROM Bugs");
```

[1] См. **www.phpunit.de/**. По всеобщему признанию, тестирование функциональности базы данных не является строго *поблочным тестированием*, но вы все же можете использовать данное инструментальное средство для организации и автоматизации тестирования.

```
        $err = $this->pdo->errorInfo();
        $this->assertType("object", $stmt, $err[2]);
        $this->assertEquals("PDOStatement", get_class($stmt));
    }

    public function testTableFooColumnBugIdExists()
    {
        $stmt = $this->pdo->query("SELECT COUNT(bug_id) FROM
Bugs");
        $err = $this->pdo->errorInfo();
        $this->assertType("object", $stmt, $err[2]);
        $this->assertEquals("PDOStatement", get_class($stmt));
    }

    static public function main()
    {
        $suite = new PHPUnit_Framework_TestSuite(__CLASS__);
        $result = PHPUnit_TextUI_TestRunner::run($suite);
    }

}

DatabaseTest::main();
```

Вы можете использовать следующий контрольный список для тестов для проверки корректности вашей базы данных.

Таблицы, столбцы, представления. Вы должны тестировать те таблицы и представления, которые по вашим ожиданиям существуют в базе данных и эти ожидания правдивы. Каждый раз вы увеличиваете базу данных новой таблицей, представлением или столбцом, добавляете новый тест, который доказывает существование объекта. Вы также можете использовать *тесты на опровержение*, чтобы подтвердить, что таблицы или столбцы, которые вы удалили в текущей модификации вашего проекта, фактически больше не существуют.

Ограничения. Это другой способ использования тестирования на опровержение. Попытайтесь выполнить операторы `INSERT`, `UPDATE` или `DELETE`, которые привести к ошибке из-за ограничения. Например, попытайтесь нарушить такие ограничения, как `NOT NULL`, уникальные ограничения или внешние ключи. Если оператор не возвращает ошибку, то ваше ограничение не работает. Вы можете определить многие ошибки на раннем этапе, идентифицируя подобные сбои.

Триггеры. Триггеры также могут иметь ограничения. Они могут производить эффекты каскадирования, преобразовывать значения, регистрировать изменения и т. п. Вы должны тестировать эти сценарии, выполняя оператор, создающий триггер, и запрашивая подтверждение того, что триггер выполнил назначенное вами действие.

Хранимые процедуры. Тестирование процедур в базе данных больше всего напоминает обычное поблочное тестирование в прикладном коде. Хранимая процедура имеет входные параметры, которые могут создавать ошибки, если вы попытаетесь передать значения вне диапазона корректного ввода. Логическая схема внутри тела процедуры допускает существование нескольких выполняемых ветвей. Процедура может вернуть одно значение или набор результатов запроса, в зависимости от входных данных и состояние данных в базе данных. Кроме того, процедура может иметь *побочные эффекты* в виде обновления базы данных. Вы можете протестировать все эти функции процедур.

Данные начальной загрузки. Даже пустая база данных обычно нуждается в некоторых исходных данных, например, в поисковых таблицах. Вы можете выполнить запрос, чтобы проверить корректность существующих исходных данных.

Запросы: В прикладной код добавлены SQL-запросы. Вы можете выполнять запросы в среде тестирования, чтобы проверять корректность синтаксиса и результатов. Проверьте, включает ли набор результатов имена столбцов и типы данных, которые вы ожидаете получить, так же как при тестировании таблиц и представлений.

Классы ORM. Как триггеры, классы ORM имеют логическую схему, включающую проверку корректности, преобразование или мониторинг. Вы должны тестировать абстрактный код базы данных, базируемый на ORM. Проверьте, выполняют ли эти классы ожидаемые от них действия с входными данными и блокируют ли они недопустимый ввод.

Если какой-либо из ваших тестов терпит неудачу, вероятно, ваше приложение использует неверный экземпляр класса базы данных. Всегда перепроверяйте, что вы подсоединяетесь к правильно работающей базе данных: ошибка зачастую — это просто проблема соединения с неверным экземпляром класса. Измените конфигурацию, если это необходимо, и попытайтесь снова. Если вы уверены, что подсоединяетесь надлежащим образом, но необходимо изменить базу данных, то вы можете выполнить сценарий миграции (см. врезку на стр. 273) с целью синхронизации данного экземпляра класса базы данных, чтобы он мог соответствовать ожиданиям вашего приложения.

Деловая нагрузка: работа в нескольких ветвях

Во время разработки приложения вы можете работать над несколькими модификациями кода. Вы даже можете работать над различными модификациями в течение одного дня. Например, вы могли исправить неотложную ошибку в ветви приложения, развернутого в настоящий момент, а минуту спустя продолжить работу над долгосрочной разработкой в главной ветви.

Однако база данных, которую использует ваше приложение, не находится под контролем модификации. Непрактично устанавливать и удалять сиюминутные записи в базе данных, даже если модель СУБД, которую вы используете, относительно динамична и проста в использовании.

В идеале лучше создать отдельные экземпляры класса базы данных для каждой модификации приложения, которое вы должны разрабатывать, тестировать, организовывать и вводить в действие. Кроме того, каждый разработчик в вашей команде нуждается в отдельном экземпляре класса базы данных, чтобы можно было работать без надобности во взаимодействии с остальной частью группы.

Ваше приложение должно поддерживать конфигурируемые устройства, чтобы указывать параметры соединения с базой данных так, чтобы вне зависимости от модификации приложения, над которым вы работаете, вы могли определять, какую использовать СУБД без надобности в дополнительном написании кода.

В настоящее время любая модель РСУБД, как коммерческая, так и проекты с открытым исходным кодом, предлагает бесплатное решение по разработке и тестированию. Такие технологии виртуализации платформ, как VMware Workstation, Xen и VirtualBox, позволяют каждому разработчику воссоздавать клон инфраструктуры сервера за небольшую плату. Для разработчиков не существует преград в разработке и тестировании в полифункциональной среде, которая соответствует промышленной среде.

ВНИМАНИЕ!

Используйте лучшие методы разработки программного обеспечения, включая документирование, тестирование, а также систему управления исходным кодом, для вашей базы данных и вашего приложения.

Объяснения существуют; они существовали всегда; для любой волнующей человека проблемы всегда легко найти решение — простое, достижимое и ошибочное.

Г. Л. Менкен

ГЛАВА 25. ВОЛШЕБНЫЕ БОБЫ

«Почему уходит так много времени на создание одной простой функции?» — ваш руководитель поручил вашей команде улучшить приложение, отслеживающее ошибки, чтобы оно могло показывать количество полученных ей комментариев. Вы работали над этой задачей в течение четырех недель.

Ваша группа разработчиков программного обеспечения не хочет отвечать на этот вопрос. Как лидер проекта, вы отвечаете: «У нас было несколько неудачных стартов, — объясняете вы. — Проект казался простым с первого взгляда, пока мы не заметили, что в приложении было несколько других экранов, в которых необходимо было показывать счетчик комментариев.»

«И проектирование экранов заняло четыре недели?» — недоумевает руководитель.

«Вообще-то нет, это лишь малая часть HTML-кода, и она довольно проста, поскольку мы использовали каркас, отделяющий код от представления», — продолжаете вы. — «Но каждый раз, когда мы добавляли этот элемент к экрану, мы были обязаны дублировать код, чтобы выбрать данные во внутреннем коде экрана. А это означало, что каждый внутренний класс нуждался в новом наборе тестов».

«Разве вы не использовали каркас тестирования?» — спрашивает руководитель. — «Сколько времени займет написание кода еще нескольких тестов?»

«Создание тестов не так просто, как написание кода», — нерешительно отвечает другой разработчик. — «Мы даже создали сценарии для тестируемых данных. К тому же необходимо было перезагружать данные в тестовой базе данных для каждого теста. Мы также должны были тестировать клиентскую часть приложения со всеми перестановками новой функции, объединенной со старыми сценариями».

Глаза вашего руководителя начинают тускнеть, ваш сотрудник, тем временем, продолжает: «Теперь мы имеем 600 тестов для клиентской части приложения, и каждый из них запускает определенный экземпляр класса в эмуляторе браузера. Проведение всех этих тестов — это только вопрос времени, — он пожимает плечами. — И мы ничего не можем с этим поделать».

Ваш руководитель глубоко вздыхает, и говорит: «Ладно... Я не поклонник всего этого; я только хочу знать, почему так сложно добавить одну простую функцию. Разве ваш объектноориентированный каркас не сделал добавление функции более простым и быстрым, как это предполагалось?»

Хороший вопрос.

25.1. ЦЕЛЬ: УПРОЩЕНИЕ МОДЕЛЕЙ В АРХИТЕКТУРЕ MVC

Каркасы веб-приложений ускоряют и упрощают добавление функций и кода в приложение. Больше всего увеличивает стоимость программного обеспечения такой фактор, как время разработки. Следовательно, чем меньше времени мы пользуемся услугами разработчиков, тем сильнее мы уменьшаем стоимость создания программного обеспечения.

Роберт Л. Гласс (Robert L. Glass) установил, что «80 процентов работы программного обеспечения — интеллектуальная. Изрядное количество ее является творческой. И совсем немного — офисной[1]».

Один из способов нашей помощи проектированию интеллектуальной части разработки программного обеспечения состоит в том, чтобы принять терминологию и правила шаблонов проектирования. Когда мы говорим «Одиночка» (Singleton), «фасад» (Facade) или «фабрика» (Factory) — любой разработчик в нашей группе знает, что мы имеем в виду. Это экономит большое количество времени.

Бóльшая часть кода в любом приложении — это фактически повторяющийся шаблонный текст. Каркасы помогают улучшить производительность написания кода, предоставляя повторно используемые компоненты и инструменты генерации кода. Мы можем создавать рабочие приложения написанием меньшего количества исходного кода.

Шаблоны проектирования и программные каркасы объединяются, когда мы используем архитектуру MVC (Model View Controller, модель-представление-контроллер). Это методика отделения проблем приложения друг от друга, проиллюстрированная на рис. 25.1.

- *Контроллеры* принимают вводимые пользователем данные, определяет, какую работу приложение должно выполнить в ответ, поручает работу соответствующим моделям и отправляет результаты в представление.
- *Модели* обрабатывают все остальное; модели являются основой приложения; они включают проверку корректности входных данных, бизнес-логику и взаимодействие базы данных.
- *Представления* представляют информацию в пользовательском интерфейсе.

---

[1] Из книги «Facts and Fallacies of Software Engineering» [6].

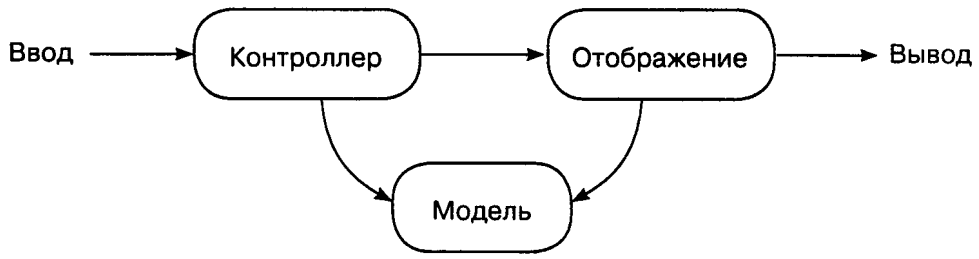


Рис. 25.1. Модель-представление-контроллер

Понять, что делают контроллеры и представления довольно просто. Но назначение моделей не совсем ясно. В сообществе разработчиков программного обеспечения существует большое желание упростить и обобщить понятие модели с целью снижения комплексности проектирования программного обеспечения. Но зачастую цель упрощения появляется из-за представления разработчиков о том, что модель — это всего лишь объект доступа к данным (Data Access Object, DAO).

25.2. АНТИПАТТЕРН: МОДЕЛЬ, ПРЕДСТАВЛЯЮЩАЯ СОБОЙ АКТИВНУЮ ЗАПИСЬ

Имея дело с простыми приложениями, вы не нуждаетесь в большой пользовательской логической схеме и модели. Относительно просто сопоставлять поля модельного объекта со столбцами единственной таблицы в базе данных. Это пример типа *объектно-реляционного отображения*. Все, что вам необходимо от объекта, чтобы он мог производить создание строк в таблице, чтение и их обновление или удаление, — таковы основные операции, имеющие аббревиатуру CRUD (Create, Read, Update, Delete).

Мартин Фаулер (Martin Fowler) описал шаблон, поддерживающий отображение, называемый *Активная запись*[1]. Активная запись — это шаблон доступа к данным. Вы определяете класс, соответствующий таблице или представлению в вашей базе данных. Можете вызвать метод класса `find()`, возвращающий объектный экземпляр класса, который соответствует отдельной строке в данной таблице или представлению. Вы также можете использовать конструктор класса для создания новых строк. Вызвав метод `save()` для данного объекта, вы можете вставить новую строку или обновить существующую.

[1] Узнайте больше о шаблоне **Активная запись** в книге *Patterns of Enterprise Application Architecture* [6], с. 160.

Файл примера: *Magic-Beans/anti/doctrine.php*

```
<?php
$bugsTable = Doctrine_Core::getTable('Bugs' );
$bugsTable->find(1234);

$bug = new Bugs();
$bug->summary = "Crashes when I save";
$bug->save();
```

ДЫРЯВЫЕ АБСТРАКЦИИ

Джоэл Спольски в 2002 году ввел термин «дырявые абстракции» (leaky abstractions)[1]. Абстракции упрощают внутреннюю работу некоторой технологии и делают эту технологию более простой в использовании. Но когда возникает ситуация, в которой вы должны знать способ сделать внутреннюю работу более производительной, то, вероятно, эта ситуация была вызвана дырявой абстракцией.

Использование шаблона Активная запись как модели в архитектуре MVC — это пример дырявой абстракции. В очень простых случаях шаблон Активная запись работает как по волшебной палочке. Но если вы пытаетесь использовать его для доступа к базе данных, вы обнаруживаете множество таких операций, как `JOIN` или `GROUP BY`, которые просты для выражения в SQL, но неудобны в шаблоне Активная Запись.

Некоторые каркасы могут улучшить Активную запись, вводя поддержку большого разнообразия операторов SQL. Чем больше этих улучшений предоставляет факт того, что класс использует SQL внутренне, тем больше вы чувствуете, что было бы лучше использовать SQL непосредственно.

Абстракция не в состоянии скрыть свои секреты, как Тото, узнающий, что Волшебник из Страны Оз — это обычный человек, прячущийся в своих ярких одеяниях.

Программный каркас Ruby on Rails популяризировал шаблон Активная запись для каркасов веб-разработки в 2004 году, и теперь большинство каркасов веб-приложений использует этот шаблон как фактический объект доступа к данным. Нет ничего плохого в использовании Активной записи — это прекрасный шаблон, обеспечивающий простой интерфейс для отдельных строк в одной таблице. Антипаттерном являются правила, унаследованные всеми моделями класса в приложениях с архитектурой MVC от базы класса Активной записи. Это пример антипаттерна **Золотой молоток**: если единственный инструмент, который у вас есть — молоток, то обработайте им все, как будто ваш материал — это гвоздь.

Было бы заманчиво охватить все правила, упрощающие проектирование программного обеспечения. Мы можем сделать нашу работу проще, если готовы пожертвовать некоторой гибкостью, а если мы особо не нуждались в гибкости с самого начала, то это даже лучше.

---

[1] Более подробную информацию узнайте в книге *The Law of Leaky Abstractions* [14].

Однако все это лишь сказка, как история о «Джеке в стране чудес». Джек верил, что его волшебные бобы за ночь вырастут в огромное дерево, пока он спит. Это сработало в истории Джека, но в реальной жизни не всем так везет. Давайте изучим последствия антипаттерна **Волшебные бобы**.

Активная Запись связывает модели со схемой

Активная запись — простой шаблон, так как простой класс Активной Записи представляет одну таблицу или представление в базе данных. Поля каждого объекта Активной записи соответствуют столбцам в соответствующей таблице. Если у вас 16 таблиц, вы определяете 16 подклассов моделей.

Это означает, что, если вы нуждаетесь в рефакторинге вашей базы данных, чтобы представить новую структуру данных, ваши классы моделей должны изменяться, так же как любая часть кода в вашем приложении, использующем классы моделей. Аналогично, если вы вводите контроллер для обработки нового экрана в приложении, вам может понадобиться продублировать код, запрашивающий модели.

Активная запись открывает функции CRUD

Следующая проблема, с которой можно столкнуться, состоит в том, что другие программисты, использующие ваш класс моделей, могут обойти намеченный вами способ использования, обновляя данные непосредственно при помощи функций CRUD.

Например, вы могли бы добавить метод `assignUser()` к модели ошибки, так как вам необходимо отправить электронное письмо разработчику после обновления ошибки.

Файл примера: *Magic-Beans/anti/crud.php*

```
<?php
class CustomBugs extends BaseBugs
{
public function assignUser(Accounts $a)
{
$this->assigned_to = $a->account_id;
$this->save();
mail($a->email, "Assigned bug",
"You are now responsible for bug #{$this->bug_id}.");
}
}
```

Однако другой программист, работающий над ошибкой, обходит ваш метод и присваивает значение ошибке вручную, без отправки электронного сообщения.

Файл примера: *Magic-Beans/anti/crud.php*

```
$bugsTable = Doctrine_Core::getTable('Bugs' );
$bugsTable->find(1234);
$bug->assigned_to = $user->account_id;
$bug->save();
```

Вашей обязанностью было отправлять уведомление по электронной почте всякий раз, когда присвоенное значение изменяется. Однако есть возможность пропустить этот шаг. Имеет ли смысл для полученного вами класса моделей представлять CRUD-методы базового класса Активной записи? Как воспрепятствовать другим программистам использовать эти методы ненадлежащим образом? Как исключить интерфейс базовой Активной записи и сгенерированной вами документации класса моделей и закодировать расширение в редакторах кода?

Активная запись поощряет анемичную модель домена

Тесно связанный с тем момент заключается в том, что модель зачастую не имеет другого поведения, кроме универсальных методов CRUD. Многие разработчики расширяют базовый класс Активной записи без добавления новых методов, связанных с работой над тем, что модель должна делать.

Интерпретация моделей как простых объектов доступа к данным поощряет вас на кодирование бизнес-логики вне модели, всегда рассеивайте множественные классы контроллеров, уменьшая связь между поведением различных моделей. Мартин Фаулер в своем блоге называет этот антипаттерн **Анемичной Моделью домена**[1]. У вас могут быть отдельные классы Активной записи, соответствующие таблицам `Bugs`, `Accounts` и `Products`. Однако во многих задачах приложения вам необходимы данные из всех трех этих таблиц.

Взгляните на простой пример кода для нашего приложения, отслеживающего ошибки, которое реализует присвоение значений ошибкам, ввод данных, отображение ошибок и задачи поиска ошибки. Оно использует PHP-каркас Doctrine, чтобы обеспечить простой интерфейс активных записей, а также каркас Zend Framework для архитектуры MVC.

[1] http://www.martinfowler.com/bliki/AnemicDomainModel.html.

Файл примера: *Magic-Beans/anti/anemic.php*

```
<?php
class AdminController extends Zend_Controller_Action
{
   public function assignAction()
   {
      $bugsTable = Doctrine_Core::getTable("Bugs");
      $bug = $bugsTable->find($_POST["bug_id"]);
      $bug->Products[] = $_POST["product_id"];
      $bug->assigned_to = $_POST["user_assigned_to"];
      $bug->save();
   }
}

class BugController extends Zend_Controller_Action
{
   public function enterAction()
   {
      $bug = new Bugs();
      $bug->summary = $_POST["summary"];
      $bug->description = $_POST["summary"];
      $bug->status = "NEW";

      $accountsTable = Doctrine_Core::getTable("Accounts");
      $auth = Zend_Auth::getInstance();
      if ($auth && $auth->hasIdentity()) {
         $bug->reported_by = $auth->getIdentity();
   }
      $bug->save();
   }

   public function displayAction()
   {
      $bugsTable = Doctrine_Core::getTable("Bugs");
      $this->view->bug = $bugsTable->find($_GET["bug_id"]);
      $accountsTable = Doctrine_Core::getTable("Accounts");
```

```
        $this->view->reportedBy = $accountsTable->find($bug-
>reported_by);
        $this->view->assignedTo = $accountsTable->find($bug-
>assigned_to);
        $this->view->verifiedBy = $accountsTable->find($bug-
>verified_by);

        $productsTable = Doctrine_Core::getTable("Products");
        $this->view->products = $bug->Products;
        }
}

class SearchController extends Zend_Controller_Action
{
   public function bugsAction()
   {
      $q = Doctrine_Query::create()
         ->from("Bugs b")
         ->join("b.Products p")
         ->where("b.status = ?", $_GET["status"])
         ->andWhere("MATCH(b.summary, b.description) AGAINST
      (?)", $_GET["search"]);
      $this->view->searchResults = $q->fetchArray();
      }
}
```

Код, использующий Активную запись в классах контроллеров, расширяется, чтобы подойти к процедурному подходу организации логики приложения. Если схема базы данных или требуемое поведение приложения в какой-то момент изменится, вам понадобится обновить множество мест в коде. Аналогично, если вы добавляете контроллер, вам необходимо написать новый код, даже если ваши запросы к модели идентичны соответствующим запросам в других контроллерах.

Схема взаимодействия класса (показанная на рис. 25.2) неаккуратна и трудна для чтения; ситуация только ухудшается, когда мы добавляем больше контроллеров и классов DAO. Это должно быть мощной подсказкой, что код, в котором используются различные модели, дублируется во всех контроллерах. Вы должны использовать другой подход, чтобы упростить и инкапсулировать часть вашего приложения.

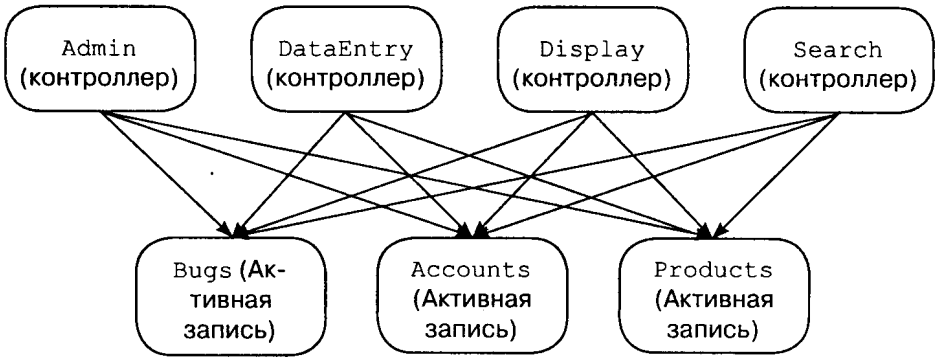


Рис. 25.2. Использование шаблона **Волшебные бобы** приводит к похожей на виноградную лозу путанице.

Поблочное тестирование при использовании антипаттерна Волшебные бобы трудно

Когда вы используете антипаттерн **Волшебные бобы**, вы находите, что тестирование каждого блока в архитектуре MVC очень трудно.

- *Тестирование модели.* С того момента, как вы сделали модель одного класса с Активной Записью, вы не можете протестировать поведение модели отдельно от доступа к данным. Чтобы протестировать модель, необходимо выполнять запрос в исполнительной базе данных.

Многие используют так называемые *приспособления базы данных* (*database fixtures*). Приспособление базы данных загружает данных в тестовую базу данных, чтобы обеспечить тестирование в базисном режиме. Выполнение очень сложной установки и удаления делает тестовые модели медленными и подверженными ошибкам, так же как требование исполнительной базы данных для тестов.

- *Тестирование представления.* Тестирование представлений включает рендеринг представления в HTML-код и парсинг результата в целях проверки корректности динамических HTML-элементов, обеспечиваемых моделями, отображающимися в выводе. Даже если используемый вами каркас упрощает утверждения в сценариях тестирования, он должен выполнить комплексный и отнимающий большое количество времени прогон кода, чтобы выполнить рендеринг и затем проанализировать HTML-код в отношении определенных элементов.
- *Тестирование контроллера.* Вы обнаружите также, что тестирование контроллера трудно, поскольку модель, являющаяся объектом доступа к данным, приводит к повторениям одних и тех же фрагментов кода в множественных контроллерах, каждый из которых нуждается в тестировании.

Чтобы протестировать контроллер, вам необходимо создать ложный HTTP-запрос. Вывод веб-приложения — это заголовок идентификации блока данных и тело. Чтобы проверить корректность теста, вы должны изучить HTTP-ответ, который возвращает контроллер. Это требует большого количества установочного кода, чтобы протестировать бизнес-логику, и делает выполнение тестов медленным.

Если бы вы могли отделить бизнес-логику от доступа к базе данных, а также от представления, это могло бы помочь в достижении целей архитектуры MVC и сделало бы тестирование более простым.

25.3. СПОСОБЫ РАСПОЗНАВАНИЯ АНТИПАТТЕРНА

Следующие утверждения могут означать, что вы имеете дело с антипаттерном **Волшебные бобы**.

- «Как мне передать пользовательский SQL-запрос в модель?»

Вопрос предполагает, что вы используете класс доступа к базе данных как класс моделей. Вам не придется предавать SQL-запросы в модель — класс модели должен инкапсулировать любой необходимый запрос.

- «Могу ли я скопировать сложные запросы моделей во все мои контроллеры или я должен кодировать их для каждого абстрактного контроллера?»

Ни одно из этих решений не даст вам стабильность и простоту, которые вы ищете. Вы должны кодировать комплексные запросы, представленные как часть интерфейса модели, в пределах модели. Таким образом, вы будете следовать принципу *DRY* (*Don't Repeat Yourself, «не повторяй самого себя»*) и сделаете использование моделей более простым[1].

- «Я должен создавать больше приспособлений базы данных до тех пор, пока не протестирую свои модели».

Если вы используете приспособления базы данных, вы тестируете вход в базу данных, а не бизнес-логику. Вы должны быть готовы к поблочному тестированию модели изолированно от базы данных.

25.4. ДОПУСТИМЫЕ СПОСОБЫ ИСПОЛЬЗОВАНИЯ АНТИПАТТЕРНА

По существу в шаблоне проектирования Активная запись нет ничего неправильного. Это удобный шаблон для простых CRUD-операций. В большинстве приложений существуют некоторые случаи, в которых необходим только простой объект доступа к данным для простых операций в опреде-

[1] Принцип DRY был описан в книге «The Pragmatic Programmer» [11] авторов Анди Ханта (Andy Hunt) и Дейва Томаса (Dave Thomas).

ленных строках таблицы. Вы можете упростить эти случаи, определяя модель, как совпадающий с ней DAO-объект.

Другой хороший способ использования шаблона Активная запись — это создание прототипа кода. Быстрое написание кода более важно, чем написание кода, готового к тестированию и обслуживанию, сокращения крайне важны. Демонстрация рабочего прототипа раньше и чаще — это отличный способ совершенствовать проект с активной обратной связью. Все, что вы можете сделать для ускорения разработки прототипа полезно при таких обстоятельствах, использование простых каркасов приложения также может помочь в данном случае.

Только убедитесь, что запланировали некоторое время для рефакторинга кода, чтобы отплатить технический долг, который вы берете на себя, начав писать код в режиме создания прототипа.

25.5. РЕШЕНИЕ: МОДЕЛЬ С АКТИВНОЙ ЗАПИСЬЮ

Контроллеры обрабатывают прикладной ввод, а представления — прикладной вывод, обе эти задачи являются относительно простыми и четкими. Каркасы лучше всего подходят для помощи в быстром объединении этих задач. Но каркасы слабо поддерживают решение «одно-для-всех» в отношении моделей, поскольку модели содержат остальную часть объектно-ориентированного проектирования вашего приложения.

Это тот случай, когда вы должны серьезно подумать о том, какие объекты находятся в вашем приложении и какие данных и поведение эти объекты имеют. Помните о том, что Роберт Л. Гласс оценил, что большая часть разработки программного обеспечения является интеллектуальным и творческим процессом?

Управление моделью

К счастью, в поле объектно-ориентированного проектирования существует много мудрости, которая может вести вас. Книга Крейга Лармана (Craig Larman) «Applying UML and Patterns» [12], к примеру, описывает рекомендации, называемые *GRASP* (*General Responsibility Assignment Software Patterns, общие шаблоны распределения обязанностей*). Некоторые из этих рекомендаций специально предназначены для отделения моделей от их объектов доступа к данным:

Информационный эксперт (Information Expert)

Объект, ответственный за операцию, должен иметь *все* необходимые данные, чтобы выполнить эту операцию. Если некоторые операции в вашем приложении содержат множественные таблицы (или не содержат их вооб-

ще), а Активная запись способна работать только с одной таблицей за раз, то вы нуждаетесь в другом классе, который объединит несколько объектов доступа к базе данных вместе и использует их для составной операции.

Связь между моделью и объектом DAO, как и Активная запись, должна быть HAS-A (агрегирование), а не IS-A (наследование). Многие каркасы, полагающиеся на Активную запись, предлагают IS-A-решение. Если ваша модель использует объекты DAO, вместо наследования от класса DAO, вы можете проектировать модель так, чтобы она содержала все данные и код домена, который предположительно моделируется, — даже если для представления этого понадобится множество таблиц базы данных.

Создатель (Creator)

Способ сохранения моделью данных в базе данных должен являться внутренней деталью реализации. Модель домена, агрегирующая свои объекты DAO, должна отвечать за создание этих объектов.

Контроллеры и представления в вашем приложении должны использовать интерфейс модели домена, не будучи осведомленными о том, какой вид взаимодействия базы данных необходим для модели, чтобы выбирать и хранить данные. Это упрощает изменение запросов к базе данных впоследствии, в одном участке вашего приложения.

Слабая связанность (Low Coupling)

Важно расцепить логически независимые блоки кода. Это даст гибкость в изменении реализации класса, не затрагивая потребителей. Вы не можете упростить требования приложения; некоторая степень комплексности постоянно должна присутствовать в некоторых участках кода. Но вы можете сделать наилучший выбор по поводу того, где именно будет применена эта комплексность.

Сильное зацепление (High Cohesion)

Интерфейс класса модели домена должен отражать предназначенное ему использование, а не физическую структуру базы данных или CRUD операции. Универсальные методы интерфейса Активной записи, например, `find()`, `first()`, `insert()` или даже `save()`, не многое скажут вам о своих требованиях к применению в приложении. Такие методы, как `assignUser()`, более описательны, следовательно контроллер кода более прост в понимании.

Когда вы расцепляете класс модели от объектов DAO, которые он использует, вы можете даже спроектировать более одного класса модели для одно-

го и того же объекта DAO. Лучше всего для зацепления подходит ситуация, когда пытаются объединить всю работу, связанную с данными таблицами, в один класс, работающий по шаблону Активная запись.

Помещение модели домена в операцию

В книге Эрика Эванса (Eric Evans) *Domain-Driven Design: Tackling Complexity in the Heart of Software* [5] описывается лучшее решение: *модель домена.*

Модель в значении исходной архитектуры MVC — не в значении самоуверенного программного обеспечения — это объектноориентированное отображение *домена* в вашем приложении, то есть в бизнес-правилах вашего приложения и в данных этих бизнес-правил. Модель — это тот объект, в котором вы реализуете бизнес-логику приложения; хранение ее в базе данных является внутренней деталью реализации модели.

Как только мы получаем модель, спроектированную вокруг концепций приложения, а не плана базы данных, вы может начать реализацию вариантов базы данных, скрытых в классах моделей. Давайте посмотрим на возможный рефакторинг показанного выше фрагмента кода:

Файл примера: *Magic-Beans/soln/domainmodel.php*

```
<?php

class BugReport
{
   protected $bugsTable;
   protected $accountsTable;
   protected $productsTable;

   public function __construct()
   {
      $this->bugsTable = Doctrine_Core::getTable("Bugs");
      $this->accountsTable=Doctrine_Core::getTable("Accounts");
      $this->productsTable=Doctrine_Core::getTable("Products");
   }

   public function create($summary, $description, $reportedBy)
   {
      $bug = new Bugs();
      $bug->summary = $summary
      $bug->description = $description
```

```
      $bug->status = "NEW";
      $bug->reported_by = $reportedBy;
      $bug->save();
   }

   public function assignUser($bugId, $assignedTo)
   {
      $bug = $bugsTable->find($bugId);
      $bug->assigned_to = $assignedTo"];
      $bug->save();
   }

   public function get($bugId)
   {
      return $bugsTable->find($bugId);
   }

   public function search($status, $searchString)
   {
      $q = Doctrine_Query::create()
         ->from("Bugs b")
         ->join("b.Products p")
         ->where("b.status = ?", $status)
         ->andWhere("MATCH(b.summary,  b.description)  AGAINST
(?)", $searchString]);
      return $q->fetchArray();
   }
   }

   class AdminController extends Zend_Controller_Action
   {
      public function assignAction()
   {
         $this->bugReport->assignUser(
            $this->_getParam("bug"),
            $this->_getParam("user"));
      }
   }
```

```
   class BugController extends Zend_Controller_Action
   {
      public function enterAction()
      {
         $auth = Zend_Auth::getInstance();
         if ($auth && $auth->hasIdentity()) {
            $identity = $auth->getIdentity();
   }
         $this->bugReport->create(
             $this->_getParam("summary"),
             $this->_getParam("description"),
            $identity);
      }

   public function displayAction()
   {
      $this->view->bug = $this->bugReport->get(
         $this->_getParam("bug"));
   }
}

class SearchController extends Zend_Controller_Action
{
   public function bugsAction()
   {
      $this->view->searchResults = $this->bugReport->search(
         $this->_getParam("status", "OPEN"),
         $this->_getParam("search"));
   }
}
```

Вы должны быть готовы обратить внимание на несколько улучшений.

- Схема взаимодействия классов (изображенная на рис. 25.3) намного более проста и удобна для чтения, что указывает на усовершенствование, к которому приводит расцепление классов.
- Расцепляя интерфейс модели от основной структуры базы данных, мы уменьшаем и упрощаем код контроллера.

- Каждый класс модели создает объекты, чтобы взаимодействовать с одной или более таблицами. Контроллерам нет необходимости знать, какие таблицы участвуют в этом взаимодействии.
- Классы моделей инкапсулируют и скрывают запросы к базе данных. Контроллер заинтересован только в поиске пользовательских вводов и выводов задач более высокого уровня через API-интерфейс модели.
- В некоторых случаях запрос может быть слишком сложным, чтобы легко пройти через взаимодействие с DAO-объектом, в связи с чем требуется написание пользовательского SQL-кода.

Тестирование простых объектов

В идеале нужно суметь протестировать модель, не подключаясь к оперативной базе данных. Если вы расцепляете модель от ее DAO-объекта, то вы можете создать *заглушки* и *фиктивные* DAO-объекты, чтобы содействовать поблочному тестированию вашей модели.

Аналогично вы можете тестировать интерфейс модели домена точно так же, как в любом другом объектноориентированном тестировании: вызовите метод объекта, а затем проверьте корректность возвращаемого методом значения. Это быстрее и проще, чем создание ложных HTTP-запросов, полагаясь на контроллер и парсинг результирующего ответа HTTP-протокола.

Вы по-прежнему тестируете контроллеры ложными HTTP-запросами, но поскольку код контроллера стал проще, больше нет необходимости в тестировании большого количества логических путей.

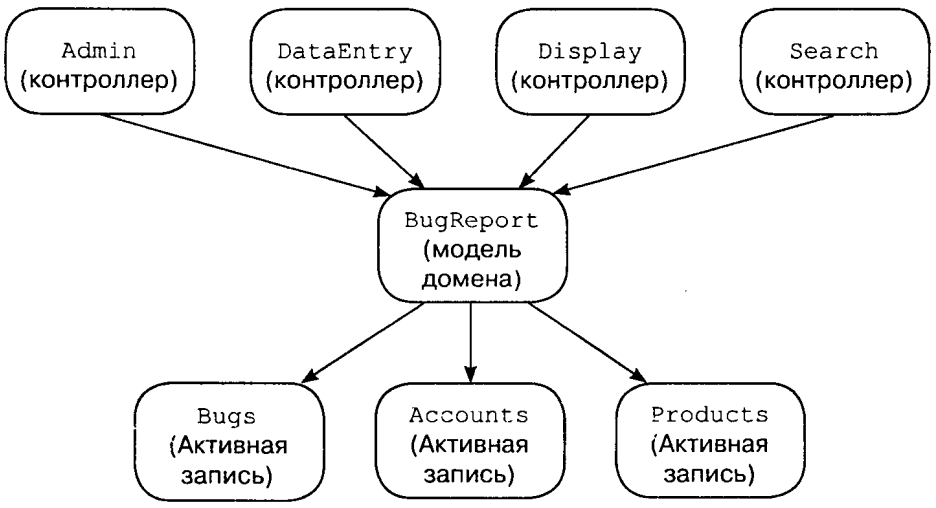


Рис. 25.3. Распутывание виноградных лоз при помощи расцепления

Если вы отделяете модели и контроллеры, а также компоненты доступа к данным от моделей, вы можете провести поблочное тестирование всех этих классов более просто и с лучшей изолированностью. Это облегчает диагностику дефектов, когда они происходят. Разве это не преимущество поблочного тестирования?

Спуск на землю

Вы можете использовать объект доступа к данным продуктивно в любом каркасе разработки программного обеспечения, даже в каркасе, поощряющем антипаттерн **Волшебные бобы**. Однако разработчики, не изучившие, как использовать принципы объектноориентированного проектирования, обречены на написание запутанного кода.

Основы моделирования домена, описанные и процитированные в этой главе, помогут вам выбрать наилучший метод проектирования, чтобы поддержать тестирование и обслуживание кода. Вы, наконец, сможете достигнуть высокой производительности в разработке приложений, управляемых СУБД.

ВНИМАНИЕ!

Расцепляйте модели и таблицы.

ЧАСТЬ V. ПРИЛОЖЕНИЯ

В математике не усваивают понятий, а привыкают к ним.

Джон фон Нейман

ПРИЛОЖЕНИЕ А. ПРАВИЛА НОРМАЛИЗАЦИИ

Реляционное проектирование базы данных не является случайным или непостижимым. Вы можете пользоваться рядом четких правил для проектирования стратегии хранения данных, избегающей дублирования и помогающей сделать приложение защищенным от ошибок, как это реализовано в идеях защиты от дурака (пока-екэ), упомянутых ранее в этой книге. Вы, вероятно, слышали и другие метафоры для этой идеи, например, *защитное проектирование* или *ошибись раньше.*

Правила нормализации не сложны, но довольно хитры. Разработчики зачастую неправильно понимают принципы их работы, возможно, потому что они ожидают, что правила будут сложнее, чем они есть.

Другая причина неудач может заключаться в том, что программисты вообще не стремятся следовать правилам. Правила — это проклятье разработчиков, ценящих новизну, креативность и новаторство. Правила являются противоположностью свободы.

Разработчики программного обеспечения непрерывно ищут компромиссы между простотой и гибкостью. Вы можете создать для себя много лишней работы, повторно изобретая колесо и разрабатывая программное обеспечение управления пользовательскими данными для каждого приложения. Но вы можете и воспользоваться преимуществом существующих знаний и технологий, если будете работать в соответствии с требованиями реляционного проектирования.

Я описал антипаттерны в этой книге, показывая их достоинства (или недостатки), стараясь избегать научности и теоретики. Однако в этом приложении мы увидим, что теория может быть практична.

А.1. ЧТО ЗНАЧИТ «РЕЛЯЦИОННЫЙ»?

В данном случае термин «реляционный» не относится к связям между таблицами. Он касается самой таблицы или, точнее, связей между столбцами в пределах таблицы. В каком-то смысле он затрагивает и то и другое.

Математики определяют понятие *реляция* (*реляционное отношение*) как комбинацию двух наборов значений из различных доменов под определенным количеством условий, дающую подмножество всех возможных комбинаций.

Например, пусть один набор состоит из названий бейсбольных команд, а другой — из названий городов. Комбинирование команда-город представляет собой долгий список возможных сочетаний. Однако нас интересует специфическое подмножество из этого списка: пары, каждая из которой представляет собой название команды и ее родного города. Правильные пары — это: Чикаго/White Sox, Чикаго/Cubs или Бостон/Red Sox, но не Майами/Red Sox.

Термин *реляция* используется в двух случаях: как правило («данный город является родным для данной команды») и как подмножество пар, выполняющих это правило. В SQL можно хранить этот результат в таблице с двумя столбцами, где одна строка будет соответствовать одной паре.

Разумеется, реляции поддерживают больше двух столбцов. Вы можете комбинировать любое количество доменов (по одному на один столбец) в реляцию. Кроме того, вы можете использовать домены как набор 32-разрядных целых чисел или набор текстовых строк определенной продолжительности.

Прежде чем начать нормализацию таблицы, необходимо убедиться, что они имеют надлежащие отношения. Нужно, чтобы таблицы отвечали нескольким критериям.

Строки не упорядочены сверху вниз

В SQL запрос будет возвращать результаты в непредсказуемом порядке, если вы не будете использовать оператор `ORDER BY`, чтобы указать порядок. Но, не считая порядка, набор строк — это то же самое.

Столбцы не упорядочены слева направо

Вне зависимости от того, просим мы Стивена протестировать продукт Open RoundFile на предмет ошибки № 1234 или хотим узнать была ли ошибка № 1234 проверена Стивеном в продукте Open RoundFile, результат должен быть одинаковым.

Это связано с антипаттерном в главе 19, в котором столбцы использовались не по названию, а по позиции.

Дублирование строк запрещено

Как только вам становится известен факт, его повторная констатация не сделает его более истинным. Известно название бейсбольной команды, СУБД находит название соответствующего города. Это означает, что город *зависит от* названия команды.

Для предотвращения дублирования необходимо отличать одну строку от другой и обращаться к определенным строкам. Чтобы гарантировать это в SQL, мы задаем ограничение первичного ключа для столбца или набора столбцов, вне зависимости от того, что необходимо, чтобы однозначно определить строки.

Мы могли бы получить дублирование в столбцах, не имеющих ключей, — в Бостоне две бейсбольные команды, — однако строка в целом по-прежнему уникальна, поскольку названия команд различны.

Каждый столбец имеет один тип и одно значение на строку

Реляция имеет *заголовок*, определяющий названия и типы данных столбцов. Каждая строка должна иметь те же столбцы, что указаны в заголовке, а указанный столбец должен иметь одно значение во всех строках.

Мы видим, что антипаттерн нарушает это правило двумя способами в главе 6.

Во-первых, EAV-таблица моделирует сущность, которая может иметь пользовательский набор атрибутов для каждого экземпляра класса, поэтому сущность не будет связана с заголовком, определяющим ее атрибуты.

Во-вторых, EAV-столбец `attr_value` содержит все атрибуты сущности, такие, как статус ошибок, учетная запись, с которой было произведено присвоение значения ошибке и т.д. Данное значение, например 1234, в этом столбце может быть корректным для двух различных атрибутов, но иметь совершенно другой смысл.

Антипаттерн в главе 7 также нарушает это правило, так как данное значение (1234) ссылается на первичный ключ какой-либо из родительских таблиц. Нельзя сказать, что 1234 в одной строке означает то же самое, что 1234 в другой строке.

Строки не имеют скрытых компонентов

Столбцы содержат значения данных, а не физические индикаторы памяти, например, идентификационные номера строк или объектов. Выше, в главе 22, мы убедились, что первичные ключи уникальны, но они не являются номерами строк.

Некоторые СУБД избегают этого правила, предоставляя вам доступ к деталям внутренней памяти при помощи расширений SQL (например, псевдостолбец `ROWNUM` СУБД Oracle или псевдостолбец `CID` в СУБД PostgreSQL). Однако эти значения не являются, по сути, частью реляции.

А.2. МИФЫ О НОРМАЛИЗАЦИИ

Тяжело найти тему, которая так часто неправильно понимается, несмотря на наличие четкого определения. Вы обязательно встретитесь с разработчиками, выражающими твердую, но ошибочную убежденность. Вот некоторые из их возможных высказываний.

- «Нормализация делает работу СУБД медленной. Денормализация ее ускоряет».

  Ложь. Это правда, что вам могут понадобиться соединения для поиска атрибутов из отдельных таблиц после применения нормализации. Если вы денормализуете таблицу, то сможете избежать некоторого количества соединений.

  Например, разделенный запятыми список в главе 2 показывает продукты с данной ошибкой. Но что, если вам также необходим список ошибок для данного продукта? Денормализация, как правило, повышает удобство или производительность, но расплачиваясь за этой высокой ценой — некоторыми другими типами запросов.

  Существуют и оправданные способы использования денормализации. Но для начала вы должны смоделировать свою базу данных в нормальной форме, прежде чем приступить к денормализации. Руководство по индексированию MENTOR в главе 13 применяется также и к денормализации: измерьте производительность до и после применения изменения и выберите вариант в пользу эффективности.

- «Нормализация — это когда рассылаешь данные по дочерним таблицам и ссылаешься на них при помощи псевдоключа».

  Ложь. Вы можете использовать псевдоключи ради удобства, производительности или эффективности накопления данных — эти причины оправданы. Но не верьте, что они каким-либо образом относятся к нормализации.

- «Нормализация происходит тогда, когда вы разделяете атрибуты на столько, на сколько это возможно, например, способ проектирования «объект-атрибут-значение».

  Ложь. Для разработчиков довольно характерно использовать слово нормализация в недопустимом значении, подразумевая, что она снижает удобочитаемость данных или делает их менее удобными для создания запросов. На самом деле верно обратное.

- «Нет необходимости нормализовать третью нормальную форму. Другие нормальные формы настолько малопонятны, что вы вряд ли когда-либо встретитесь с ними».

Ложь. Исследование показало, что более 20% деловых баз данных имеют дизайн, удовлетворяющий первым трем нормальным формам, но нарушающим четвертую. Это меньшинство, но ничего не значащим его назвать нельзя. Если бы вы узнали, что существует ошибка, потенциально приводящая к потере данных и происходящая в 20% ваших приложений, вы бы не захотели исправить ее?

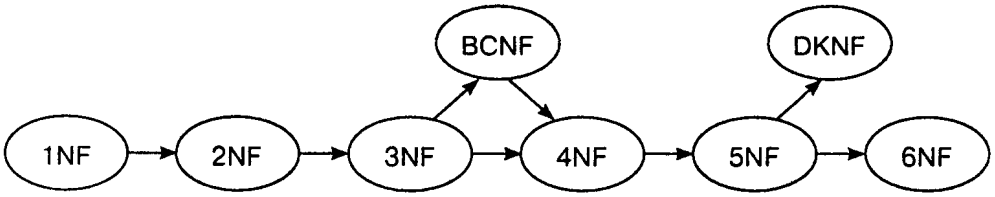


Рис. А.1. Последовательность нормальных форм.

А.3. ЧТО ТАКОЕ НОРМАЛИЗАЦИЯ?

Цели нормализации:

- представление фактов о реальном мире в понятном виде;
- уменьшение накопления избыточных фактов и предотвращение аномальных или противоречивых данных;
- поддержка ограничений целостности.

Обратите внимание, что улучшение производительности базы данных в этот список не вошло. Нормализация помогает хранить данные *правильно* и избегать неприятностей. Фактически неизбежно, что ненормализованная база данных превратится в кашу. Мы обнаружим, что разрабатываем намного больше кода, чтобы подчистить противоречивые и дублирующиеся данные. Будут происходить временные задержки и лишние расходы в бизнесе из-за дефектных данных. Если же принять эти сценарии, выигранная, благодаря нормализации, производительность базы данных станет более чистой.

Когда таблица удовлетворяет правилам нормализации, то говорят, что таблица находится в *нормальной форме*. Существует пять традиционных нормальных форм, описывающих прогрессивные уровни нормализации. Каждая нормальная форма устраняет определенный вид дублирования или аномалии, когда вы проектируете реляцию. Обычно, если ваша таблица удовлетворяет нормальной форме, таблица тоже удовлетворяет всем предыдущим нормальным формам. Существует три дополнительных нормальных формы, описанных исследователями. Последовательность нормальных форм показана на рис. А.1.

Первая нормальная форма

Самое основное требование для первой нормальной формы заключается в том, что таблица должна быть реляционной. Если она не отвечает критериям реляции, описанным в первом разделе, то ваша таблица не может находиться в первой нормальной форме или даже ни в одной из последующих форм.

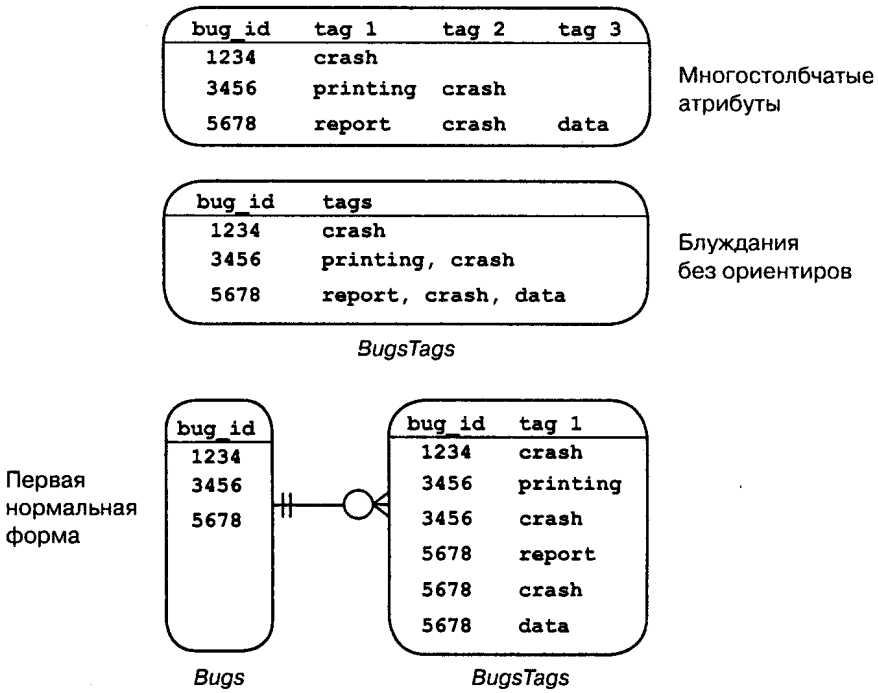


Рис. А.2. Повторение групп по сравнению с первой нормальной формой

Следующее требование заключается в том, что таблицы не должны иметь *повторяющихся групп*. Запомните, что каждая строка в реляции представляет собой комбинацию нескольких наборов, то есть вы выбираете одно значение из каждого набора. Термин «повторяющаяся группа» означает, что одна строка может иметь несколько значений в данном наборе.

Мы знаем два антипаттерна, создающих повторяющиеся группы:

- Множественные значения в одном домене в нескольких столбцах, глава 8,
- Множественные значения в одном столбце, глава 2.

На рис. А.2 изображены повторяющиеся группы, соответствующие каждому из этих антипаттернов. Надлежащий дизайн, удовлетворяющий первой

нормальной форме, будет представлять собой создание отдельной таблицы. Теги теперь находятся в отдельной таблице и мы можем поддерживать множество тегов, храня по одному тегу для каждой строки.

Вторая нормальная форма

Вторая нормальная форма идентична первой, если ваша таблица не имеет составного первичного ключа. Давайте проследим в примере с тегами, какой пользователь хотел применить тег к каждой ошибке.

Файл примера: *Normalization/2NF-anti.sql*

```
CREATE TABLE BugsTags (
   bug_id   BIGINT NOT NULL,
   tag   VARCHAR(20) NOT NULL,
   tagger   BIGINT NOT NULL,
   coiner   BIGINT 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)
);
```

На рис. А.3 можно увидеть, что идентификаторы столбца `coiner` дублируются[1]. Это означает, что кто-то мог создать *аномалию*, заменив идентификатор в одной строке для данного тега, но не изменив при этом остальные строки с тем же тегом.

Чтобы удовлетворить второй нормальной форме, необходимо сохранять значение в столбце только однажды. Это значит, что нужно определить другую таблицу, `Tags`, в которой тег будет являться первичным ключом, таким образом, одна строка будет связана с одним тегом. Затем можно будет сохранить значение данного столбца с определенным тегом в новой таблице, вместо таблицы `BugsTags`, и избежать аномалий.

Файл примера: *Normalization/2NF-normal.sql*

```
CREATE TABLE Tags (
   tag   VARCHAR(20) PRIMARY KEY,
   coiner   BIGINT NOT NULL,
```

[1] В примере используются имена вместо идентификационных номеров для облегчения идентификации пользователями.

```
   FOREIGN KEY (coiner) REFERENCES Accounts(account_id)
);

CREATE TABLE BugsTags (
   bug_id   BIGINT NOT NULL,
   tag   VARCHAR(20) NOT NULL,
   tagger   BIGINT 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)
);
```

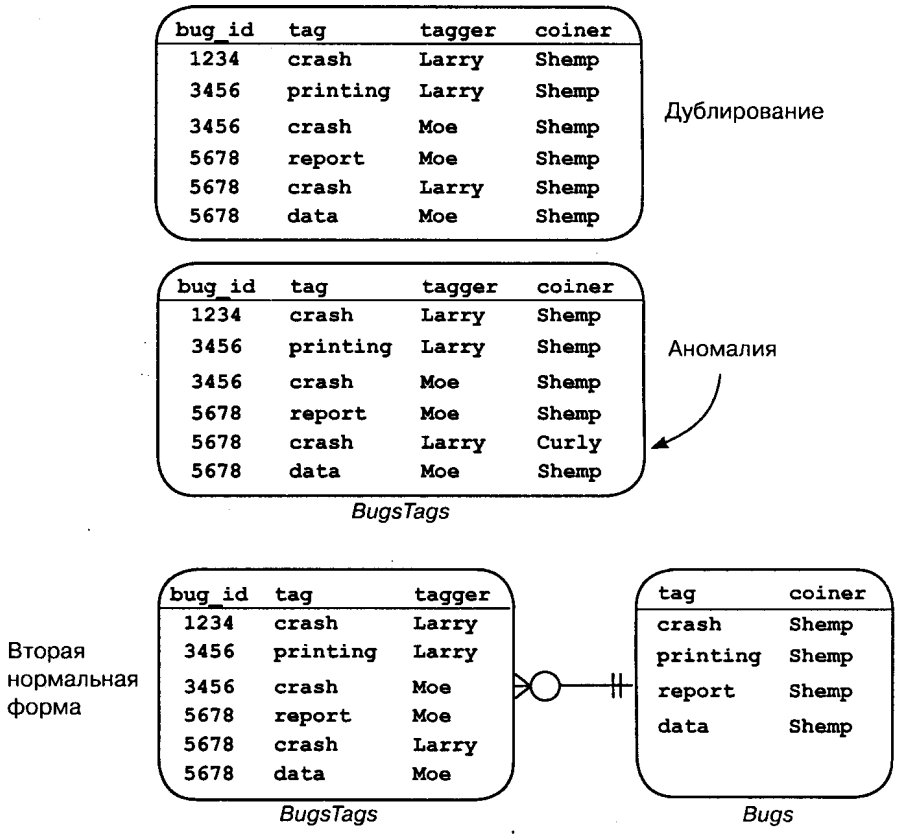


Рис. А.3. Дублирование по сравнению со второй нормальной формой

Третья нормальная форма

В таблице Bugs вам может понадобиться сохранить адрес электронной почты, разработчика, работавшего над ошибкой.

Файл примера: *Normalization/3NF-anti.sql*

```
CREATE TABLE Bugs (
   bug_id SERIAL PRIMARY KEY
   -- . . .
   assigned_to BIGINT,
   assigned_email VARCHAR(100),
   FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id)
);
```

Однако электронная почта — это атрибут, присвоенный учетной записи разработчика; он не является атрибутом ошибки. Храня адрес электронной почты таким образом, мы получим дублирование, вследствие чего появляется риск возникновения аномалий, как в таблице, не соответствующей второй нормальной форме.

В примере второй нормальной формы нарушающий нормальную работу столбец связан с последней частью составного первичного ключа. В примере нарушения третьей нормальной формы столбец-нарушитель совершенно не соответствует первичному ключу.

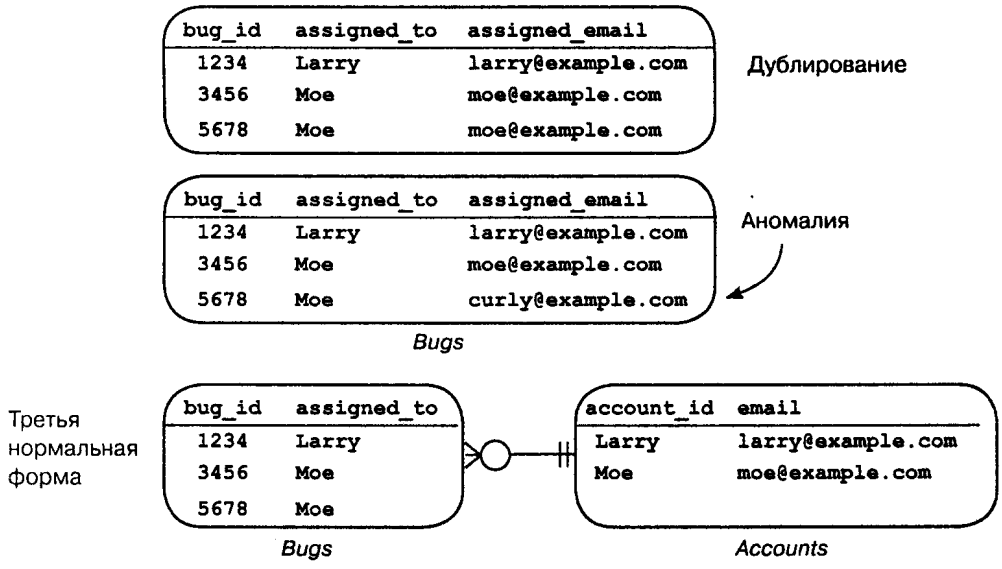


Рис. А.4. Дублирование по сравнению с третьей нормальной формой

Чтобы исправить это, необходимо ввести адрес электронной почты в таблицу `Accounts`. Посмотрите, как отделить столбец от таблицы `Bugs` на рис. А.4. Таблица `Accounts` — правильное место хранения, поскольку адрес электронной почты напрямую согласуется с первичным ключом этой таблицы и не создается дублирование.

Нормальная форма Бойса-Кодда

Наименее строгая версия третьей нормальной формы называется нормальной формой Бойса-Кодда. Разница между двумя этими формами заключается в том, что в третьей форме все неключевые атрибуты должны зависеть от ключа в таблице. В нормальной форме Бойса-Кодда ключевые столбцы являются основным предметом правила. Данная форма становится актуальной, когда таблица имеет множество наборов столбцов, которые *могут* служить ключом таблицы.

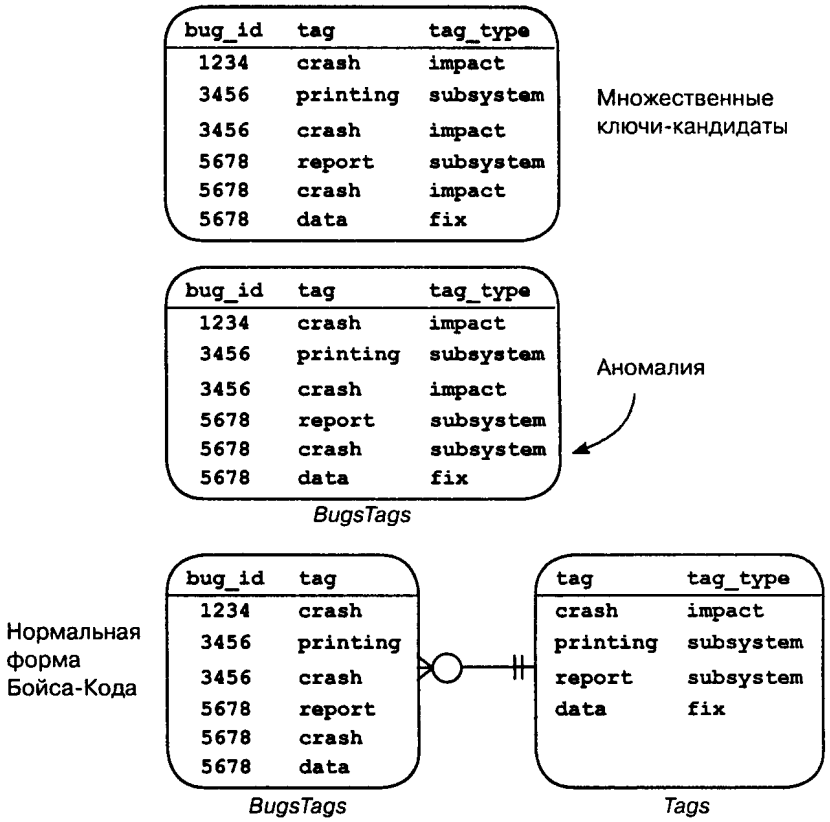


Рис. А.5. Третья нормальная форма в сравнении с нормальной формой Бойса-Кодда

Для примера, давайте представим, что у нас есть три типа тегов: теги, описывающие воздействие ошибки, теги подсистемы, на которую влияет ошибка, и теги, описывающие, как исправить ошибку. Наш возможный ключ может быть столбцом `bug_id`, спаренный со столбцом `tag`, но он также может быть спаренным со столбцом `tag_type`. Любая пара столбцов будет достаточно определенной, чтобы адресовать каждую отдельную строку.

На рис. 5 можно увидеть пример таблицы, соответствующей третьей нормальной форме, но не нормальной форме Бойса-Кода, также в ней описано, как это изменить.

Четвертая нормальная форма

Теперь давайте изменим базу данных так, чтобы об ошибках могли сообщить многочисленные пользователи, ошибкам присваивались значения многочисленными программистами, корректность исправлений могла проверяться многочисленными квалифицированными разработчиками. Мы знаем, что отношение «множество-множество» требует создания отдельной таблицы:

Файл примера: *Normalization/4NF-anti.sql*

```
CREATE TABLE BugsAccounts (
   bug_id   BIGINT NOT NULL,
   reported_by    BIGINT,
   assigned_to    BIGINT,
   verified_by    BIGINT,
   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)
);
```

Мы не можем использовать столбец `bug_id` без привязки к первичному ключу. Необходимо большое количество строк для каждой ошибки, поэтому мы можем поддерживать некоторое количество учетных записей в каждом столбце. Мы также не можем присвоить первичный ключ первым двум или трем столбцам, поскольку не поддерживаются множественные значения в последнем столбце. Поэтому первичный ключ необходимо присвоить всем четырем столбцам. При этом столбцы `assigned_to` и `verified_by` должны быть способны иметь значение `NULL`, поскольку об ошибке может быть сообщено до того, как ей будет присвоено значение или будет проверена ее корректность. Все столбцы с первичным ключом стандартно имеет ограничение `NOT NULL`.

Другая проблема состоит в том, что могут появиться избыточные значения, когда некоторый столбец содержит меньше учетных записей, чем другой столбец. Пример избыточных значений показан на рис. А.6.

Все проблемы, описанные ранее, вызываются при попытке составить перекрестную таблицу, которая имеет двойной или тройной режим работы. Когда вы пытаетесь использовать одну перекрестную таблицу для представления нескольких связей «множество-множество», то противоречите четвертой нормальной форме.

Рисунок показывает, как можно разрешить эту ситуацию, разбивая таблицу так, чтобы получить по одной перекрестной таблице для каждого типа связей множество-множество. Это решает проблему дублирования и несоответствия номеров значений в каждом столбце.

Файл примера: *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)
);
```

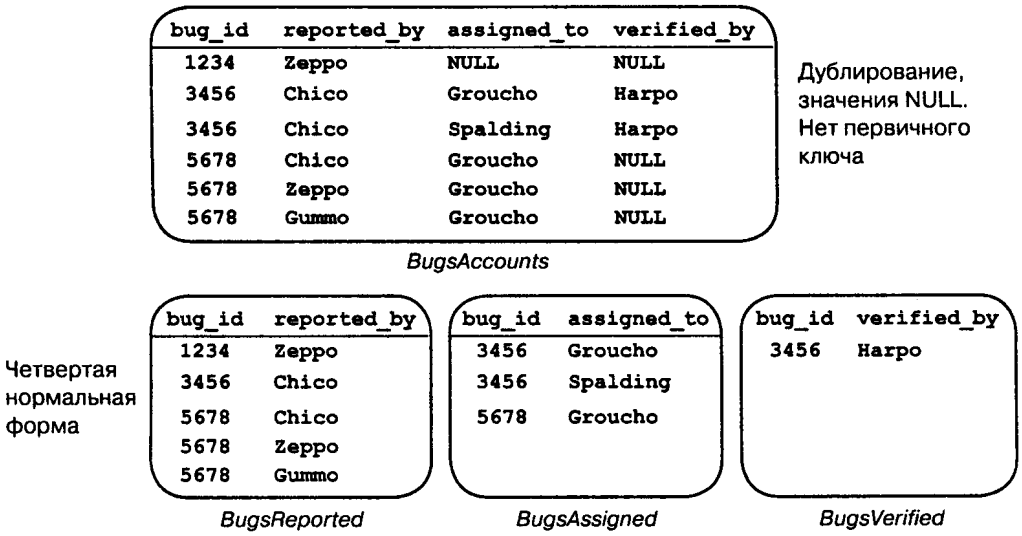


Рис. А.6. Объединенные связи в сравнении с четвертой нормальной формой

Пятая нормальная форма

Любая таблица, отвечающая критериям нормальной формы Бойса-Кодда и не имеющая составного первичного ключа, соответствует требованиям пятой нормальной формы. Но чтобы понять ее суть, давайте разберем пример.

Некоторые разработчики работают только над определенными продуктами. Необходимо разработать базу данных так, чтобы знать факты о том, кто над каким продуктом и ошибкой работает, с минимумом дублирования информации. Наша первая попытка в поддержке этого состоит в добавлении столбца к таблице `BugsAssigned`, чтобы показать, какой разработчик работает над продуктом:

Файл примера: *Normalization/5NF-anti.sql*

```
CREATE TABLE BugsAssigned (
   bug_id   BIGINT NOT NULL,
   assigned_to    BIGINT NOT NULL,
   product_id     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),
   FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

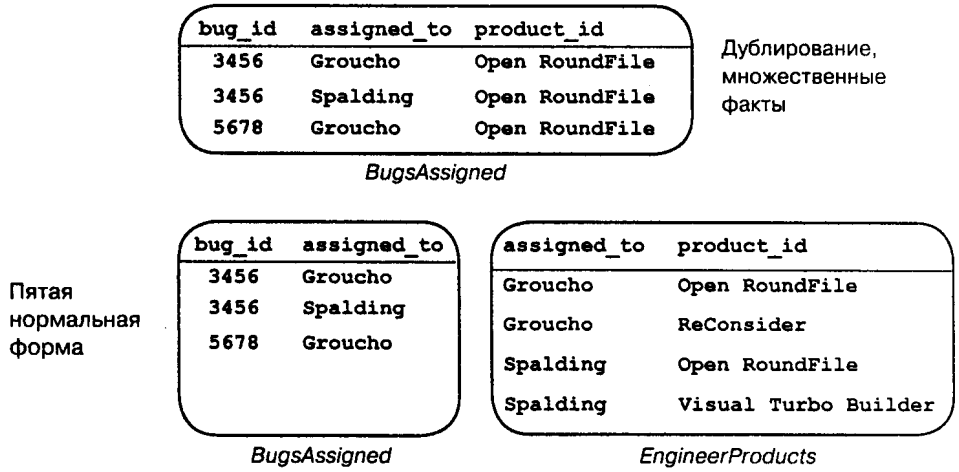


Рис. А.7. Объединенные связи в сравнении с пятой нормальной формой

Это не скажет нам, какие продукты мы можем назначить для продолжения работы разработчику; мы можем только узнать, какие продукты назначены разработчику в данный момент. Мы также можем узнать, что разработчик работает над данным продуктом повторно. Это вызвано попыткой хранить множественные факты о независимых связях «множество-множество» в одной таблице, подобную проблему мы видели в четвертной нормальной форме. Дублирование проиллюстрировано на рис. А.7[1].

Решение состоит в том, чтобы распределить все связи по отдельным таблицам, изолируя их друг от друга:

Файл примера: *Normalization/5NF-normal.sql*

```
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),
   FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

[1] В рисунке используются имена вместо идентификационных номеров.

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

Теперь можно указать факт того, что конкретный инженер может работать над данным продуктом, вне зависимости от того, что он работает над данной ошибкой этого продукта.

Другие нормальные формы

Доменно-ключевая нормальная форма (*Domain-Key normal form, DKNF*) утверждает, что каждое ограничение в таблице — это логический результат ограничений домена таблицы и ключевых ограничений. Третья, четвертая и пятая нормальные формы, а также нормальная форма Бойса-Кодда охватываются формой DKNF.

Например, можно решить, что ошибка, имеющая статус *NEW* или *DUPLICATE*, не привела к какой-либо работе, в связи с чем не должно быть зарегистрировано значений в столбце `hours`, а также не имеет смысла назначать квалифицированного разработчика в столбце `verified_by`. Вы могли бы реализовать эти ограничения при помощи триггера или ограничения `CHECK`. Эти ограничения существуют между столбцами, не содержащими ключей, поэтому они не удовлетворяют критерию DKNF-формы.

Шестая нормальная форма стремится устранить все зависимости от соединений. Как правило, она поддерживается для поддержки журнала изменений атрибутов. Например, столбец `Bugs.status` изменяется долгое время, и нам может понадобиться записать историю изменений в дочерней таблице, чтобы иметь данные о том, когда были произведены изменения, кто их произвел и другие возможные детали.

Можно предположить, что в столбце `Bugs`, чтобы поддерживать шестую нормальную форму, почти каждому столбцу, возможно, понадобится отдельная сопроводительная таблица истории. Это приводит к переизбытку таблиц. Шестая нормальная форма — это массовое убийство большинства приложений, однако некоторые технологии организации хранилищ данных используют ее[1].

[1] Например, в якорном моделировании (Anchor Modeling): **www.anchormodeling.com/**.

А.4. ЗДРАВЫЙ СМЫСЛ

Правила нормализации не являются непостижимыми или сложными. Они лишь являются техникой здравого смысла, помогающей сократить дублирование и улучшить согласованность данных.

Вы можете использовать этот краткий обзор реляций и нормальных форм как справочник, который поможет вам разработать базы данных в будущих проектах наилучшим образом.

МНЕНИЯ ЧИТАТЕЛЕЙ О КНИГЕ «ПРОГРАММИРОВАНИЕ БАЗ ДАННЫХ SQL»

Я — убежденный сторонник передовых методов и предпочитаю учиться на ошибках других людей. В этой книге собрана куча ошибок, и, что удивительно, в ней содержатся мои собственные. Жаль, что я не прочитал эту книгу раньше.

Маркус Адамс (Marcus Adams)
Ведущий разработчик программного обеспечения

Билл написал захватывающую, полезную, важную и уникальную книгу. Разработчики программного обеспечения, несомненно, извлекут пользу, прочитав о программировании баз данных SQL и решениях, описанных здесь. Я не преминул сразу же воспользоваться методами из этой книги и усовершенствовал свои приложения. Фантастический труд!

Фредерик Дауд (Frederic Daoud)
Автор книг «Stripes: ...And Java Web Development Is Fun Again» и «Getting Started with Apache Click»

«Программирование баз данных SQL» — книга, которую обязательно должны прочитать разработчики программного обеспечения, часто сталкивающиеся с вариантами разработки баз данных, представленными в данной книге. Она помогает уяснить последствия создания структур баз данных и принять оптимальные решения на основе требований, ожиданий, измерений и реальных условий.

Дарби Фэлтон (Darby Felton)
Соучредитель компании DevBots Software Development

Мне на самом деле понравился подход Билла к написанию этой книги. Он демонстрирует уникальный стиль и чувство юмора. Эти факторы действительно важны, когда обсуждаются потенциально малоинтересные темы. Билл преуспел в популяризации обучения среди разработчиков, используя отличный стиль описания и удобный макет книги, облегчающий ее применение в качестве справочника. Короче говоря, это отличный новый ресурс — кандидат на полку особо ценных практических пособий.

Арджен Ленц (Arjen Lentz)
директор-исполнитель веб-ресурса Open Query (**openquery.com**)
соавтор книги «High Performance MySQL», второе издание

Бесспорно, книга — продукт многолетней практической работы с базами данных SQL. Каждая тема раскрывается подробно, а внимание к деталям превзошло все мои ожидания. Хотя книга не для новичков, любой опытный разработчик, имеющий дело с SQL, найдет, что она представляет собой ценный ресурс и обязательно преподнесет что-нибудь новое.

Майк Набережный (Mike Naberezny)
Партнер по бизнесу в компании Maintainable Software;
соавтор книги «Rails for PHP Developers»

Это отличная книга для разработчиков программного обеспечения, изучивших основы SQL, которым необходимо проектировать SQL-базы данных для проектов, требующих более основательных знаний.

Лиз Нили (Liz Neely)
Ведущий программист баз данных

Книга Карвина насыщена хорошими практическими советами. Она издана вовремя. Хотя многих пользователей сегодня привлекают авангардные, даже причудливые решения, профессионалы, вооруженные этой книгой, получили возможность дальнейшего совершенствования в области SQL.

Мейк Шмидт (Maik Schmidt)
Автор книг «Enterprise Recipes with Ruby and Rails»
и «Enterprise Integration with Ruby»

Биллу удалось ухватить суть той массы ошибок, в которых, вероятно, так или иначе погрязли мы все, работая с SQL, даже не осознавая, что столкнулись с неприятностями. Решения Билла охватывают великое множество случаев: от традиционных «Не могу поверить, что это опять сделал я» до хитрых сценариев, где оптимальный вариант противоречит SQL-догмам, на которых выросли все профессионалы. Отличное чтиво и для консерваторов, и для новичков, и для всех, кто попадает в диапазон между первыми и вторыми.

Дэни Торп
Ведущий инженер корпорации Microsoft;
автор книги «Delphi Component Design»

Производственно-практическое издание

ПРОФЕССИОНАЛЬНЫЕ КОМПЬЮТЕРНЫЕ КНИГИ

Билл Карвин

**ПРОГРАММИРОВАНИЕ БАЗ ДАННЫХ SQL.
ТИПИЧНЫЕ ОШИБКИ И ИХ УСТРАНЕНИЕ**

Главный редактор *И. Федосова*
Зав. редакцией *А. Баранов*
Ведущий редактор *И. Липкин*
Художественный редактор *М. Левыкин*
Компьютерная верстка *С. Птицына*

ООО «Рид Групп»
119021, Москва, ул. Россолимо, д. 17, стр. 1; тел.: 788-0075(76)

Свои пожелания и предложения по качеству и содержанию книг
Вы можете сообщить по эл. адресу: editorial@readgroup.ru

Издание осуществлено при техническом содействии
ООО «Издательство АСТ»

Подписано в печать 27.10.2011. Формат 70x100/ 16.
Усл. печ. л. 27,3. Печать офсетная. Бумага писчая.
Тираж 2 000 экз. Заказ № 5454М.

**Отпечатано с готовых диапозитивов
в типографии ООО «Полиграфиздат»
144003, г. Электросталь, Московская область, ул. Тевосяна, д. 25**