Алан Бьюли

Изучаем SQL

Санкт-Петербург — Москва
2007

Alan Beaulieu

Learning SQL

O'REILLY®

Оглавление

Предисловие

Языки программирования постоянно появляются и исчезают, и очень немногие из современных языков имеют более чем 10-летнюю историю. Среди долгожителей можно назвать КОБОЛ, который до сих пор довольно широко используется в мэйнфреймовых средах, и C, по-прежнему весьма популярный при разработке операционных систем, серверов и встроенных систем. В области баз данных это SQL, корни которого уходят в далекие 1970-е.

SQL – язык для формирования, манипулирования и извлечения данных из реляционной БД. Одна из причин популярности реляционных БД в том, что, будучи правильно спроектированными, они могут оперировать гигантскими объемами данных. В работе с большими наборами данных SQL напоминает современную цифровую фотокамеру с мощным объективом: он позволяет просматривать большие объемы данных или перейти к «крупному плану», т. е. сфокусироваться на отдельных строках (подвластно и все, что между этими крайностями). Другие СУБД дают сбой при мощных нагрузках, потому что их фокус слишком узок (увеличительные линзы достигают своего максимума). Именно по этой причине все попытки низвергнуть реляционные БД и SQL оканчиваются неудачей. Поэтому, даже несмотря на то, что SQL – старый язык, похоже, его ждет еще очень долгая жизнь и блестящее будущее.

Зачем изучать SQL?

Если вы собираетесь работать с реляционными БД – писать приложения, или выполнять задачи по администрированию, или формировать отчеты, – вам понадобится знать, как взаимодействовать с данными БД. Даже при использовании инструмента, генерирующего SQL (например, инструмента создания отчетов), могут возникнуть ситуации, в которых понадобится обойти автоматические возможности и создавать собственные SQL-выражения.

Дополнительное преимущество изучения SQL в том, что вы быстрее рассмотрите и поймете структуры данных, применяемые для хранения информации о вашей организации. Почувствовав себя уверенно со своей БД, вы сможете вносить предложения по изменению или дополнению ее схемы.

Почему именно эта книга?

Язык SQL включает несколько категорий. Выражения, с помощью которых создаются объекты БД (таблицы, индексы, ограничения и т. д.), называют *SQL-выражениями управления схемой данных* (*schema statements*). Выражения, предназначенные для создания, манипулирования и извлечения данных, хранящихся в БД, называют *SQL-выражениями для работы с данными* (*data statements*). Если вы администратор, то будете использовать и те и другие SQL-выражения. Если вы программист или составитель отчетов, то сможете (или вам будет *позволено*) использовать только SQL-выражения для работы с данными. Хотя в этой книге встречается много SQL-выражений управления схемой, основное внимание в ней уделено возможностям программирования.

Поскольку команд немного, SQL-выражения для работы с данными кажутся простыми. По-моему, многие из имеющихся книг по SQL только усиливают это впечатление, давая лишь поверхностный обзор того, что можно делать с помощью этого языка. Однако если вы собираетесь работать с SQL, вам следует полностью понимать все его возможности и то, как сочетать их для получения мощных результатов. На мой взгляд, эта книга – единственная, где язык SQL описан подробно, и при этом она не является «кирпичом» (вам знакомы эти «полные руководства» по 1250 страниц, пылящиеся у народа на полках).

Хотя примеры из книги подходят для MySQL, Oracle Database и SQL Server, мне пришлось отобрать один из этих продуктов, чтобы разместить БД для выполнения примеров и форматировать результирующие наборы, возвращаемые примерами запросов. Из этих трех я выбрал MySQL, потому что он свободно доступен, его легко установить и просто администрировать. Читателей, использующих другой сервер, прошу скачать и установить MySQL и загрузить предлагаемую БД, чтобы иметь возможность выполнять примеры и экспериментировать с данными.

Структура книги

Книга содержит 13 глав и 4 приложения:

- В главе 1 «Немного истории» рассматривается история компьютерных БД, включая возникновение реляционной модели и языка SQL.
- В главе 2 «Создание и заполнение базы данных» показывается, как создавать БД MySQL и таблицы, используемые в примерах к книге, и как заполнять таблицы данными.
- Глава 3 «Азбука запросов» знакомит с выражением `select` и представляет наиболее распространенные блоки (clauses): `select`, `from`, `where`.

- Глава 4 «Фильтрация» представляет разные типы условий, которые могут использоваться в блоке `where` выражений `select`, `update` и `delete`.
- В главе 5 «Запрос к нескольким таблицам» показывается, как запросы могут работать с несколькими таблицами посредством соединений таблиц.
- Глава 6 «Работа с множествами» – все о множествах данных и о том, как они могут взаимодействовать внутри запросов.
- Глава 7 «Создание, преобразование и работа с данными» представляет несколько встроенных функций, используемых для манипулирования или преобразования данных.
- В главе 8 «Группировка и агрегаты» показывается, как можно агрегировать данные.
- Глава 9 «Подзапросы» представляет подзапрос (мой любимый прием) и показывает, как применяются подзапросы.
- Глава 10 «И снова соединения» продолжает рассматривать различные типы соединений таблиц.
- В главе 11 «Условная логика» рассматривается использование условной логики (т. е. `if-then-else`) в выражениях `select`, `insert`, `update` и `delete`.
- Глава 12 «Транзакции» знакомит с транзакциями и их использованием.
- В главе 13 «Индексы и ограничения» исследуются индексы и ограничения.
- Приложение A «ER-диаграмма примера базы данных» содержит схему базы данных, используемой для всех примеров книги.
- Приложение B «MySQL-расширения языка SQL» представляет некоторые интересные возможности реализации SQL – MySQL, не входящие в стандарт ANSI.
- Приложение C «Решения к упражнениям» содержит решения упражнений, приводимых в главах.
- Приложение D «Дополнительные источники» подсказывает, куда можно обратиться, чтобы получить более глубокие навыки.

Условные обозначения, используемые в книге

В книге используются следующие типографские обозначения:

Курсив

Используется для имен файлов, имен каталогов и URL-адресов. Также используется для выделения и при первом упоминании технического термина.

Моноширинный шрифт

Используется для примеров кода и обозначения ключевых слов SQL в тексте.

Моноширинный курсив

Используется для обозначения пользовательских терминов.

ВЕРХНИЙ РЕГИСТР

Используется для обозначения ключевых слов SQL в примерах кода.

Моноширинный полужирный шрифт

Выделяет ввод пользователя в примерах с интерактивным взаимодействием. Также выделяет элементы кода, на которые следует обратить особое внимание.

Так выделяются советы, рекомендации или общие примечания. Например, с помощью примечаний я обращаю ваше внимание на полезные новые возможности Oracle9*i*.

Обозначает предупреждение или предостережение. Так я предупреждаю, например, о том, что неаккуратное применение некоего блока SQL может иметь неожиданные последствия.

Контакты

Пожалуйста, присылайте комментарии и вопросы по данной книге издателю:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (в Соединенных Штатах или Канаде)
(707) 829-0515 (международный или местный)
(707) 829-0104 (факс)

Для этой книги издательство O'Reilly поддерживает веб-страницу, на которой приведены список опечаток, примеры и вся дополнительная информация. Эту страницу можно найти по адресу:

http://www.oreilly.com/catalog/learningsql

Чтобы прокомментировать или задать технические вопросы по этой книге, присылайте электронные сообщения по адресу:

bookquestions@oreilly.com

Более подробная информация о книгах издательства O'Reilly, конференциях, центрах ресурсов и портале O'Reilly Network представлена на веб-сайте:

http://www.oreilly.com

Использование примеров кода

Цель этой книги – помочь вам выполнить работу. Код, представленный в книге, в общем случае можно использовать в программах и документации. На воспроизведение небольших фрагментов кода в вашей программе разрешение не требуется. Для продажи или распространения CD-ROM с примерами из книг O'Reilly разрешение *необходимо*. Если, отвечая на вопросы, вы ссылаетесь на книгу и цитируете пример кода, разрешение не требуется. Для включения существенного объема кода примеров из этой книги в документацию собственного продукта разрешение *необходимо*.

Мы признательны за указание авторства, но не требуем этого. Обычно указание источника включает название, автора, издателя и ISBN. Например: «Learning SQL by Alan Beaulieu. Copyright 2005 O'Reilly Media, Inc., 0-596-00727-2.»

Если вы сомневаетесь в корректности использования вами примеров кода, обратитесь за разъяснениями по адресу *permissions@oreilly.com*.

Safari Enabled

Если на обложке книги есть пиктограмма «Safari® Enabled», это означает, что книга доступна в Сети через O'Reilly Network Safari Bookshelf.

Safari предлагает намного лучшее решение, чем электронные книги. Это виртуальная библиотека, позволяющая без труда находить тысячи лучших технических книг, вырезать и вставлять примеры кода, загружать главы и находить быстрые ответы, когда требуется наиболее верная и свежая информация. Она свободно доступна по адресу *http://safari.oreilly.com*.

Благодарности

Книга – живой организм, и то, что сейчас перед вами, далеко от моих первоначальных набросков. Эта метаморфоза произошла во многом благодаря моему редактору Джонатану Геннику (Jonathan Gennick). Спасибо тебе за помощь на каждом этапе проекта – как за твою редакторскую доблесть, так и за экспертную поддержку в вопросах, связанных с языком SQL. Еще я хотел бы поблагодарить трех моих технических рецензентов: Питера Гулутзана (Peter Gulutzan), Джозефа Молинаро (Joseph Molinaro) и Джеффа Кокса (Jeff Cox), побудивших меня сделать эту книгу и технически насыщенной, и подходящей для читателей, не знакомых с SQL. Также огромная благодарность многим сотрудникам O'Reilly Media, помогавшим воплотить эту книгу в реальность, в том числе корректору Мэтт Хатчинсон (Matt Hutchinson), дизайнеру обложки Элли Волкхаузен (Ellie Volckhausen) и художнику-оформителю Робу Романо (Rob Romano).

1

Немного истории

Прежде чем засучить рукава и приступить к работе, полезно представить некоторые базовые концепции и заглянуть в историю компьютеризированного хранения и извлечения данных.

Введение в базы данных

База данных – это всего лишь набор взаимосвязанных данных. Например, телефонный справочник – это база данных имен, телефонных номеров и адресов всех людей, проживающих в определенной местности. Будучи, несомненно, широко и часто используемой базой данных, телефонный справочник не лишен следующих недостатков:

- Поиск конкретного телефонного номера занимает много времени, особенно если в телефонном справочнике очень много записей.
- Телефонный справочник проиндексирован только по именам/фамилиям, поэтому для поиска абонента по определенному адресу такая база данных практически не используется, хотя теоретически это и возможно.
- С момента публикации телефонного справочника адекватность представленной в нем информации постепенно снижается, поскольку люди уезжают и приезжают из этой местности, меняют свои телефонные номера или переезжают по другому адресу в той же местности.

Недостатками, присущими телефонным справочникам, обладает любая другая некомпьютеризированная система хранения данных, например регистратура поликлиники, хранящая медицинские карты пациентов. Из-за громоздкости бумажных баз данных одними из первых компьютерных приложений были разработаны *системы баз данных* (*database systems*) – средства компьютеризированного хранения и извлечения данных. Поскольку система БД хранит информацию в элек-

тронном виде, а не на бумаге, она может быстрее извлекать данные, индексировать их разными способами и поставлять своим пользователям самую свежую информацию.

В первых системах БД информация хранилась на магнитных лентах. Количество лент во много раз превышало количество устройств считывания, поэтому техническим специалистам приходилось постоянно менять ленты, чтобы предоставить ту или иную запрашиваемую информацию. Поскольку объем памяти у компьютеров той эпохи был невелик, при многократных запросах одних и тех же данных обычно требовалось многократное считывание этих данных с магнитной ленты. Хотя эти системы БД и были существенным усовершенствованием по сравнению с бумажными БД, им было еще далеко до возможностей современных технологий. (Современные системы БД могут работать с терабайтами данных, распределенными по многим дискам с быстрым доступом, и держать в быстродействующей памяти десятки гигабайт этих данных; впрочем, я немного забегаю вперед.)

Нереляционные системы баз данных

Первые несколько десятилетий данные в компьютеризированных системах БД хранились и представлялись по-разному. Например, в *иерархической системе баз данных* (*hierarchical database system*) данные были представлены в виде одной или нескольких древовидных структур. На рис. 1.1 показано, как с помощью древовидных структур можно организовать данные банковских счетов Джорджа Блейка (George Blake) и Сью Смит (Sue Smith).

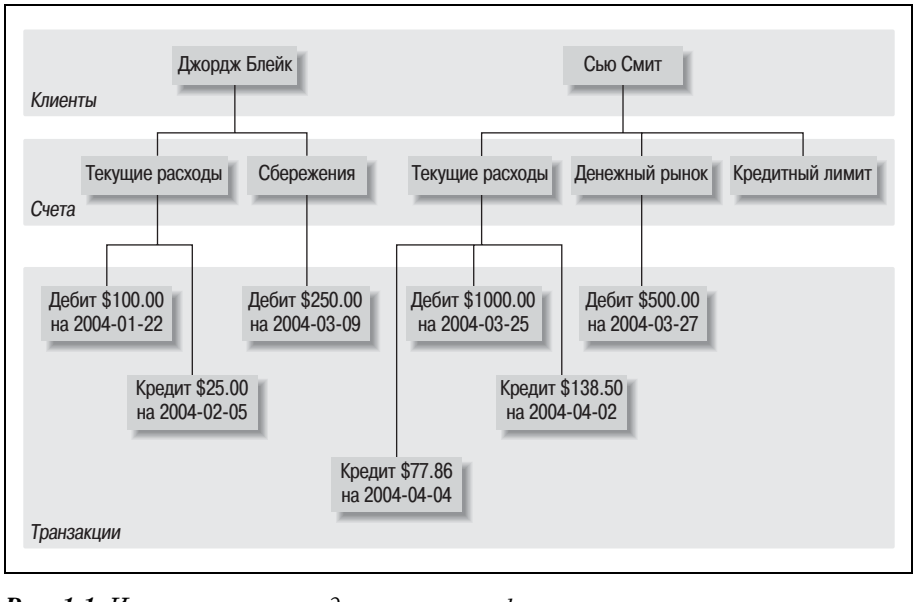


Рис. 1.1. *Иерархическое представление информации по счетам*

И у Джорджа, и у Сью есть собственное дерево, включающее их счета и транзакции, производимые по этим счетам. Иерархическая система базы данных предоставляет средства для нахождения дерева конкретного клиента и последующего обхода этого дерева в поисках нужных счетов и/или транзакций. У каждого узла дерева может быть ни одного или один родитель и ни одного, один или много дочерних узлов. Такую конфигурацию называют *иерархией с одним родителем* (*single-parent hierarchy*).

Другой распространенный подход, называемый *сетевой базой данных* (*network database system*), представляет собой наборы записей и наборы связей (links), определяющих отношения (relationships) между разными записями. На рис. 1.2 показано, как выглядели бы те же счета Джорджа и Сью в такой системе.

Чтобы найти транзакции, производимые по депозитному счету денежного рынка Сью, понадобилось бы сделать следующее:

1. Найти клиентскую запись Сью Смит.
2. Перейти по связи от клиентской записи Сью Смит к списку ее счетов.
3. Просматривать цепочку счетов до тех пор, пока не будет найден счет денежного рынка.

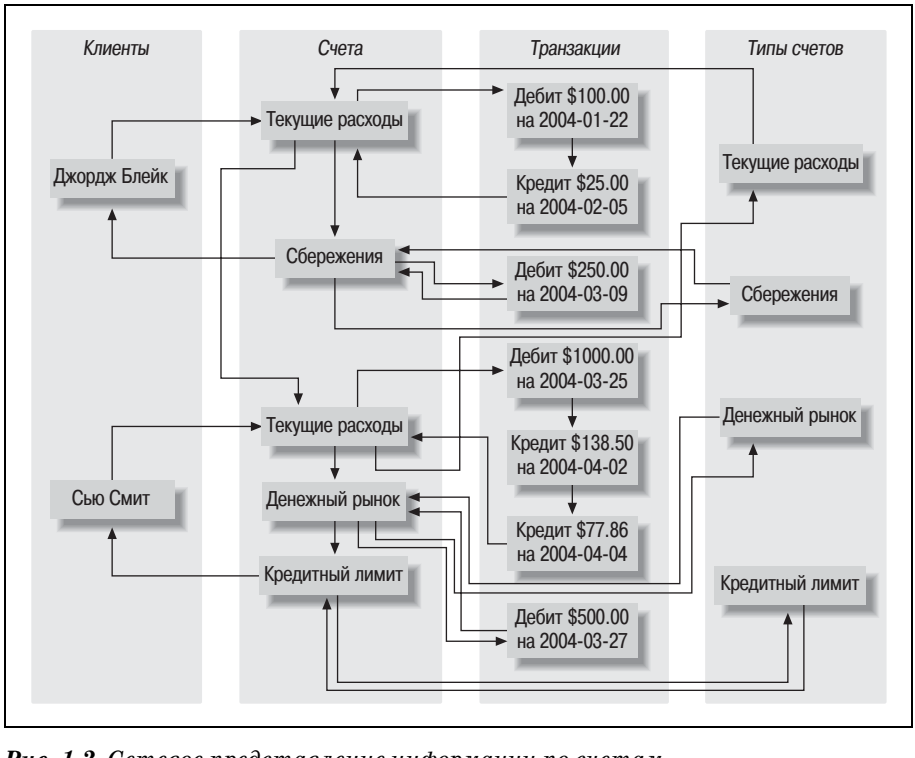


Рис. 1.2. *Сетевое представление информации по счетам*

4. Перейти по связи от записи денежного рынка к списку его транзакций.

Одну интересную особенность сетевых баз данных демонстрирует набор записей `product` (тип счета), на рис. 1.2 крайний справа. Обратите внимание, что каждая запись `product` (`Checking` (текущие расходы), `Savings` (сбережения) и т. д.) указывает на список записей `account` (счет), соответствующих этому типу счета. Поэтому доступ к записям `account` может быть осуществлен из нескольких мест (и через записи `customer`, и через записи `product`), что делает сетевую базу данных *иерархией с несколькими родителями* (*multiparent hierarchy*).

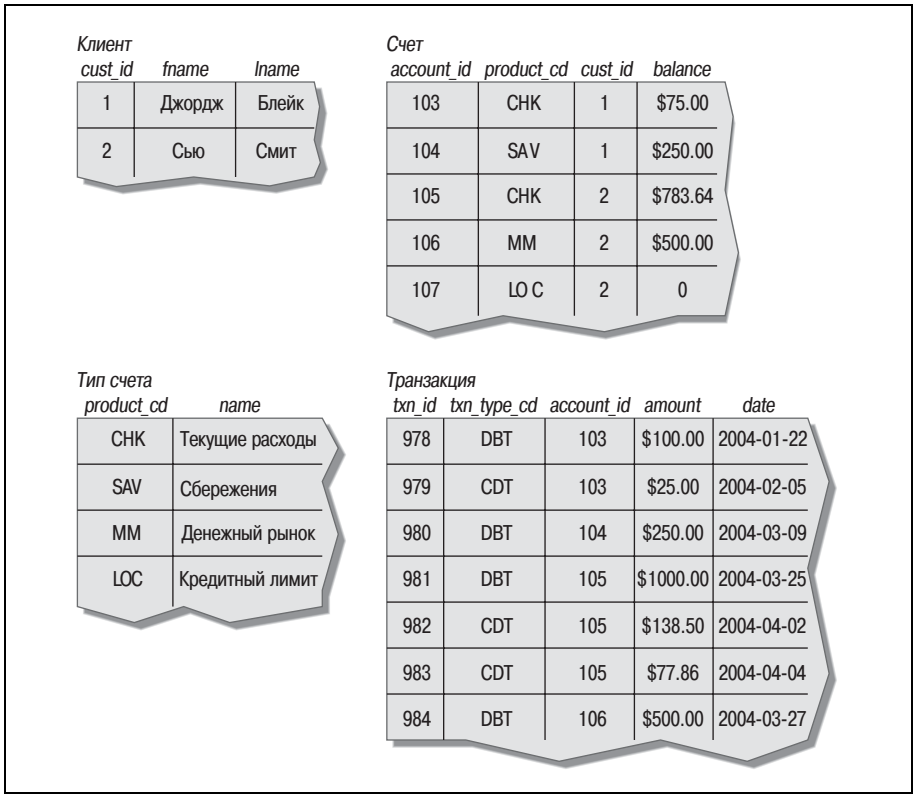
И иерархические, и сетевые системы баз данных ныне живы и здоровы, хотя преимущественно в мире мэйнфреймов. Кроме того, иерархические системы БД возродились в службах каталогов, таких как Active Directory компании Microsoft и Directory Server компании Netscape, а также с появлением XML (Extensible Markup Language, расширяемый язык разметки). Однако начиная с 1970-х годов все большую популярность приобретает новый способ представления данных, более строгий, но при этом более понятный и удобный.

Реляционная модель

В 1970 году сотрудник исследовательской лаборатории IBM доктор Е. Ф. Кодд (E. F. Codd) опубликовал статью под названием «A Relational Model of Data for Large Shared Data Banks» (Реляционная модель данных для больших банков данных коллективного пользования), в которой предложил представлять данные как наборы *таблиц*. Вместо указателей для навигации по взаимосвязанным сущностям используются избыточные данные, связывающие записи разных таблиц. На рис. 1.3 представлена информация счетов Джорджа и Сью в таком контексте.

На рис. 1.3 есть четыре таблицы, представляющие четыре обсуждаемые сущности: `customer`, `product`, `account` и `transaction` (транзакция). Посмотрев на таблицу `customer`, можно увидеть три *столбца*: `cust_id` (идентификационный номер клиента), `fname` (имя клиента) и `lname` (фамилия клиента). Ниже в таблице `customer` видим две *строки*: первая содержит данные Джорджа Блейка, вторая – данные Сью Смит. Максимально возможное количество столбцов в таблице отличается для разных серверов, но обычно это достаточно большое число, и с ним нет проблем (Microsoft SQL Server, например, допускает до 1024 столбцов в таблице). Число строк в таблице – это больше вопрос физических возможностей (т. е. определяется доступным дисковым пространством), чем ограничений серверов БД.

Каждая таблица реляционной базы данных включает информацию, уникально идентифицирующую строку этой таблицы (*первичный ключ* (*primary key*)), а также дополнительные данные, необходимые для полного описания сущности. Возвращаясь к таблице `customer`: в столбце `cust_id` каждому клиенту соответствует определенный номер. Напри-



Клиент

cust_id	fname	lname
1	Джордж	Блейк
2	Сью	Смит

Счет

account_id	product_cd	cust_id	balance
103	CHK	1	$75.00
104	SAV	1	$250.00
105	CHK	2	$783.64
106	MM	2	$500.00
107	LOC	2	0

Тип счета

product_cd	name
CHK	Текущие расходы
SAV	Сбережения
MM	Денежный рынок
LOC	Кредитный лимит

Транзакция

txn_id	txn_type_cd	account_id	amount	date
978	DBT	103	$100.00	2004-01-22
979	CDT	103	$25.00	2004-02-05
980	DBT	104	$250.00	2004-03-09
981	DBT	105	$1000.00	2004-03-25
982	CDT	105	$138.50	2004-04-02
983	CDT	105	$77.86	2004-04-04
984	DBT	106	$500.00	2004-03-27

Рис. 1.3. *Реляционное представление информации по счетам*

мер, Джорджа Блейка можно уникально идентифицировать с помощью клиентского идентификатора (ID №). Никогда никакому другому клиенту не будет присвоен такой же идентификатор, и этой информации достаточно, чтобы обнаружить данные Джорджа Блейка в таблице `customer`. Хотя в качестве первичного ключа можно было бы выбрать сочетание столбцов `fname` и `lname` (первичный ключ, состоящий из двух и более столбцов, называют *составным ключом* (*compound key*)), у двух и более человек, имеющих счета в банке, могут быть одинаковые имена и фамилии. Поэтому специально для первичных ключей в таблицу `customer` был включен столбец `cust_id`.

Некоторые из таблиц также содержат информацию, используемую для навигации к другой таблице. Например, в таблице `account` есть столбец `cust_id`, содержащий уникальный идентификатор клиента, открывшего счет, и столбец `product_cd`, содержащий уникальный идентификатор типа счета, которому будет соответствовать счет. Эти столбцы называют *внешними ключами* (*foreign keys*). Они служат той же цели, что и линии, соединяющие сущности в иерархической и сетевой версиях пред-

ставления информации по счетам. Однако, в отличие от жесткой структуры иерархической/сетевой моделей, реляционные таблицы можно использовать по-разному (даже так, как разработчики этой базы данных и не представляли себе).

Может показаться излишним хранить одни и те же данные в нескольких местах, но реляционная модель использует избыточность данных очень четко. Например, если таблица `account` включает столбец для уникального идентификатора клиента, открывшего счет, это правильно, а если включены также его имя и фамилия, то это неправильно. Например, если клиент изменяет имя, нужна уверенность, что его имя хранится только в одном месте базы данных. В противном случае данные могут быть изменены в одном месте, но не изменены в другом, что приведет к их недостоверности. Правильное решение – хранить эту информацию в таблице `customer`. В другие таблицы следует включить только `cust_id`. Также неправильно располагать в одном столбце несколько элементов данных, например в столбец `name` помещать имя и фамилию человека или в столбце `address` указывать улицу, город, страну и почтовый индекс. Процесс улучшения структуры базы данных с целью обеспечения хранения всех независимых элементов данных только в одном месте (за исключением внешних ключей) называют *нормализацией* (*normalization*).

Вернемся к четырем таблицам на рис. 1.3; на первый взгляд может быть непонятно, как использовать их для поиска транзакций Джорджа Блейка по его текущему счету. Во-первых, находим уникальный идентификатор Джорджа Блейка в таблице `customer`. Затем строку в таблице `account`, столбец `cust_id` которой содержит уникальный идентификатор Джорджа, а столбец `product_cd` соответствует строке таблицы `product`, столбец `name` которой содержит значение `"Checking"`. Наконец, в таблице `transaction` находим строки, столбец `account_id` которых соответствует уникальному идентификатору из таблицы `account`. Возможно, все это кажется сложным, но с помощью языка SQL может быть осуществлено одной-единственной командой, как вы вскоре увидите.

Немного терминологии

В предыдущих разделах были введены некоторые новые термины, поэтому приведем кое-какие формальные определения. В табл. 1.1 приведены термины, используемые в данной книге, и их определения.

Таблица 1.1. Термины и определения

Термин	Определение
Сущность (entity)	То, что представляет интерес для пользователей базы данных, например клиенты, запчасти, географическое положение и т. д.
Столбец (column)	Отдельный элемент данных, хранящийся в таблице.

Термин	Определение
Строка (row)	Набор столбцов, которые вместе полностью описывают сущность или некоторое действие, производимое над сущностью. Также называется *записью* (*record*).
Таблица (table)	Набор строк, хранящийся в памяти (непостоянная таблица) или на постоянном запоминающем устройстве (постоянная таблица).
Результирующий набор (result set)	Другое название непостоянной таблицы, обычно являющейся результатом SQL-запроса.
Первичный ключ (primary key)	Один или более столбцов, которые можно использовать как уникальный идентификатор для каждой строки таблицы.
Внешний ключ (foreign key)	Один или более столбцов, которые можно совместно использовать для идентификации одной строки другой таблицы.

Что такое SQL?

Помимо определения реляционной модели Кодд предложил язык для работы с данными в реляционных таблицах, названный DSL/Alpha. Вскоре после публикации статьи Кодда в IBM была организована группа для создания прототипа языка на базе его идей. Эта группа разработала упрощенную версию DSL/Alpha, которую назвали SQUARE. В результате усовершенствования SQUARE появился язык SEQUEL, который в конце концов получил имя SQL.

Сейчас SQL разменял четвертый десяток, претерпев за свой век множество изменений. В середине 1980-х Национальный институт стандартизации США (American National Standards Institute, ANSI) начал разрабатывать первый стандарт языка SQL, который был опубликован в 1986 г. Дальнейшие доработки были отражены в следующих версиях стандарта SQL (1989, 1992, 1999 и 2003 гг.). Наряду с усовершенствованием базового языка в SQL появились и новые возможности для обеспечения объектно-ориентированной функциональности.

SQL идет рука об руку с реляционной моделью, потому что результатом SQL-запроса является таблица (в данном контексте также называемая *результирующим набором*). Таким образом, в реляционной базе данных можно создать новую постоянную таблицу, просто сохранив результирующий набор запроса. Аналогично в качестве входных данных запрос может использовать как постоянные таблицы, так и результирующие наборы других запросов (подробно это будет рассмотрено в главе 9).

И последнее замечание: SQL не акроним (хотя многие настаивают, что это сокращение от Structured Query Language (Структурированный язык запросов)). Название этого языка произносится по буквам (т. е. «S», «Q», «L») или как «sequel» (сиквел).

Классы SQL-выражений

Язык SQL разбит на несколько отдельных частей. В данной книге будут рассмотрены: SQL-выражения управления схемой данных (SQL schema statements), предназначенные для определения структур данных, хранящихся в базе данных; SQL-выражения для работы с данными (SQL data statements), предназначенные для работы со структурами данных, ранее определенными с помощью SQL-выражений управления схемой; SQL-выражения управления транзакциями, предназначенные для начала, завершения и отката транзакций (рассматриваются в главе 12). Например, новая таблица базы данных создается с помощью SQL-выражения управления схемой `create table` (создать таблицу), а чтобы заполнить ее данными, потребуется SQL-выражение для работы с данными `insert` (вставить).

Чтобы дать представление об этих выражениях, приведем SQL-выражение управления схемой, создающее таблицу `corporation` (корпорация):

```
CREATE TABLE corporation
 (corp_id SMALLINT,
  name VARCHAR(30),
  CONSTRAINT pk_corporation PRIMARY KEY (corp_id)
 );
```

Это выражение создает таблицу с двумя столбцами, `corp_id` и `name`, где столбец `corp_id` определен как первичный ключ таблицы. Подробная информация о данном выражении, например доступные в MySQL типы данных, приводится в следующей главе. Теперь рассмотрим SQL-выражение для работы с данными, которое вставляет в таблицу `corporation` запись для корпорации Acme Paper Corporation:

```
INSERT INTO corporation (corp_id, name)
VALUES (27, 'Acme Paper Corporation');
```

Это выражение добавляет в таблицу `corporation` строку со значением `27` в столбце `corp_id` и значением `Acme Paper Corporation` в столбце `name`.

Наконец, приведем простое выражение `select` (выбрать) для извлечения только что созданных данных:

```
mysql> SELECT name
    -> FROM corporation
    -> WHERE corp_id = 27;
+------------------------+
| name                   |
+------------------------+
| Acme Paper Corporation |
+------------------------+
```

Все элементы БД, созданные посредством SQL-выражений управления схемой, хранятся в специальном наборе таблиц, который называется *словарем данных* (*data dictionary*). Все эти «данные о базе данных» на-

зывают *метаданными* (*metadata*). К таблицам словаря данных можно делать запросы с помощью оператора `select`, в точности как к созданным вами таблицам. Таким образом, текущие структуры данных, развернутые в БД во время выполнения, становятся доступными. Например, если требуется создать отчет о новых счетах, открытых за последний месяц, можно жестко закодировать известные на момент написания отчета имена столбцов таблицы `account` либо сделать запрос к словарю данных, получить текущий набор столбцов и динамически генерировать отчет при каждом выполнении.

Данная книга посвящена главным образом той части языка для работы с данными, к которой относятся команды `select`, `update` (обновить), `insert` и `delete` (удалить). SQL-выражения управления схемой рассмотрены в главе 2, где создается БД, используемая в примерах данной книги. Вообще говоря, SQL-выражения управления схемой не требуют особого внимания, за исключением их синтаксиса, тогда как у SQL-выражений для работы с данными (хотя их и немного) есть масса нюансов, нуждающихся в подробном изучении. Поэтому большинство глав данной книги посвящены SQL-выражениям для работы с данными.

SQL: непроцедурный язык

Если в прошлом вам приходилось работать с языками программирования, вы привыкли к описанию переменных и структур данных, использованию условной логики (if-then-else), циклическим конструкциям (do while ... end) и разделению кода на небольшие многократно используемые части (объекты, функции, процедуры). Код передается компилятору, и результирующий исполняемый код делает в точности (ну, не всегда *в точности*) то, что вы запрограммировали. С каким бы языком программирования ни работали, Java, C#, C, Visual Basic или любым другим *процедурным* языком, вы полностью управляете действиями программы. С SQL, однако, понадобится отказаться от привычного контроля над выполнением, потому что SQL-выражения определяют необходимые входные и выходные данные, а способ выполнения выражения зависит от компонента механизма СУБД (database engine), называемого *оптимизатором* (*optimizer*). Работа оптимизатора заключается в том, чтобы рассмотреть SQL-выражение и с учетом конфигурации таблиц и доступных индексов принять решение о самом эффективном пути выполнения запроса (ну, не всегда *самом* эффективном). Большинство СУБД позволяют программисту влиять на решения оптимизатора с помощью *подсказок оптимизатору* (*optimizer hints*), например предложений по использованию конкретного индекса. Однако большинство пользователей SQL никогда не доберется до этого уровня сложности и будет оставлять подобные тонкости администраторам БД или специалистам по вопросам производительности.

Следовательно, с SQL писать полные приложения не получится. Если требуется создать что-то сложнее простого сценария для работы с оп-

ределенными данными, понадобится интегрировать SQL со своим любимым языком программирования. Некоторые производители баз данных сделали это за вас, например Oracle с языком PL/SQL или Microsoft с TransactSQL. Благодаря этим языкам SQL-выражения для работы с данными являются частью грамматики языка программирования, что позволяет свободно интегрировать запросы к БД с процедурными командами. Однако при использовании не характерного для БД языка, такого как Java, для выполнения SQL-выражений понадобится специальное средство. Некоторые из этих программных средств предоставляются производителями БД, тогда как другие создаются сторонними производителями или разработчиками ПО с открытым исходным кодом. В табл. 1.2 показаны некоторые доступные варианты интегрирования SQL в конкретные языки программирования.

Таблица 1.2. Средства интегрирования SQL

Язык программирования	Программное средство
Java	JDBC (Java Database Connectivity) (JavaSoft)
C++	RogueWave SourcePro DB (инструмент сторонних производителей для соединения с БД Oracle, SQL Server, MySQL, Informix, DB2, Sybase и PostgreSQL)
C/C++	Pro*C (Oracle) MySQL C API (с открытым исходным кодом) DB2 Call Level Interface (IBM)
C#	ADO.NET (Microsoft)
VisualBasic	ADO.NET (Microsoft)

Если требуется только интерактивное выполнение SQL-команд, каждый производитель БД обеспечивает как минимум простой инструмент передачи SQL-команд механизму СУБД и просмотра результатов. Большинство производителей предлагает также графический инструмент, в одном окне которого вводятся SQL-команды, а в другом выводятся результаты их выполнения. Поскольку примеры данной книги работают с базой данных MySQL, для запуска примеров и форматирования результатов я буду использовать утилиту командной строки *mysql*.

Примеры SQL

Ранее в этой главе я обещал показать SQL-выражение, возвращающее все транзакции текущего счета Джорджа Блейка. Не будем тянуть, вот оно:

```
SELECT t.txn_id, t.txn_type_cd, t.date, t.amount
FROM customer c INNER JOIN account a ON c.cust_id = a.cust_id
  INNER JOIN product p ON p.product_cd = a.product_cd
  INNER JOIN transaction t ON t.account_id = a.account_id
```

```
WHERE c.fname = 'George' and c.lname = 'Blake'
  AND p.name = 'checking';
```

Без лишних на этом этапе подробностей: данный запрос идентифицирует в таблице account строку Джорджа Блейка, а в таблице product – строку с типом счета 'checking' (текущие расходы), в таблице account находит строку, соответствующую данной комбинации «клиент/тип счета», и возвращает четыре столбца таблицы transaction для всех транзакций по этому счету. Все концепции, присутствующие в данном запросе (и многие другие), будут рассмотрены в следующих главах; здесь мне просто хотелось показать, как выглядел бы запрос.

Предыдущий запрос содержит три разных *блока* (*clauses*): select, from и where. Практически каждый сформированный вами запрос будет включать, по крайней мере, эти три блока, хотя есть и другие блоки, применяемые для более сложных целей. Роль каждого из этих трех блоков можно продемонстрировать следующим образом:

```
SELECT /* одна или более сущностей */ ...
FROM   /* одно или более мест */ ...
WHERE  /* удовлетворяется одно или более условий */ ...
```

Большинство реализаций SQL воспринимают текст, расположенный между тегами /* и */, как комментарии.

Обычно первая задача при создании запроса – определить, какая таблица или таблицы понадобятся, а затем добавить их в блок from. Далее необходимо отсеять данные этих таблиц, которые не помогут ответить на запрос. Для этого в блок where вводятся условия. Наконец, принимается решение о том, какие столбцы разных таблиц требуется извлечь, и они добавляются в блок select. Вот простой пример поиска всех клиентов по фамилии Smith (Смит):

```
SELECT cust_id, fname
FROM customer
WHERE lname = 'Smith'
```

Этот запрос выполняет поиск в таблице customer всех строк, столбец lname которых соответствует строке 'Smith', и возвращает столбцы cust_id и fname этих строк.

Кроме создания запросов к БД вам, скорее всего, придется заполнять и изменять данные БД. Вот простой пример добавления новой строки в таблицу product:

```
INSERT INTO product (product_cd, name)
VALUES ('CD', 'Certificate of Depasit')
```

Ой, кажется, в слове «Deposit» ошибка! Никаких проблем. Это можно исправить с помощью выражения update:

```
UPDATE product
```

```
SET name = 'Certificate of Deposit'
WHERE product_cd = 'CD';
```

Обратите внимание, что в выражении `update` тоже есть блок `where`, как и в выражении `select`, потому что `update` должно отобрать строки, подлежащие изменению. В данном случае задано, что должны быть изменены только те строки, столбцы `product_cd` которых соответствуют строке `'CD'`. Поскольку столбец `product_cd` является первичным ключом таблицы `product`, следует ожидать, что выражение `update` изменит только одну строку (или ни одной, если такого значения в таблице нет). При выполнении любого SQL-выражения для работы с данными механизм СУБД выводит отчет с указанием того, сколько строк было подвержено его воздействию. Если используется интерактивный инструмент, например уже упомянутый инструмент командной строки *mysql*, будет получено сообщение о том, сколько строк было:

- возвращено выражением `select`;
- создано выражением `insert`;
- изменено выражением `update`;
- удалено выражением `delete`.

Если используется процедурный язык с одним из уже упомянутых программных средств, то после выполнения SQL-выражения для работы с данными это средство включит вызов функции запроса этой информации. В общем, не мешает проверять эти данные, чтобы убедиться, что выражение не сделало ничего непредвиденного (например, если забыть включить в выражение `delete` блок `where`, будут удалены все строки таблицы!).

Что такое MySQL?

Реляционные базы данных продаются уже более двух десятилетий. К самым зрелым и популярным продуктам относятся:

- Oracle Database от Oracle Corporation
- SQL Server от Microsoft
- DB2 Universal Database от IBM
- Sybase Adaptive Server от Sybase
- Informix Dynamic Server от IBM

Все эти серверы БД делают примерно одно и то же, хотя некоторые лучше оснащены для работы с очень большими или высокопроизводительными БД. Другие лучше ведут себя при работе с объектами, или очень большими файлами, или XML-документами и т. д. Кроме того, очень хорошо, что все эти серверы совместимы с последним стандартом ANSI SQL. Это положительный момент, и я обязательно покажу, как писать SQL-выражения, которые будут выполняться на любой из этих платформ (с небольшими изменениями или вообще без них).

Наряду с этим последние пять лет в сообществе сторонников открытого исходного кода наблюдалась активная деятельность по созданию жизнеспособной альтернативы коммерческим серверам БД. Два наиболее распространенных сервера БД с открытым исходным кодом – PostgreSQL и MySQL. Веб-сайт MySQL (*http://www.mysql.com*) в настоящее время заявляет о более чем 6 000 000 установок, их сервер доступен бесплатно, и я убедился, что скачать и установить его чрезвычайно просто. Поэтому я решил, что все примеры для данной книги будут выполняться на БД MySQL (версии 4.1.11). Для форматирования результатов запросов будет использоваться инструмент командной строки *mysql*. Даже если вы уже работаете с другим сервером и вообще не планируете использовать MySQL, я рекомендую установить последнюю версию сервера MySQL, загрузить схему и данные примера и экспериментировать с примерами этой книги.

Однако помните, что:

> Эта книга не о реализации SQL в MySQL.

Скорее, данная книга создана, чтобы обучить читателя создавать SQL-выражения, которые будут выполняться на MySQL и последних версиях Oracle Database, Sybase Adaptive Server и SQL Server с небольшими изменениями или вообще без них. Возможно, при использовании одного из упомянутых серверов IBM хлопот у вас будет чуть больше.

Чтобы по возможности сохранить код из данной книги платформонезависимым, я воздержусь от демонстрации некоторых интересных вещей, реализованных в языке SQL для MySQL и не осуществимых в других реализациях БД. Но для читателей, планирующих продолжать работу с MySQL, некоторые из этих возможностей рассмотрены в приложении B.

Дополнительные источники

Общая цель следующих четырех глав – представить SQL-выражения для работы с данными, уделив при этом особое внимание трем основным блокам выражения `select`. Кроме того, приводится множество примеров, использующих банковскую схему (она представлена в следующей главе и задействована во всех примерах данной книги). Надеюсь, что постоянное использование одной и той же БД позволит читателю вникать в суть примера, не тратя время на изучение применяемых таблиц.

Твердо усвоив основы, с помощью оставшихся глав вы изучите дополнительные концепции, по большей части не зависимые друг от друга. Поэтому, столкнувшись с какими-либо трудностями, всегда можно двинуться дальше, а позже перечитать главу. Прочитав книгу и проработав все примеры, вы уверенно пойдете к вершинам мастерства SQL.

Вот несколько заслуживающих внимания источников для читателей, желающих узнать больше о реляционных БД, истории компьютеризированных систем управления БД или языке SQL:

- К. Дж. Дейт (C. J. Date) «Database in Depth: Relational Theory for Practitioners», O'Reilly.
- К. Дж. Дейт «An Introduction to Database Systems, Eighth Edition», Addison Wesley.[1]
- К. Дж. Дейт «The Database Relational Model: A Retrospective Review and Analysis: A Historical Account and Assessment of E. F. Codd's Contribution to the Field of Database Technology», Addison Wesley.
- *http://en.wikipedia.org/wiki/Database_management_system*
- *http://www.mcjones.org/System_R/*

[1] К. Дейт «Введение в системы баз данных», 8-е издание, Вильямс, 2005.

2

Создание и заполнение базы данных

В этой главе представлена информация, необходимая для создания вашей первой БД, таблиц и ассоциированных данных, используемых в примерах книги. Также рассказывается о различных типах данных и об их применении при создании таблиц. Поскольку примеры книги выполняются на СУБД MySQL, здесь наблюдается небольшое смещение акцентов представляемого материала в сторону возможностей и синтаксиса MySQL, но большинство концепций применимы к любому серверу.

Создание базы данных MySQL

Если в вашем распоряжении уже есть СУБД MySQL, можно выполнять приведенные ниже инструкции, начиная с п. 8. Но не забывайте, что эта книга ориентирована на MySQL версии 4.1.11 или более поздних, поэтому если вы используете более раннюю версию, скорее всего, не помешает обновить ее или установить другой сервер.

Следующие инструкции отражают минимальный набор действий, необходимых для установки сервера MySQL на компьютере, работающем под управлением Windows, создания базы данных и загрузки тестовых данных для этой книги:

1. Скачайте MySQL Database Server (версии 4.1.11 или более поздней) с *http://dev.mysql.com*. Если сервер планируется использовать только для обучения, скачайте Essentials Package (Основной пакет), включающий только широко используемые инструменты, а не Complete Package (Полный пакет).
2. Двойным щелчком по загруженному файлу запустите процесс установки.

3. Установите сервер, используя вариант «typical install» (обычная установка). Установка должна пройти быстро и безболезненно, но не стесняйтесь обращаться к онлайновому руководству по установке (*http://dev.mysql.com/doc/mysql/en/Installing.html*).
4. По завершении установки, перед тем как нажать кнопку завершения, убедитесь, что флажок Configure the MySQL Server now (Конфигурировать сервер MySQL сейчас) установлен. Это нужно, чтобы запустился Configuration Wizard (Мастер конфигурации).
5. При запуске Configuration Wizard выберите переключатель Standard Configuration (Стандартная конфигурация) и затем установите флажки Install as Windows Service (Установить как службу Windows) и Include Bin Directory in Windows Path (Включить каталог Bin в путь поиска Windows).
6. Во время конфигурирования вам будет предложено выбрать пароль для привилегированного пользователя root. Не забудьте записать пароль, он понадобится позже.
7. Откройте консоль (с помощью Start→Run→Command (Пуск→Выполнить→Command)) и из консоли зарегистрируйтесь как привилегированный пользователь с помощью команды `mysql -u root -p`. Вам будет предложено ввести пароль, после этого появится подсказка `mysql>`.
8. Создайте нового пользователя базы данных. Я создал пользователя lrngsql с помощью команды `grant all privileges on *.* to 'lrngsql'@'localhost' identified by 'xxxxx';` (замените *xxxxx* паролем, который выбрали для этого пользователя).
9. Завершите сеанс с помощью команды `quit;` (выйти) и зарегистрируйтесь из консоли как новый пользователь посредством команды `mysql -u lrngsql -p`.
10. Создайте базу данных. Я создал БД «bank» (банк) с помощью выражения `create database bank;`.
11. Выберите новую БД с помощью выражения `use bank;`.
12. Скачайте тестовые данные для этой книги. Файл можно найти на сайте *learningsql* в разделе *Examples* (примеры) для данной книги.
13. Из инструмента командной строки *mysql* с помощью команды `source` (источник) загрузите данные из закачанного файла, например `source c:\tmp\learning_sql.sql`. Вместо пути `c:\tmp\` укажите каталог, в котором находится сценарий с тестовыми данными.

Теперь у вас должна быть рабочая БД, заполненная всеми данными, необходимыми для примеров данной книги.

Инструмент командной строки mysql

При вызове инструмента командной строки *mysql* можно задать имя пользователя и используемую БД:

```
mysql -u lrngsql -p bank
```

Будет запрошен ваш пароль, и затем появится приглашение `mysql>`, с помощью которого вы сможете создавать SQL-выражения и просматривать результаты их выполнения. Например, чтобы узнать текущие дату и время, можно выполнить следующий запрос:

```
mysql> SELECT now( );
+---------------------+
| now( )              |
+---------------------+
| 2005-05-06 16:48:46 |
+---------------------+
1 row in set (0.01 sec)
```

Функция `now( )` – это встроенная функция MySQL, возвращающая текущие дату и время. Как видите, инструмент командной строки *mysql* форматирует результаты запросов, помещая их в прямоугольник, очерченный символами `+`, `-` и `|`. Выведя все результаты (в данном случае это всего одна строка), инструмент командной строки *mysql* покажет количество возвращенных строк и длительность выполнения выражения SQL.

Завершив работу с инструментом командной строки *mysql*, для возвращения в консоль просто введите `quit;` или `exit;`.

О пропущенном блоке from

При работе с некоторыми серверами БД нельзя создать запрос без блока `from` (из), в котором должна быть указана по крайней мере одна таблица. Oracle Database – именно такой сервер. Для тех случаев, когда требуется только вызвать функцию, Oracle предоставляет таблицу `dual` (двойственная), состоящую всего из одного столбца `dummy` (макет), который содержит всего одну строку данных. Для обеспечения совместимости с Oracle Database MySQL тоже предоставляет таблицу `dual`. Следовательно, предыдущий запрос текущих даты и времени можно было бы написать так:

```
mysql> SELECT now( )
       FROM dual;
+---------------------+
| now( )              |
+---------------------+
| 2005-05-06 16:48:46 |
+---------------------+
1 row in set (0.01 sec)
```

Если вы не работаете с Oracle и вам не нужна совместимость с этой СУБД, таблицу `dual` можно полностью игнорировать.

Типы данных MySQL

Вообще говоря, все популярные серверы БД обладают способностью хранить одни и те же типы данных, такие как строки, даты и числа. Обычно их различие заключается в возможности хранения специальных типов данных, например XML-документов, или очень больших текстов, или двоичных документов. Поскольку данная книга является введением в SQL и 98 % всех столбцов, которые вы когда-либо встретите, будут простыми типами данных, мы рассмотрим только символьные, числовые и временные типы данных.

Символьные данные

Символьные данные могут храниться как строки фиксированной или переменной длины. Разница заключается в том, что строки фиксированной длины справа дополняются пробелами, тогда как строки переменной длины – нет. При определении столбца символьного типа необходимо задать максимальный размер сохраняемой в нем строки. Например, если предполагается хранить строки длиной до 20 символов, можно использовать любое из этих описаний:

```
CHAR(20)     /* строка фиксированной длины */
VARCHAR(20)  /* строка переменной длины */
```

В настоящее время максимальная длина этого типа данных составляет 255 символов (хотя в будущих версиях будут допустимы более длинные строки). Для сохранения более длинных строк (таких как сообщения электронной почты, XML-документы и т. д.) используйте один из текстовых типов – `tinytext` (крошечный текст), `text` (текст), `mediumtext` (средний текст), `longtext` (длинный текст)), – рассматриваемых в данном разделе позже. В общем, тип `char` подходит для случая, когда в столбце предполагается хранить только строки одинаковой длины, например сокращенные названия государств, а тип `varchar` – для строк разной длины. Типы `char` и `varchar` одинаково применимы во всех основных серверах БД.

Когда речь идет о применении типа данных `varchar`, СУБД Oracle Database является исключением. Пользователи Oracle при описании символьных столбцов переменной длины должны применять тип `varchar2`.

Наборы символов

В языках, использующих латинский алфавит, например в английском, довольно мало символов, то есть каждый символ хранится как один байт. В других языках, таких как японский и корейский, много символов. Таким образом, в них для хранения одного символа требуется несколько байт. Поэтому такие наборы символов называют *многобайтовыми наборами символов* (*multibyte character sets*).

MySQL может хранить данные, используя разные наборы символов, как одно-, так и многобайтовые. Просмотреть поддерживаемые сервером наборы символов можно с помощью команды show (показать):

```
mysql> SHOW CHARACTER SET;
+----------+-----------------------------+---------------------+--------+
| Charset  | Description                 | Default collation   | Maxlen |
+----------+-----------------------------+---------------------+--------+
| big5     | Big5 Traditional Chinese    | big5_chinese_ci     |      2 |
| dec8     | DEC West European           | dec8_swedish_ci     |      1 |
| cp850    | DOS West European           | cp850_general_ci    |      1 |
| hp8      | HP West European            | hp8_english_ci      |      1 |
| koi8r    | KOI8-R Relcom Russian       | koi8r_general_ci    |      1 |
| latin1   | ISO 8859-1 West European    | latin1_swedish_ci   |      1 |
| latin2   | ISO 8859-2 Central European | latin2_general_ci   |      1 |
| swe7     | 7bit Swedish                | swe7_swedish_ci     |      1 |
| ascii    | US ASCII                    | ascii_general_ci    |      1 |
| ujis     | EUC-JP Japanese             | ujis_japanese_ci    |      3 |
| sjis     | Shift-JIS Japanese          | sjis_japanese_ci    |      2 |
| hebrew   | ISO 8859-8 Hebrew           | hebrew_general_ci   |      1 |
| tis620   | TIS620 Thai                 | tis620_thai_ci      |      1 |
| euckr    | EUC-KR Korean               | euckr_korean_ci     |      2 |
| koi8u    | KOI8-U Ukrainian            | koi8u_general_ci    |      1 |
| gb2312   | GB2312 Simplified Chinese   | gb2312_chinese_ci   |      2 |
| greek    | ISO 8859-7 Greek            | greek_general_ci    |      1 |
| cp1250   | Windows Central European    | cp1250_general_ci   |      1 |
| gbk      | GBK Simplified Chinese      | gbk_chinese_ci      |      2 |
| latin5   | ISO 8859-9 Turkish          | latin5_turkish_ci   |      1 |
| armscii8 | ARMSCII-8 Armenian          | armscii8_general_ci |      1 |
| utf8     | UTF-8 Unicode               | utf8_general_ci     |      3 |
| ucs2     | UCS-2 Unicode               | ucs2_general_ci     |      2 |
| cp866    | DOS Russian                 | cp866_general_ci    |      1 |
| keybcs2  | DOS Kamenicky Czech-Slovak  | keybcs2_general_ci  |      1 |
| macce    | Mac Central European        | macce_general_ci    |      1 |
| macroman | Mac West European           | macroman_general_ci |      1 |
| cp852    | DOS Central European        | cp852_general_ci    |      1 |
| latin7   | ISO 8859-13 Baltic          | latin7_general_ci   |      1 |
| cp1251   | Windows Cyrillic            | cp1251_general_ci   |      1 |
| cp1256   | Windows Arabic              | cp1256_general_ci   |      1 |
| cp1257   | Windows Baltic              | cp1257_general_ci   |      1 |
| binary   | Binary pseudo charset       | binary              |      1 |
| geostd8  | GEOSTD8 Georgian            | geostd8_general_ci  |      1 |
```

Когда я установил сервер MySQL, в качестве набора символов, применяемого по умолчанию, был автоматически выбран latin1. Однако для каждого символьного столбца в БД можно выбрать отдельный набор символов. Можно даже хранить разные наборы символов в одной таблице. Чтобы при описании столбца выбрать набор символов, отличный от применяемого по умолчанию, надо просто указать один из поддерживаемых наборов символов после описания типа:

```
VARCHAR(20) CHARACTER SET utf8
```

С MySQL можно также задавать набор символов по умолчанию для всей базы данных:

```
CREATE DATABASE foreign_sales CHARACTER SET utf8;
```

Пожалуй, для ознакомительной книги о наборах символов сказано достаточно, но еще есть тема локализации, которая гораздо шире. Если вы планируете работать с несколькими наборами символов или использовать незнакомые наборы символов, обратитесь к таким книгам, как «Java Internationalization» (O'Reilly) или «Unicode Demystified: A Practical Programmer's Guide to the Encoding Standard» (Addison Wesley).

Текстовые данные

Если нужно хранить данные, для которых не хватит 255 символов столбца типа `char` или `varchar`, вам понадобится один из текстовых типов.

В табл. 2.1 показаны доступные текстовые типы и их максимальные размеры.

Таблица 2.1. Текстовые типы данных MySQL

Тип	Максимальное число символов
`Tinytext`	255
`Text`	65 535
`Mediumtext`	16 777 215
`Longtext`	4 294 967 295

Выбирая тот или иной текстовый тип, необходимо помнить следующее:

- Если размер данных, загружаемых в текстовый столбец, превышает максимальный размер для этого типа, не поместившиеся данные отсекаются.
- В отличие от столбца типа `varchar`, при загрузке данных в такой столбец пробелы в конце строки не удаляются.
- При использовании столбцов типа `text` для сортировки или группировки используются только первые 1024 байта, хотя при необходимости это ограничивающее значение можно увеличить.
- Разные текстовые типы присущи исключительно MySQL. У SQL Server для больших символьных данных есть только один тип `text`, а в DB2 и Oracle применяется тип данных под названием `clob` (Character Large Object, большой символьный объект).

При создании столбца для данных произвольного формата, например столбца `notes` (примечания) для хранения информации о взаимодействиях клиента с отделом клиентского сервиса вашей компании, которую вам не хотелось бы ограничивать 255 символами, следует выбрать тип `text` или `mediumtext`.

В Oracle Database допускаются столбцы `char` до 2000 байт и `varchar` до 4000 байт. SQL Server может оперировать данными типа `char` и `varchar` размером до 8000 байт. Поэтому при работе с Oracle или SQL Server потребность в текстовых типах данных меньше, чем при работе с MySQL. Однако начиная с версии 5.0.3 (в настоящее время она проходит бета-тестирование) MySQL обскачет оба эти сервера, поскольку максимальный размер столбцов типа `char` и `varchar` в нем достигнет 65 535 байт.

Числовые данные

Хотя и кажется, что хватило бы одного числового типа данных с названием «numeric» (числовой), все же есть разные числовые типы, отражающие разные способы использования чисел, как показано ниже:

Столбец, являющийся индикатором поставки заказа покупателю

Столбец такого типа, называемого *Boolean* (*булев*), может содержать `0`, что означает `false` (ложь) и `1`, что означает `true` (истина).

Первичный ключ для таблицы транзакций, генерируемый системой

Обычно начинается с `1` и увеличивается с шагом 1, возможно, до очень больших значений.

Номер позиции в клиентской электронной корзине для покупок

Значениями столбца данного типа являются положительные целые числа от 1 до (максимум) 200 (для фанатов шопинга).

Данные позиционирования сверлильного станка для печатных плат

Высокоточные научные или технологические данные часто требуют точности до восьми десятичных знаков.

MySQL располагает несколькими разными числовыми типами для работы с этими (и многими другими) видами информации. Наиболее часто числовые типы используют для хранения целых чисел. При задании одного из таких типов можно также указать, что данные *беззнаковые*, тогда сервер будет знать, что все хранящиеся в столбце данные неотрицательные. В табл. 2.2 показано пять разных типов данных, предназначенных для хранения целых чисел.

Таблица 2.2. Целые типы данных MySQL

Тип	Диапазон значений со знаком	Диапазон значений без знака
`Tinyint`	от -128 до 127	от 0 до 255
`Smallint`	от -32 768 до 32 767	от 0 до 65 535
`Mediumint`	от -8 388 608 до 8 388 607	от 0 до 16 777 215
`Int`	от -2 147 483 648 до 2 147 483 647	от 0 до 4 294 967 295
`Bigint`	от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807	от 0 до 18 446 744 073 709 551 615

При создании столбца одного из целых типов MySQL выделит для хранения данных соответствующее количество памяти – от 1 байта для типа `tinyint` до 8 байт для `bigint`. Поэтому попытайтесь подобрать тип достаточного размера для хранения самого большого из предполагаемых чисел без неоправданного расхода памяти.

Для чисел с плавающей точкой (таких как 3,1415927) можно выбрать один из типов, приведенных в табл. 2.3.

Таблица 2.3. Типы данных MySQL для чисел с плавающей точкой

Тип	Числовой диапазон
`Float(p,s)`	от -3,402823466E+38 до -1,175494351E-38 и от 1,175494351E-38 до 3,402823466E+38
`Double(p,s)`	от 1,7976931348623157E+308 до -2,2250738585072014E-308 и от 2,2250738585072014E-308 до 1,7976931348623157E+308

Для типа с плавающей точкой можно задать *точность* (*precision*) (общее допустимое число разрядов, как справа, так и слева от десятичной точки) и *масштаб* (*scale*) (допустимое число разрядов справа от десятичной точки), но эти параметры не являются обязательными. В табл. 2.3 они представлены как *p* и *s*. Задавая точность и масштаб для столбца, имеющего тип с плавающей точкой, необходимо помнить, что сохраняемые в нем данные будут округляться, если число разрядов в них превысит заданный масштаб и/или точность. Например, столбец, определенный как `float(4,2)`, будет сохранять всего четыре разряда, два слева и два справа от десятичной точки. Поэтому с такими числами, как 27,44 и 8,19, будет все в порядке, а вот число 17,8675 будет округлено до 17,87, а число 178,5 будет округлено (грубо) до 99,99 – самое большое число, которое может быть сохранено в этом столбце.

Как и данные целого типа, данные с плавающей точкой могут быть определены как `unsigned` (беззнаковые), но это только предотвращает хранение в столбце отрицательных чисел, но не изменяет диапазон данных, которые могут быть сохранены в столбце.

Временные данные

Наряду со строками и числами довольно часто приходится работать с информацией о датах и/или времени. Этот тип данных называют *временным* (*temporal*). К примерам временных данных в базе данных относятся:

- Дата будущего события, например доставки заказа покупателю
- Фактическая дата доставки заказа покупателю
- Дата и время изменения пользователем определенной строки таблицы
- Дата рождения сотрудника
- Год, соответствующий строке таблицы `yearly_sales` (продажи за год) в хранилище данных

- Время, необходимое для монтажа электропроводки в автомобиле на сборочном конвейере

В MySQL есть типы данных для обработки всех подобных ситуаций. В табл. 2.4 показаны временные типы данных, поддерживаемые MySQL.

Таблица 2.4. Временные типы данных MySQL

Тип	Формат по умолчанию	Допустимые значения
Date	YYYY-MM-DD	от 1000-01-01 до 9999-12-31
Datetime	YYYY-MM-DD HH:MI:SS	от 1000-01-01 00:00:00 до 9999-12-31 23:59:59
Timestamp	YYYY-MM-DD HH:MI:SS	от 1970-01-01 00:00:00 до 2037-12-31 23:59:59
Year	YYYY	от 1901 до 2155
Time	HHH:MI:SS	от -838:59:59 до 838:59:59

Серверы БД хранят временные данные по-разному, и назначение форматирующей строки (второй столбец табл. 2.4) – показать, как будут представлены данные при извлечении, а также то, как должна быть сформирована строка даты при вставке или обновлении столбца временного типа. Таким образом, если бы вам понадобилось вставить дату 23 марта 2005 года в столбец date (дата) с форматом по умолчанию YYYY-MM-DD, то вы бы использовали строку '2005-03-23'. Построение и отображение временных данных подробно рассмотрено в главе 7.

На всех серверах БД допустимы различные диапазоны дат для столбцов временного типа. Oracle Database допускает даты от 4712 г. до н. э. до 9999 г. н.э., тогда как SQL Server обрабатывает только даты от 1753 г. н. э. до 9999 г. н. э. Хотя для большинства систем, отслеживающих текущие и будущие события, большой разницы здесь нет, об этом важно помнить при хранении исторических дат.

Различные компоненты форматов даты, приведенных в табл. 2.4, описаны в табл. 2.5.

Таблица 2.5. Компоненты формата даты

Компонент	Описание	Диапазон
YYYY	Год, включая столетие	от 1000 до 9999
MM	Месяц	от 01 (январь) до 12 (декабрь)
DD	День	от 01 до 31
HH	Час	от 01 до 24
HHH	Часы (прошедшие)	от -838 до 838
MI	Минута	от 01 до 60
SS	Секунда	от 01 до 60

Вот как были бы использованы различные временные типы при реализации приведенных выше примеров:

- Для хранения предполагаемой даты доставки заказа покупателю и даты рождения сотрудника использовались бы столбцы типа `date`, поскольку знать точное время рождения человека необязательно, а спланировать будущую доставку с точностью до секунды нереально.
- Для хранения информации о фактической доставке заказа покупателю использовался бы тип `datetime` (дата и время), поскольку важно отследить не только дату, но и точное время доставки.
- Столбец, отслеживающий время последнего изменения пользователем определенной строки таблицы, использовал бы тип `timestamp` (временная метка). Этот тип содержит ту же информацию, что и тип `datetime` (год, месяц, день, час, минуту, секунду), но при добавлении или изменении строки таблицы сервер MySQL автоматически заполнит столбец `timestamp` текущими значениями даты/времени.
- Столбец для хранения только данных о годе использовал бы тип `year` (год).
- Столбцы, содержащие данные о временном интервале, необходимом для выполнения задачи, использовали бы тип `time` (время). Этому типу данных не нужно хранить компонент даты – это сбивало бы с толку, поскольку интерес представляет только количество часов/минут/секунд, необходимое для выполнения задания. Эту информацию можно было бы получить, найдя разность значений из двух столбцов типа `datetime` (первый хранит дату/время начала выполнения задания, а второй – дату/время его завершения). Но проще использовать один столбец `time`.

В главе 7 будет рассказано, как работать с каждым из этих временных типов данных.

Создание таблиц

Теперь, имея четкое представление о том, какие типы данных могут храниться в базе данных MySQL, самое время взглянуть, как эти типы используются при описании таблиц. Начнем с описания таблицы для хранения информации о человеке.

Шаг 1: проектирование

Хорошо начать проектирование таблицы с небольшого мозгового штурма – это позволит определить информацию, которую было бы полезно включить. Немного поразмыслив о данных, описывающих человека, я получил вот что:

- Имя, фамилия (name)
- Пол (gender)
- Дата рождения (birth date)
- Адрес (address)

- Любимое блюдо (favorite foods)

Разумеется, список не полный, но этого пока достаточно. Следующий шаг – дать столбцам имена и назначить типы данных. В табл. 2.6 показан первый вариант.

Таблица 2.6. Таблица Person (человек), первое приближение

Столбец	Тип	Допустимые значения
Name	Varchar(40)	
Gender	Char(1)	M, F
Birth_date	Date	
Address	Varchar(100)	
Favorite_foods	Varchar(200)	

Столбцы name, address и favorite_foods типа varchar позволяют записывать информацию в свободной форме. В столбце gender (пол) допускается только один символ, M (М) или F (Ж). Столбцу birth_date (дата рождения) назначен тип date, поскольку точное время не требуется.

Шаг 2: уточнение

В главе 1 была представлена концепция *нормализации*, что является процессом обеспечения отсутствия в БД дублирующихся (кроме внешних ключей) или составных столбцов. При повторном анализе столбцов таблицы возникают следующие соображения:

- Столбец name на самом деле является составным объектом, включающим имя и фамилию.
- Поскольку несколько человек могут иметь одинаковые имя, пол, дату рождения и т. д., в таблице person нет столбцов, гарантирующих уникальность.
- Столбец address – тоже составной объект, включающий улицу, город, штат/область, страну и почтовый индекс.
- Столбец favorite_foods – это список, содержащий 0, 1 или более независимых элементов. Было бы лучше вынести эти данные в отдельную таблицу, включающую внешний ключ к таблице person, чтобы обозначить человека, к которому приписано конкретное блюдо.

В табл. 2.7 можно увидеть нормализованный вариант таблицы person после учета всех этих замечаний.

Теперь, когда у таблицы person есть первичный ключ (person_id), гарантирующий уникальность, следующим шагом будет построение таблицы favorite_food, включающей внешний ключ к таблице person. Результат показан в табл. 2.8.

Столбцы person_id и food (блюдо) образуют первичный ключ таблицы favorite_food. Столбец person_id также является внешним ключом к таблице person.

Таблица 2.7. Таблица Person, второе приближение

Столбец	Тип	Допустимые значения
Person_id	Smallint (unsigned)	
First_name	Varchar(20)	
Last_name	Varchar(20)	
Gender	Char(1)	M, F
Birth_date	Date	
Street	Varchar(30)	
City	Varchar(20)	
State	Varchar(20)	
Country	Varchar(20)	
Postal_code	Varchar(20)	

Таблица 2.8. Таблица Favorite_food (любимое блюдо)

Столбец	Тип
Person_id	Smallint (unsigned)
Food	Varchar(20)

Шаг 3: построение SQL-выражений управления схемой данных

Теперь, по завершении проектирования двух таблиц для размещения персональной информации, следующим шагом является формирование SQL-выражений для создания таблиц в БД. Вот выражение для создания таблицы person:

```
CREATE TABLE person
 (person_id SMALLINT UNSIGNED,
  fname VARCHAR(20),
  lname VARCHAR(20),
  gender CHAR(1),
  birth_date DATE,
  address VARCHAR(30),
  city VARCHAR(20),
  state VARCHAR(20),
  country VARCHAR(20),
  postal_code VARCHAR(20),
  CONSTRAINT pk_person PRIMARY KEY (person_id)
 );
```

В этом выражении должно быть понятно все, кроме последнего элемента. При описании таблицы необходимо сообщить серверу БД, какой столбец или столбцы будут играть роль первичного ключа табли-

цы. Осуществляется это путем создания *ограничения* (*constraint*) для таблицы. В описание таблицы можно добавить ограничение одного из нескольких типов. Данное ограничение является *ограничением первичного ключа* (*primary-key constraint*). Оно накладывается на столбец `person_id` и получает имя `pk_person`. Обычно я начинаю имена ограничений первичного ключа с приставки `pk_`, а затем указываю имя таблицы, чтобы при просмотре списка таких ограничений было ясно, чем каждое из них является.

Говоря об ограничении, упомянем еще один тип, который мог бы быть полезным для таблицы `person`. В табл. 2.7 был добавлен третий столбец для допустимых значений определенных столбцов (например `'M'` и `'F'` для столбца `gender`). Это другой тип ограничения – *проверочное ограничение* (*check constraint*), ограничивающее допустимые значения конкретного столбца. MySQL позволяет вводить в описание столбца проверочное ограничение:

```
gender CHAR(1) CHECK (gender IN ('M','F')),
```

На большинстве серверов БД проверочные ограничения работают соответствующим образом, а сервер MySQL допускает описание проверочных ограничений, но не выполняет их проверку. Но MySQL предоставляет другой символьный тип данных – `enum` (перечисление), который вводит проверочное ограничение в описание типа. Вот как это выглядело бы для описания столбца `gender`:

```
gender ENUM('M','F'),
```

Вот как выглядит создание таблицы `person` с введением типов данных `enum` для столбца `gender`:

```
CREATE TABLE person
 (person_id SMALLINT UNSIGNED,
  fname VARCHAR(20),
  lname VARCHAR(20),
  gender ENUM('M','F'),
  birth_date DATE,
  address VARCHAR(30),
  city VARCHAR(20),
  state VARCHAR(20),
  country VARCHAR(20),
  postal_code VARCHAR(20),
  CONSTRAINT pk_person PRIMARY KEY (person_id)
 );
```

Позже в данной главе будет показано, что происходит при попытке добавить в столбец данные, не соответствующие проверочному ограничению (или, в случае MySQL, значениям перечисления).

Теперь все готово для выполнения выражения `create table` с помощью инструмента командной строки *mysql*. Вот как это выглядит:

```
mysql> CREATE TABLE person
    ->  (person_id SMALLINT UNSIGNED,
```

```
    ->   fname VARCHAR(20),
    ->   lname VARCHAR(20),
    ->   gender ENUM('M','F'),
    ->   birth_date DATE,
    ->   address VARCHAR(30),
    ->   city VARCHAR(20),
    ->   state VARCHAR(20),
    ->   country VARCHAR(20),
    ->   postal_code VARCHAR(20),
    ->   CONSTRAINT pk_person PRIMARY KEY (person_id)
    ->  );
Query OK, 0 rows affected (0.27 sec)
```

После обработки выражения `create table` сервер MySQL возвращает сообщение «Query OK, 0 rows affected» (Запрос выполнен без ошибок, 0 строк подверглось обработке), что говорит об отсутствии синтаксических ошибок в выражении. Если требуется убедиться, что таблица `person` действительно существует, можно использовать команду `describe` (описать) (или `desc` для краткости) и посмотреть описание таблицы:

```
mysql> DESC person;
+-------------+----------------------+------+-----+---------+-------+
| Field       | Type                 | Null | Key | Default | Extra |
+-------------+----------------------+------+-----+---------+-------+
| person_id   | smallint(5) unsigned |      | PRI | 0       |       |
| fname       | varchar(20)          | YES  |     | NULL    |       |
| lname       | varchar(20)          | YES  |     | NULL    |       |
| gender      | enum('M','F')        | YES  |     | NULL    |       |
| birth_date  | date                 | YES  |     | NULL    |       |
| address     | varchar(30)          | YES  |     | NULL    |       |
| city        | varchar(20)          | YES  |     | NULL    |       |
| state       | varchar(20)          | YES  |     | NULL    |       |
| country     | varchar(20)          | YES  |     | NULL    |       |
| postal_code | varchar(20)          | YES  |     | NULL    |       |
+-------------+----------------------+------+-----+---------+-------+
10 rows in set (0.06 sec)
```

Что такое Null?

В некоторых случаях невозможно предоставить значение определенного столбца таблицы. Например, при добавлении данных о новом заказе покупателя значение столбца `ship_date` (дата доставки) еще не может быть определено. В этом случае говорят, что столбец является *нулевым* (*null*) (обратите внимание, я не сказал, *равен* нулю), что указывает на отсутствие значения.

При проектировании таблицы можно определить, какие столбцы могут быть нулевыми (по умолчанию), а какие – нет (это обозначается путем добавления ключевых слов `not null` (ненулевой) после описания типа).

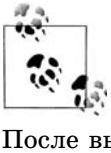
Смысл столбцов 1 и 2 результата выполнения выражения describe очевиден. Столбец 3 показывает, можно ли пропустить тот или иной столбец при вводе данных в таблицу. Я намеренно пока не включил эту тему в обсуждение (краткие рассуждения по этому вопросу можно найти во врезке «Что такое Null?»), она будет полностью рассмотрена в главе 4. Четвертый столбец показывает, участвует ли столбец в формировании какого-либо ключа (первичного или внешнего). В данном случае столбец person_id отмечен как первичный ключ. Столбец 5 показывает, будет ли определенный столбец заполнен значением по умолчанию в случае, если он пропущен при вводе данных в таблицу. Для столбца person_id значением по умолчанию является 0, хотя оно будет использовано только один раз, поскольку каждая строка таблицы person должна содержать в данном столбце уникальное значение (это первичный ключ). Шестой столбец (названный Extra (дополнительно)) содержит любую другую информацию, относящуюся к столбцу.

Теперь, после создания таблицы person, следующий шаг – создать таблицу favorite_food:

```
mysql> CREATE TABLE favorite_food
    ->  (person_id SMALLINT UNSIGNED,
    ->   food VARCHAR(20),
    ->   CONSTRAINT pk_favorite_food PRIMARY KEY (person_id, food),
    ->   CONSTRAINT fk_person_id FOREIGN KEY (person_id)
    ->     REFERENCES person (person_id)
    ->  );
Query OK, 0 rows affected (0.10 sec)
```

Это очень похоже на выражение create table для таблицы person, за несколькими исключениями:

- Поскольку у человека может быть несколько любимых блюд (что и стало причиной создания данной таблицы), одного столбца person_id для обеспечения уникальности в таблице недостаточно. Поэтому первичный ключ данной таблицы состоит из двух столбцов: person_id и food.
- Таблица favorite_food содержит другой тип ограничения – *ограничение внешнего ключа* (*foreign-key constraint*). Оно ограничивает значения столбца person_id таблицы favorite_food, позволяя ему включать только те значения, которые есть в таблице person. При таком ограничении не получится включить в таблицу favorite_food строку, показывающую, что person_id 27 любит пиццу, если в таблице person нет строки со значением 27 для person_id.



Если при создании таблицы ограничение внешнего ключа не было указано, его можно добавить позже с помощью оператора alter table (изменить таблицу).

После выполнения выражения create table по команде describe будет выведено следующее:

```
mysql> DESC favorite_food;
+--------------+----------------------+------+-----+---------+-------+
| Field        | Type                 | Null | Key | Default | Extra |
+--------------+----------------------+------+-----+---------+-------+
| person_id    | smallint(5) unsigned |      | PRI | 0       |       |
| food         | varchar(20)          |      | PRI |         |       |
+--------------+----------------------+------+-----+---------+-------+
```

Теперь, когда есть таблица, следующим логичным шагом будет добавить в нее кое-какие данные.

Заполнение и изменение таблиц

Имея таблицы `person` и `favorite_food`, можно приступить к изучению четырех SQL-выражений для работы с данными: `insert`, `update`, `delete` и `select`.

Вставка данных

Поскольку пока что в наших таблицах `person` и `favorite_food` нет данных, из четырех SQL-выражений для работы с данными первым рассмотрим `insert`. В выражении `insert` три основных компонента:

- Имя таблицы, в которую должны быть добавлены данные.
- Имена тех столбцов таблицы, которые должны быть заполнены.
- Значения, которыми должны быть заполнены столбцы.

Таким образом, не обязательно предоставлять данные для всех столбцов таблицы (если только все столбцы таблицы не были определены как `not null`). В некоторых случаях столбцы, не включенные в исходное выражение `insert`, будут заполнены позже с помощью выражений `update`. Бывает, что столбец в какой-то строке вообще никогда не заполняется данными (например, если заказ покупателя отменяется до поставки, столбец `ship_date` остается незаполненным).

Формирование числовых ключей

Прежде чем заполнить таблицу `person` данными, полезно обсудить процесс формирования значений числовых первичных ключей. Кроме выбора числа «от фонаря» есть два варианта:

- Найти в таблице самое большое на данный момент значение первичного ключа и прибавить 1.
- Позволить серверу БД предоставить значение.

Хотя первый вариант и кажется допустимым, он становится проблематичным в многопользовательской среде, поскольку два пользователя могут одновременно работать с таблицей и сгенерировать одно и то же значение первичного ключа. Напротив, все серверы БД, присутствующие сегодня на рынке, обеспечивают более надежный, более устойчи-

вый к ошибкам метод формирования числовых ключей. Иногда, например в Oracle Database, используется отдельный объект схемы (называемый *последовательностью* (*sequence*)). Однако в случае с MySQL надо просто включить для столбца первичного ключа свойство *auto-increment* (*автоприращение*). Обычно это делается при создании таблицы, но мы занимаемся этим сейчас, чтобы изучить еще одно SQL-выражение управления схемой, которое меняет описание существующей таблицы:

```
ALTER TABLE person MODIFY person_id SMALLINT UNSIGNED AUTO_INCREMENT;
```

Это выражение, по существу, переопределяет столбец `person_id` таблицы `person`. Теперь команда `describe` для этой таблицы отобразит атрибут автоприращения в столбце `Extra` для `person_id`:

```
mysql> DESC person;
+------------+----------------------+------+-----+---------+----------------+
| Field      | Type                 | Null | Key | Default | Extra          |
+------------+----------------------+------+-----+---------+----------------+
| person_id  | smallint(5) unsigned |      | PRI | NULL    | auto_increment |
| .          |                      |      |     |         |                |
| .          |                      |      |     |         |                |
| .          |                      |      |     |         |                |
```

При вводе данных в таблицу `person` просто задайте значение `null` для столбца `person_id`, и MySQL заполнит столбец следующим доступным числом (для столбцов с автоприращением MySQL по умолчанию начинает отсчет с 1).

Выражение insert

Теперь, когда все расставлено по местам, пора добавить кое-какие данные. Следующее выражение создает в таблице `person` строку для Вильяма Тернера (William Turner):

```
mysql> INSERT INTO person
    ->  (person_id, fname, lname, gender, birth_date)
    -> VALUES (null, 'William','Turner', 'M', '1972-05-27');
Query OK, 1 row affected (0.01 sec)
```

Обратная связь («Query OK, 1 row affected») сообщает, что синтаксис выражения правильный и что в базу данных была добавлена одна строка (поскольку это было выражение `insert`). С помощью выражения `select` можно увидеть только что добавленные в таблицу данные:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.06 sec)
```

Как видите, сервер MySQL генерирует для первичного ключа значение 1. Поскольку в таблице person всего одна строка, я не стал указывать, какая именно строка меня интересует, и попросту извлек все строки таблицы. Если бы строк было несколько, можно было бы добавить блок where и указать, что требуется извлечь данные для строки, значение person_id которой равно единице:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person
    -> WHERE person_id = 1;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.00 sec)
```

В данном запросе задается конкретное значение первичного ключа. Но для поиска строк может использоваться любой столбец таблицы, о чем свидетельствует следующий запрос, выбирающий все строки, столбцы lname которых содержат значение 'Turner':

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person
    -> WHERE lname = 'Turner';
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
+-----------+---------+--------+------------+
1 row in set (0.00 sec)
```

Прежде чем двигаться дальше, сделаю несколько замечаний относительно предыдущего выражения insert:

- Не заданы значения ни для одного столбца адреса. Это нормально, поскольку для них допускается значение null.
- Для столбца birth_date было предоставлено строковое значение. Поскольку строка соответствует формату, приведенному в табл. 2.4, MySQL преобразует ее для вас в дату.
- Количество и типы столбцов и предоставляемых значений должны совпадать. Если указывается семь столбцов и предоставляется только шесть значений, или если предоставленные значения не могут быть преобразованы в соответствующий тип данных для соответствующего столбца, вы получите ошибку.

Вильям также предоставил информацию о своих любимых блюдах. Вот три выражения вставки, позволяющих записать его кулинарные предпочтения:

```
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'pizza');
Query OK, 1 row affected (0.01 sec)
```

```
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'cookies');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (1, 'nachos');
Query OK, 1 row affected (0.01 sec)
```

А вот запрос, извлекающий любимые блюда Вильяма в алфавитном порядке с помощью блока `order by` (упорядочить по):

```
mysql> SELECT food
    -> FROM favorite_food
    -> WHERE person_id = 1
    -> ORDER BY food;
+---------+
| food    |
+---------+
| cookies |
| nachos  |
| pizza   |
+---------+
3 rows in set (0.02 sec)
```

Блок `order by` указывает серверу, как сортировать данные, возвращаемые запросом. Без `order by` данные таблицы будут извлечены в произвольном порядке.

Чтобы Вильям не скучал, можно выполнить еще одно выражение `insert` и добавить в таблицу `person` Сьюзен Смит (Susan Smith):

```
mysql> INSERT INTO person
    ->  (person_id, fname, lname, gender, birth_date,
    ->   address, city, state, country, postal_code)
    -> VALUES (null, 'Susan','Smith', 'F', '1975-11-02',
    ->   '23 Maple St.', 'Arlington', 'VA', 'USA', '20220');
Query OK, 1 row affected (0.01 sec)
```

Если снова запросить таблицу, мы увидим, что строке Сьюзен в качестве первичного ключа было присвоено значение `2`:

```
mysql> SELECT person_id, fname, lname, birth_date
    -> FROM person;
+-----------+---------+--------+------------+
| person_id | fname   | lname  | birth_date |
+-----------+---------+--------+------------+
|         1 | William | Turner | 1972-05-27 |
|         2 | Susan   | Smith  | 1975-11-02 |
+-----------+---------+--------+------------+
2 rows in set (0.00 sec)
```

Обновление данных

При первичном вводе информации в таблицу о Вильяме Тернере в выражение `insert` не были включены данные для различных столбцов ад-

реса. Следующее выражение показывает, как заполнить эти столбцы с помощью выражения update:

```
mysql> UPDATE person
    -> SET address = '1225 Tremont St.',
    ->   city = 'Boston',
    ->   state = 'MA',
    ->   country = 'USA',
    ->   postal_code = '02138'
    -> WHERE person_id = 1;
Query OK, 1 row affected (0.04 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Сервер ответил двустрочным сообщением: фраза «Rows matched: 1» (Подходящих строк: 1) говорит о том, что условия блока where соответствуют только одной строке таблицы, а «Changed: 1» (Изменено: 1) означает, что была изменена одна строка таблицы. Поскольку в блоке where задан первичный ключ строки Вильяма, именно так и должно было произойти.

Как видите, одним выражением update можно изменять несколько столбцов. Одним выражением также можно изменять несколько строк в зависимости от условий блока where. Рассмотрим, к примеру, что произошло бы, если бы блок where выглядел следующим образом:

```
WHERE person_id < 10
```

Поскольку значение person_id и у Вильяма, и у Сьюзен меньше 10, изменениям подвергнуться обе строки. Если опустить блок where совсем, выражение update обновит все строки таблицы.

Удаление данных

Похоже, Вильям и Сьюзен не вполне ладят друг с другом, поэтому один из них должен уйти. Поскольку Вильям был первым, Сьюзен будет вежливо «выставлена» выражением delete:

```
mysql> DELETE FROM person
    -> WHERE person_id = 2;
Query OK, 1 row affected (0.01 sec)
```

Опять же для выделения интересующей строки используется первичный ключ, поэтому из таблицы удаляется всего одна строка. Как и в случае выражения update, можно удалить и несколько строк. Все зависит от условий, заданных в блоке where. Если блок where опущен, будут удалены все строки.

Когда портятся хорошие выражения

До сих пор все SQL-выражения для работы с данными, приведенные в этой главе, были правильными и играли по правилам. Однако, исхо-

дя из описаний таблиц person и favorite_food, у вас есть много возможностей наделать ошибок при вставке или изменении данных. В этом разделе приведены некоторые из распространенных ошибок и показано, как сервер MySQL будет на них реагировать.

Неуникальный первичный ключ

Поскольку описания таблиц включают создание ограничений первичного ключа, MySQL проверит, чтобы в таблицы не вводились дублирующие значения. Следующее выражение делает попытку обойти свойство автоприращения столбца person_id и создать в таблице person еще одну строку со значением person_id, равным 1:

```
mysql> INSERT INTO person
    ->  (person_id, fname, lname, gender, birth_date)
    -> VALUES (1, 'Charles','Fulton', 'M', '1968-01-15');
ERROR 1062 (23000): Duplicate entry '1' for key 1
```

Ничто не мешает (по крайней мере, в текущей схеме) создать две строки с идентичными именами, адресами, датами рождения и т. д., если в столбце person_id у них разные значения.

Несуществующий внешний ключ

Описание таблицы favorite_food включает создание ограничения внешнего ключа для столбца person_id. Это ограничение гарантирует, что все значения person_id, введенные в таблицу favorite_food, имеются в таблице person. Вот что произошло бы при попытке создания строки, нарушающей это ограничение:

```
mysql> INSERT INTO favorite_food (person_id, food)
    -> VALUES (999, 'lasagna');
ERROR 1216 (23000): Cannot add or update a child row:
  a foreign key constraint fails
```

В этом случае таблица favorite_food считается *дочерней* (*child*), а таблица person – *родителем* (*parent*), поскольку таблица favorite_food зависит от данных таблицы person. Если требуется ввести данные в обе таблицы, сначала следует создать строку в parent, а затем уже можно будет ввести данные в favorite_food.

Ограничения внешнего ключа выполняются, только если таблицы создаются с использованием механизма хранения InnoDB. Механизмы хранения MySQL обсуждаются в главе 12.

Применение недопустимых значений

Столбец gender таблицы person может иметь только два значения: 'M' для мужчин и 'F' для женщин. Если по ошибке делается попытка задать любое другое значение, будет получен следующий ответ:

```
mysql> UPDATE person
    -> SET gender = 'Z'
    -> WHERE person_id = 1;
Query OK, 1 row affected, 1 warning (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 1
```

Выражение `update` не дало сбой, но было сформировано предупреждение. Чтобы увидеть описание предупреждения, можно выполнить команду `show warnings` (показать предупреждения):

```
mysql> SHOW WARNINGS;
+---------+------+-------------------------------------------------+
| Level   | Code | Message                                         |
+---------+------+-------------------------------------------------+
| Warning | 1265 | Data truncated for column 'gender' at row 1     |
+---------+------+-------------------------------------------------+
1 row in set (0.00 sec)
```

Это я назвал бы *безопасной ошибкой* (*soft error*), поскольку сервер MySQL не забраковал выражение, но также и не произвел ожидаемых результатов. Чтобы решить эту проблему, сервер MySQL заполняет столбец `gender` пустой строкой (`''`) – определенно не тем, что предполагалось получить. Лично я предпочел бы, чтобы выражение было отвергнуто с сообщением об ошибке, что и сделали бы большинство других серверов БД.

Недействительные преобразования дат

Если предлагаемая для заполнения столбца `date` строка не соответствует ожидаемому формату, будет сформирована другая безопасная ошибка. Вот пример использования формата даты, не соответствующего применяемому по умолчанию «YYYY-MM-DD»:

```
mysql> UPDATE person
    -> SET birth_date = 'DEC-21-1980'
    -> WHERE person_id = 1;
Query OK, 1 rows affected, 1 warning (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 1
```

Команда `show warnings` выдает следующее:

```
mysql> SHOW WARNINGS;
+---------+------+-------------------------------------------------+
| Level   | Code | Message                                         |
+---------+------+-------------------------------------------------+
| Warning | 1265 | Data truncated for column 'birth_date' at row 1 |
+---------+------+-------------------------------------------------+
```

Поскольку это столбец типа `date`, `birth_date` не может быть пустой строкой, поэтому MySQL задает значение `'0000-00-00'`, как показано ниже:

```
mysql> SELECT birth_date
    -> FROM person
    -> WHERE person_id = 1;
```

```
+------------+
| birth_date |
+------------+
| 0000-00-00 |
+------------+
```

Опять же я бы предпочел предупреждению ошибку, поскольку сейчас в таблицу `person` внесены неверные данные (`0000-00-00`).

Банковская схема

Далее в книге используется группа таблиц, моделирующих банк, обслуживающий небольшой населенный пункт. Среди этих таблиц можно назвать `Employee` (сотрудник), `Branch` (отделение), `Account` (счет), `Customer` (клиент), `Product` (услуга), `Transaction` (транзакция) и `Loan` (заем). Всю схему и пример данных следует создать после выполнения 13 шагов для загрузки сервера MySQL и формирования примера данных, приведенных в начале этой главы. Диаграмму с таблицами, их столбцами и связями можно увидеть в приложении А.

В табл. 2.9 показаны все таблицы, используемые в банковской схеме, и даны их краткие описания.

Таблица 2.9. Описания банковской схемы

Таблица	Описание
`Account`	Конкретный счет, открытый для конкретного клиента
`Business`	Клиент-юридическое лицо (подтип таблицы `Customer`)
`Customer`	Физическое или юридическое лицо, известные банку
`Department`	Группа сотрудников банка, реализующая определенную банковскую функцию
`Employee`	Человек, работающий в банке
`Individual`	Клиент-физическое лицо (подтип таблицы `Customer`)
`Officer`	Человек, которому разрешено вести дела от лица клиента-юридического лица
`Product`	Услуга банка, предлагаемая клиентам
`Product_type`	Группа функционально схожих услуг
`Transaction`	Изменение баланса счета

Не бойтесь экспериментировать с таблицами, добавляйте собственные таблицы, чтобы расширить бизнес-функцию банка. Чтобы получить гарантированно нетронутый пример данных, всегда можно удалить БД и восстановить ее из загруженного файла.

Чтобы посмотреть доступные таблицы БД, можно использовать команду `show tables`:

```
mysql> SHOW TABLES;
+----------------+
| Tables_in_bank |
+----------------+
| account        |
| branch         |
| business       |
| customer       |
| department     |
| employee       |
| favorite_food  |
| individual     |
| officer        |
| person         |
| product        |
| product_type   |
| transaction    |
+----------------+
13 rows in set (0.10 sec)
```

Вместе с 11 таблицами банковской схемы в список вошли две таблицы, созданные в этой главе – `person` и `favorite_food`. Эти таблицы не будут использоваться в последующих главах, поэтому их можно свободно удалить с помощью следующих команд:

```
mysql> DROP TABLE favorite_food;
Query OK, 0 rows affected (0.56 sec)

mysql> DROP TABLE person;
Query OK, 0 rows affected (0.05 sec)
```

Посмотреть столбцы таблицы можно с помощью команды `describe`. Вот пример результата выполнения этой команды для таблицы `customer`:

```
mysql> DESC customer;
+--------------+------------------+------+-----+---------+----------------+
| Field        | Type             | Null | Key | Default | Extra          |
+--------------+------------------+------+-----+---------+----------------+
| cust_id      | int(10) unsigned |      | PRI | NULL    | auto_increment |
| fed_id       | varchar(12)      |      |     |         |                |
| cust_type_cd | enum('I','B')    |      |     | I       |                |
| address      | varchar(30)      | YES  |     | NULL    |                |
| city         | varchar(20)      | YES  |     | NULL    |                |
| state        | varchar(20)      | YES  |     | NULL    |                |
| postal_code  | varchar(10)      | YES  |     | NULL    |                |
+--------------+------------------+------+-----+---------+----------------+
7 rows in set (0.03 sec)
```

Чем свободнее вы будете чувствовать себя с примером БД, тем понятнее будут примеры и, следовательно, концепции, представленные в следующих главах.

3

Азбука запросов

Первые две главы содержали несколько примеров запросов к базам данных (т. е. выражений `select`). Теперь пришло время поближе рассмотреть разные части выражения `select` и их взаимодействие.

Механика запроса

Прежде чем анализировать выражение `select`, любопытно узнать, как сервер MySQL (или, коли на то пошло, любой сервер БД) выполняет запросы. Если вы используете клиентскую программу командной строки *mysql* (что я предполагаю), то уже зарегистрировались на сервере MySQL, предоставив свои имя пользователя и пароль (и, возможно, имя хоста, если сервер MySQL выполняется на другом компьютере). Как только сервер проверил правильность имени пользователя и пароля, для вас создается соединение с БД. Это соединение удерживается запросившим его приложением (которым в данном случае является инструмент *mysql*) до тех пор, пока приложение не высвободит соединение (например, в результате введения команды *quit*) или пока соединение не будет закрыто сервером (например, при выключении сервера). Каждому соединению с сервером MySQL присваивается идентификатор (ID), предоставляемый пользователю сразу после регистрации:

```
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 2 to server version: 4.1.11-nt

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.
```

В данном случае ID соединения – 2. Эта информация может быть полезной администратору БД в случае каких-либо неполадок. Например, если требуется прервать плохо сформированный запрос, выполняющийся часами.

После того как сервер открыл соединение, проверив достоверность имени пользователя и пароля, можно выполнять запросы (и другие

SQL-выражения). При каждом запросе перед выполнением выражения сервер проверяет следующее:

- Есть ли у вас разрешение на выполнение выражения?
- Есть ли у вас разрешение на доступ к необходимым данным?
- Правилен ли синтаксис выражения?

Если выражение проходит все три теста, оно передается *оптимизатору запросов*, работа которого заключается в определении наиболее эффективного способа выполнения запроса. Оптимизатор рассмотрит порядок соединения таблиц, перечисленных в запросе, и доступные индексы, а затем определит *план выполнения*, используемый сервером при выполнении этого запроса.

Многие из вас заинтересуются тем, как понять и воздействовать на выбор сервером БД плана выполнения. Читатели, использующие MySQL, могут посмотреть книгу «High Performance MySQL» (O'Reilly). Кроме прочего, вы научитесь генерировать индексы, анализировать планы выполнения, оказывать влияние на оптимизатор посредством подсказок запроса и настраивать параметры запуска сервера. Для пользователей Oracle Database или SQL Server есть десятки книг по этой тематике.

По завершении выполнения запроса сервер возвращает в вызывающее приложение (опять же в инструмент *mysql*) *результирующий набор* (*result set*). Как было упомянуто в главе 1, результирующий набор – это просто еще одна таблица со строками и столбцами. Если по запросу не удается найти никаких данных, инструмент *mysql* отобразит сообщение, приведенное в конце следующего примера:

```
mysql> SELECT emp_id, fname, lname
    -> FROM employee
    -> WHERE lname = 'Bkadfl';
Empty set(0.00 sec)
```

Если запрос возвращает одну или более строк, программа форматирует результаты, добавляя заголовки столбцов и обводя столбцы рамкой из символов -, | и +, как показано в следующем примере:

```
mysql> SELECT fname, lname
-> FROM employee;
+----------+-----------+
| fname    | lname     |
+----------+-----------+
| Michael  | Smith     |
| Susan    | Barker    |
| Robert   | Tyler     |
| Susan    | Hawthorne |
| John     | Gooding   |
| Helen    | Fleming   |
| Chris    | Tucker    |
```

```
| Sarah    | Parker    |
| Jane     | Grossman  |
| Paula    | Roberts   |
| Thomas   | Ziegler   |
| Samantha | Jameson   |
| John     | Blake     |
| Cindy    | Mason     |
| Frank    | Portman   |
| Theresa  | Markham   |
| Beth     | Fowler    |
| Rick     | Tulman    |
+----------+-----------+
18 rows in set (0.00 sec)
```

Этот запрос возвращает имена и фамилии всех сотрудников из таблицы `employee`. После отображения последней строки данных инструмент *mysql* выводит на экран сообщение, указывающее, сколько строк было возвращено, в данном случае – 18 строк.

Блоки запроса

Выражение `select` могут образовывать несколько компонентов, или *блоков* (*clauses*). Хотя при работе с MySQL обязательным является только один из них (блок `select`), обычно в запрос включаются, по крайней мере, два-три из шести доступных блоков. В табл. 3.1 показаны разные блоки и их назначение.

Таблица 3.1. Блоки запроса

Блок	Назначение
`Select`	Определяет столбцы, которые должны быть включены в результирующий набор запроса
`From`	Указывает таблицы, из которых должны быть извлечены данные, и то, как эти таблицы должны быть соединены
`Where`	Ограничивает число строк в окончательном результирующем наборе
`Group by`	Используется для группировки строк по одинаковым значениям столбцов
`Having`	Ограничивает число строк в окончательном результирующем наборе с помощью группировки данных
`Order by`	Сортирует строки окончательного результирующего набора по одному или более столбцам

Все показанные в табл. 3.1 блоки включены в спецификацию ANSI. Кроме того, есть еще несколько блоков, используемых только в MySQL. Они будут рассмотрены в приложении B. В следующих разделах мы подробнее рассмотрим использование шести основных блоков запроса.

Блок select

Даже несмотря на то, что блок `select` является первым в выражении `select`, сервер БД обрабатывает его одним из последних. Причина в том, что прежде чем можно будет определить, что включать в окончательный результирующий набор, необходимо знать все столбцы, которые *могли бы* быть включены в этот набор. Поэтому, чтобы полностью понять роль блока `select`, надо немного разобраться с блоком `from`. Вот запрос для начала:

```
mysql> SELECT *
    -> FROM department;
+---------+----------------+
| dept_id | name           |
+---------+----------------+
|       1 | Operations     |
|       2 | Loans          |
|       3 | Administration |
+---------+----------------+
3 rows in set (0.04 sec)
```

В данном запросе в блоке `from` указана всего одна таблица (`department`), и блок `select` показывает, что в результирующий набор должны быть включены *все* столбцы (обозначено символом «*») таблицы `department`. Этот запрос можно перевести на естественный язык следующим образом:

Покажи мне все столбцы таблицы `department`.

Выбрать все столбцы можно не только с помощью символа звездочки, но и явно указав имена интересующих столбцов:

```
mysql> SELECT dept_id, name
    -> FROM department;
+---------+----------------+
| dept_id | name           |
+---------+----------------+
|       1 | Operations     |
|       2 | Loans          |
|       3 | Administration |
+---------+----------------+
3 rows in set (0.01 sec)
```

Результаты аналогичны первому запросу, поскольку в блоке `select` указаны все столбцы таблицы `department` (`dept_id` и `name`). А можно выбрать только некоторые из столбцов таблицы `department`:

```
mysql> SELECT name
    -> FROM department;
```

```
+----------------+
| name           |
+----------------+
| Operations     |
| Loans          |
| Administration |
+----------------+
3 rows in set (0.00 sec)
```

Таким образом, задача блока `select` заключается в следующем:

> Блок `select` определяет, какие из всех возможных столбцов должны быть включены в результирующий набор запроса.

Если бы приходилось выбирать столбцы только из таблицы или таблиц, указанных в блоке `from`, было бы скучновато. Хорошо, что можно добавить остроты, включив в блок `select` такие вещи, как:

- Литералы, например числа или строки
- Выражения, например `transaction.amount * -1`
- Вызовы встроенных функций, например `ROUND(transaction.amount, 2)`

Следующий запрос демонстрирует использование столбца таблицы, литерала, выражения и вызова встроенной функции в одном запросе к таблице `employee`:

```
mysql> SELECT emp_id,
    ->  'ACTIVE',
    ->  emp_id * 3.14159,
    ->  UPPER(lname)
    -> FROM employee;
+--------+--------+------------------+--------------+
| emp_id | ACTIVE | emp_id * 3.14159 | UPPER(lname) |
+--------+--------+------------------+--------------+
|      1 | ACTIVE |          3.14159 | SMITH        |
|      2 | ACTIVE |          6.28318 | BARKER       |
|      3 | ACTIVE |          9.42477 | TYLER        |
|      4 | ACTIVE |         12.56636 | HAWTHORNE    |
|      5 | ACTIVE |         15.70795 | GOODING      |
|      6 | ACTIVE |         18.84954 | FLEMING      |
|      7 | ACTIVE |         21.99113 | TUCKER       |
|      8 | ACTIVE |         25.13272 | PARKER       |
|      9 | ACTIVE |         28.27431 | GROSSMAN     |
|     10 | ACTIVE |         31.41590 | ROBERTS      |
|     11 | ACTIVE |         34.55749 | ZIEGLER      |
|     12 | ACTIVE |         37.69908 | JAMESON      |
|     13 | ACTIVE |         40.84067 | BLAKE        |
|     14 | ACTIVE |         43.98226 | MASON        |
|     15 | ACTIVE |         47.12385 | PORTMAN      |
|     16 | ACTIVE |         50.26544 | MARKHAM      |
|     17 | ACTIVE |         53.40703 | FOWLER       |
|     18 | ACTIVE |         56.54862 | TULMAN       |
```

```
+--------+--------+-----------------+--------------+
18 rows in set (0.05 sec)
```

Выражения и встроенные функции будут подробно рассмотрены позже, но я хотел дать представление о том, что может быть включено в блок `select`. Если требуется только выполнить встроенную функцию или вычислить простое выражение, можно вообще обойтись без блока `from`. Вот пример:

```
mysql> SELECT VERSION( ),
    ->   USER( ),
    ->   DATABASE( );
+-----------+-------------------+------------+
| VERSION() | USER()            | DATABASE( )|
+-----------+-------------------+------------+
| 4.1.11-nt | lrngsql@localhost | bank       |
+-----------+-------------------+------------+
1 row in set (0.02 sec)
```

Поскольку данный запрос просто вызывает три встроенные функции и не извлекает данные из таблиц, блок `from` здесь не нужен.

Псевдонимы столбцов

Хотя инструмент *mysql* и генерирует имена для столбцов, возвращаемых в результате запроса, вы можете задавать эти имена самостоятельно. Кроме того, что при желании можно дать другое имя столбцу из таблицы (если у него «плохое» или неоднозначное имя), практически наверняка вы захотите по-своему назвать те столбцы результирующего набора, которые будут сформированы в результате выполнения выражения или встроенной функции. Сделать это можно добавлением *псевдонима столбца* после каждого элемента блока `select`. Вот предыдущий запрос к таблице `employee`, в котором для трех столбцов указаны псевдонимы:

```
mysql> SELECT emp_id,
    ->   'ACTIVE' status,
    ->   emp_id * 3.14159 empid_x_pi,
    ->   UPPER(lname) last_name_upper
    -> FROM employee;
+--------+--------+------------+-----------------+
| emp_id | status | empid_x_pi | last_name_upper |
+--------+--------+------------+-----------------+
|      1 | ACTIVE |    3.14159 | SMITH           |
|      2 | ACTIVE |    6.28318 | BARKER          |
|      3 | ACTIVE |    9.42477 | TYLER           |
|      4 | ACTIVE |   12.56636 | HAWTHORNE       |
|      5 | ACTIVE |   15.70795 | GOODING         |
|      6 | ACTIVE |   18.84954 | FLEMING         |
|      7 | ACTIVE |   21.99113 | TUCKER          |
|      8 | ACTIVE |   25.13272 | PARKER          |
```

```
|      9 | ACTIVE |   28.27431 | GROSSMAN        |
|     10 | ACTIVE |   31.41590 | ROBERTS         |
|     11 | ACTIVE |   34.55749 | ZIEGLER         |
|     12 | ACTIVE |   37.69908 | JAMESON         |
|     13 | ACTIVE |   40.84067 | BLAKE           |
|     14 | ACTIVE |   43.98226 | MASON           |
|     15 | ACTIVE |   47.12385 | PORTMAN         |
|     16 | ACTIVE |   50.26544 | MARKHAM         |
|     17 | ACTIVE |   53.40703 | FOWLER          |
|     18 | ACTIVE |   56.54862 | TULMAN          |
+--------+--------+------------+-----------------+
18 rows in set (0.00 sec)
```

Как видно из заголовков столбцов, второй, третий и четвертый столбцы теперь имеют осмысленные имена, а не обозначены просто функцией или выражением, сформировавшим этот столбец. Если посмотреть на блок `select`, можно увидеть, что псевдонимы `status`, `empid_x_pi` и `last_name_upper` добавлены после второго, третьего и четвертого столбцов. Думаю, все согласятся с тем, что с присвоенными псевдонимами столбцов выходные данные стали понятнее; кроме того, с ними легче работать программно, если запросы формируются из Java или C#, а не интерактивно посредством инструмента командной строки *mysql*.

Уничтожение дубликатов

В некоторых случаях запрос может возвратить дублирующие строки данных. Например, при выборе ID всех клиентов, имеющих счета, было бы представлено следующее:

```
mysql> SELECT cust_id
    -> FROM account;
+---------+
| cust_id |
+---------+
|       1 |
|       1 |
|       1 |
|       2 |
|       2 |
|       3 |
|       3 |
|       4 |
|       4 |
|       4 |
|       5 |
|       6 |
|       6 |
|       7 |
|       8 |
|       8 |
|       9 |
```

```
|       9 |
|       9 |
|      10 |
|      10 |
|      11 |
|      12 |
|      13 |
+---------+
24 rows in set (0.00 sec)
```

Поскольку у некоторых клиентов по нескольку счетов, один и тот же ID клиента будет выведен столько раз, сколько счетов имеет клиент. Но, очевидно, целью данного запроса является *выбор* клиентов, имеющих счета, а не получение ID клиента для каждой строки таблицы `account`. Добиться этого можно, поместив ключевое слово `distinct` (отличный) непосредственно после ключевого слова `select`, как в следующем примере:

```
mysql> SELECT DISTINCT cust_id
    -> FROM account;
+---------+
| cust_id |
+---------+
|       1 |
|       2 |
|       3 |
|       4 |
|       5 |
|       6 |
|       7 |
|       8 |
|       9 |
|      10 |
|      11 |
|      12 |
|      13 |
+---------+
13 rows in set (0.01 sec)
```

Теперь в результирующем наборе 13 строк, по одной для каждого клиента, а не 24 строки, по одной для каждого счета.

Если не требуется, чтобы сервер удалял дублирующие данные, или вы уверены, что в результирующем наборе их не будет, вместо `DISTINCT` можно указать ключевое слово `ALL` (все). Однако ключевое слово `ALL` применяется по умолчанию и в явном указании не нуждается, поэтому большинство программистов не включает `ALL` в запросы.

Запомните, что формирование набора уникальных значений требует сортировки данных, что в случае больших результирующих наборов может занять много времени. Не поддавайтесь

соблазну использовать DISTINCT только для того, чтобы гарантировать отсутствие дублирования; лучше потратьте некоторое время на осмысление данных, с которыми работаете, чтобы уже наверняка знать, где дублирование возможно.

Блок from

До сих пор мы рассматривали запросы, в блоках from которых была указана только одна таблица. Хотя большинство книг по SQL определяют блок from просто как список из одной или более таблиц, мне бы хотелось расширить это определение следующим образом:

> Блок from определяет таблицы, используемые запросом, а также средства связывания таблиц.

Это определение включает две разные, но взаимосвязанные концепции, которые будут изучены в следующих разделах.

Таблицы

При встрече с термином table большинство людей представляют себе набор взаимосвязанных строк, хранящихся в базе данных. Хотя один из типов таблиц действительно описывается именно так, мне бы хотелось использовать это слово в более общем значении – избавиться от любого упоминания о способах хранения данных, сосредоточившись только на наборе взаимосвязанных строк. Этому свободному определению соответствуют три разных типа таблиц:

- Постоянные таблицы (т. е. созданные с помощью выражения create table)
- Временные таблицы (т. е. строки, возвращенные подзапросом)
- Виртуальные таблицы (представления) (т. е. созданные с помощью выражения create view)

Каждый из этих типов таблиц может быть включен в блок from запроса. На данный момент вы уже вполне освоили включение постоянных таблиц, поэтому далее кратко описаны другие типы таблиц, которые могут использоваться в блоке from.

Таблицы, формируемые подзапросом

Подзапрос (subquery) – это запрос, содержащийся в другом запросе. Подзапросы заключаются в круглые скобки и могут располагаться в различных частях выражения select. Однако в рамках блока from подзапрос выполняет функцию формирования временной таблицы, видимой для всех остальных блоков запроса и способной взаимодействовать с другими таблицами, указанными в блоке from. Вот простой пример:

```
mysql> SELECT e.emp_id, e.fname, e.lname
    -> FROM (SELECT emp_id, fname, lname, start_date, title
```

```
    ->   FROM employee) e;
+--------+----------+-----------+
| emp_id | fname    | lname     |
+--------+----------+-----------+
|      1 | Michael  | Smith     |
|      2 | Susan    | Barker    |
|      3 | Robert   | Tyler     |
|      4 | Susan    | Hawthorne |
|      5 | John     | Gooding   |
|      6 | Helen    | Fleming   |
|      7 | Chris    | Tucker    |
|      8 | Sarah    | Parker    |
|      9 | Jane     | Grossman  |
|     10 | Paula    | Roberts   |
|     11 | Thomas   | Ziegler   |
|     12 | Samantha | Jameson   |
|     13 | John     | Blake     |
|     14 | Cindy    | Mason     |
|     15 | Frank    | Portman   |
|     16 | Theresa  | Markham   |
|     17 | Beth     | Fowler    |
|     18 | Rick     | Tulman    |
+--------+----------+-----------+
18 rows in set (0.00 sec)
```

Здесь подзапрос к таблице `employee` возвращает пять столбцов, а *основной запрос* (*containing query*) ссылается на три из пяти доступных столбцов. Запрос ссылается на подзапрос посредством псевдонима, в данном случае `e`. Это упрощенный, практически бесполезный пример подзапроса в блоке `from`; подробный рассказ о подзапросах можно найти в главе 9.

Представления

Представление (view) – это запрос, хранящийся в словаре данных (data dictionary). Оно выглядит и работает как таблица, но с представлением не ассоциированы никакие данные (вот почему я называю это *виртуальной* таблицей). При выполнении запроса к представлению запрос соединяется с описанием представления и создается окончательный запрос, который и будет выполнен.

Чтобы продемонстрировать это, приведу описание представления, запрашивающего таблицу `employee` и включающего вызов встроенной функции:

```
CREATE VIEW employee_vw AS
SELECT emp_id, fname, lname,
  YEAR(start_date) start_year
FROM employee;
```

После создания представления никакие дополнительные данные не создаются: сервер просто сохраняет выражение `select` для дальнейшего

использования. Теперь, когда представление существует, можно делать запросы к нему:

```
SELECT emp_id, start_year
FROM employee_vw;

 Emp_id  start_year
-------- ----------
      1        2001
      2        2002
      3        2000
      4        2002
      5        2003
      6        2004
      7        2004
      8        2002
      9        2002
     10        2002
     11        2000
     12        2003
     13        2000
     14        2002
     15        2003
     16        2001
     17        2002
     18        2002
```

Представления создаются по разным причинам, в том числе с целью скрыть столбцы от пользователей и упростить сложно устроенные БД.

MySQL до версии 5.0.1 не поддерживает представления. Однако они широко используются другими серверами БД, поэтому тот, кто планирует работать с MySQL, должен помнить о них.

Поскольку MySQL версии 4.1.11 не включает представления, в предыдущем запросе намеренно не показано приглашение *mysql>* и обычное форматирование результирующего набора. Этот же прием применяется в других главах книги при описании возможности SQL, еще не реализованной в MySQL.

Связи таблиц

Второе отступление от определения простого блока `from`: если в блоке `from` присутствует более одной таблицы, обязательно должны быть включены и условия, используемые для *связывания* (*link*) таблиц. Это не требование MySQL или какого-то другого сервера БД, а утвержденный ANSI метод соединения нескольких таблиц, и это способ, наиболее переносимый между серверами БД. Соединение нескольких таблиц будет подробно рассматриваться в главах 5 и 10; здесь приведен лишь простой пример для утоления любопытства:

```
mysql> SELECT employee.emp_id, employee.fname,
    ->   employee.lname, department.name dept_name
```

```
    -> FROM employee INNER JOIN department
    ->   ON employee.dept_id = department.dept_id;
+--------+----------+-----------+----------------+
| emp_id | fname    | lname     | dept_name      |
+--------+----------+-----------+----------------+
|      1 | Michael  | Smith     | Administration |
|      2 | Susan    | Barker    | Administration |
|      3 | Robert   | Tyler     | Administration |
|      4 | Susan    | Hawthorne | Operations     |
|      5 | John     | Gooding   | Loans          |
|      6 | Helen    | Fleming   | Operations     |
|      7 | Chris    | Tucker    | Operations     |
|      8 | Sarah    | Parker    | Operations     |
|      9 | Jane     | Grossman  | Operations     |
|     10 | Paula    | Roberts   | Operations     |
|     11 | Thomas   | Ziegler   | Operations     |
|     12 | Samantha | Jameson   | Operations     |
|     13 | John     | Blake     | Operations     |
|     14 | Cindy    | Mason     | Operations     |
|     15 | Frank    | Portman   | Operations     |
|     16 | Theresa  | Markham   | Operations     |
|     17 | Beth     | Fowler    | Operations     |
|     18 | Rick     | Tulman    | Operations     |
+--------+----------+-----------+----------------+
18 rows in set (0.05 sec)
```

Предыдущий запрос выводит данные из таблиц `employee` (`emp_id`, `fname`, `lname`) и `department` (`name`), поэтому обе таблицы включены в блок `from`. Механизм связывания двух таблиц (называемый *соединением* (*join*)) заключается в присоединении данных об отделе, в котором работает сотрудник, хранящихся в таблице `employee`. Таким образом, серверу БД отдается распоряжение использовать значение столбца `dept_id` таблицы `employee` для поиска соответствующего названия отдела в таблице `department`. Условия соединения находятся в подблоке `on` блока `from`. В данном случае условие соединения: `ON e.dept_id = d.dept_id`. Всестороннее обсуждение соединения нескольких таблиц также можно найти в главе 5.

Определение псевдонимов таблиц

При соединении нескольких таблиц в одном запросе вам понадобится идентифицировать таблицу, на которую делается ссылка при указании столбцов в блоках `select`, `where`, `group by`, `having` и `order by`. Дать ссылку на таблицу вне блока `from` можно одним из двух способов:

- Использовать полное имя таблицы, например `employee.emp_id`.
- Присвоить каждой таблице псевдоним и использовать его в запросе.

В предыдущем запросе я решил использовать в блоках `select` и `on` полное имя таблицы. А вот как выглядит этот же запрос с применением псевдонимов:

```
SELECT e.emp_id, e.fname, e.lname,
  d.name dept_name
FROM employee e INNER JOIN department d
  ON e.dept_id = d.dept_id;
```

Если внимательнее посмотреть на блок `from`, видно, что таблица `employee` получила псевдоним `e`, а таблица `department` – псевдоним `d`. Затем эти псевдонимы используются в блоке `on` при описании условия соединения, а также в блоке `select` при задании столбцов, которые должны быть включены в результирующий набор. Надеюсь, все согласятся, что использование псевдонимов делает выражение более компактным, не приводя к путанице (при условии разумного выбора псевдонимов).

Блок where

До сих пор запросы, приводимые в данной главе, осуществляли выбор всех строк из таблиц `employee`, `department` или `account` (кроме примера с ключевым словом `distinct`). Однако чаще всего извлекать *все* строки таблицы не требуется, и нужен способ, позволяющий отфильтровывать строки, не представляющие интереса. Это работа для блока `where`.

Блок `where` – это механизм отсеивания нежелательных строк из результирующего набора.

Например, требуется извлечь из таблицы `employee` данные, но только для сотрудников, нанятых в качестве старших операционистов (head tellers). В следующем запросе блок `where` служит для извлечения *только* четырех старших операционистов:

```
mysql> SELECT emp_id, fname, lname, start_date, title
    -> FROM employee
    -> WHERE title = 'Head Teller';
+--------+---------+---------+------------+-------------+
| emp_id | fname   | lname   | start_date | title       |
+--------+---------+---------+------------+-------------+
|      6 | Helen   | Fleming | 2004-03-17 | Head Teller |
|     10 | Paula   | Roberts | 2002-07-27 | Head Teller |
|     13 | John    | Blake   | 2000-05-11 | Head Teller |
|     16 | Theresa | Markham | 2001-03-15 | Head Teller |
+--------+---------+---------+------------+-------------+
4 rows in set (0.00 sec)
```

В данном случае блоком `where` были отсеяны 14 из 18 строк. Этот блок `where` содержит всего одно *условие фильтрации* (*filter condition*), но этих условий может быть столько, сколько потребуется. Условия разделяются с помощью таких операторов, как `and`, `or` и `not` (подробно блок `where` и условия фильтрации обсуждаются в главе 4). Вот расширенный вариант предыдущего запроса со вторым условием – должны быть включены только сотрудники, принятые на работу после 1 января 2002 года:

```
mysql> SELECT emp_id, fname, lname, start_date, title
    -> FROM employee
    -> WHERE title = 'Head Teller'
    ->   AND start_date > '2002-01-01';
+--------+-------+---------+------------+-------------+
| emp_id | fname | lname   | start_date | title       |
+--------+-------+---------+------------+-------------+
|      6 | Helen | Fleming | 2004-03-17 | Head Teller |
|     10 | Paula | Roberts | 2002-07-27 | Head Teller |
+--------+-------+---------+------------+-------------+
2 rows in set (0.00 sec)
```

По первому условию (`title = 'Head Teller'`) было отфильтровано 14 из 18 строк, а по второму (`start_date > '2002-01-01'`) – еще 2. В итоге в результирующем наборе осталось 2 строки. Давайте посмотрим, что произойдет, если заменить разделяющий условия оператор `and` оператором `or`:

```
mysql> SELECT emp_id, fname, lname, start_date, title
    -> FROM employee
    -> WHERE title = 'Head Teller'
    ->   OR start_date > '2002-01-01';
+--------+----------+-----------+------------+--------------------+
| emp_id | fname    | lname     | start_date | title              |
+--------+----------+-----------+------------+--------------------+
|      2 | Susan    | Barker    | 2002-09-12 | Vice President     |
|      4 | Susan    | Hawthorne | 2002-04-24 | Operations Manager |
|      5 | John     | Gooding   | 2003-11-14 | Loan Manager       |
|      6 | Helen    | Fleming   | 2004-03-17 | Head Teller        |
|      7 | Chris    | Tucker    | 2004-09-15 | Teller             |
|      8 | Sarah    | Parker    | 2002-12-02 | Teller             |
|      9 | Jane     | Grossman  | 2002-05-03 | Teller             |
|     10 | Paula    | Roberts   | 2002-07-27 | Head Teller        |
|     12 | Samantha | Jameson   | 2003-01-08 | Teller             |
|     13 | John     | Blake     | 2000-05-11 | Head Teller        |
|     14 | Cindy    | Mason     | 2002-08-09 | Teller             |
|     15 | Frank    | Portman   | 2003-04-01 | Teller             |
|     16 | Theresa  | Markham   | 2001-03-15 | Head Teller        |
|     17 | Beth     | Fowler    | 2002-06-29 | Teller             |
|     18 | Rick     | Tulman    | 2002-12-12 | Teller             |
+--------+----------+-----------+------------+--------------------+
15 rows in set (0.00 sec)
```

Посмотрев на выходные данные, можно увидеть, что в результирующий набор включены все четыре старших операциониста (Head Teller), а также все остальные сотрудники, приступившие к работе в банке после 1 января 2002 года. Для 15 из 18 сотрудников из таблицы `employee` выполняется по крайней мере одно из двух условий. Таким образом, чтобы строка попала в результирующий набор, когда условия разделяются оператором `and`, для нее должны выполняться *все* условия; а при использовании оператора `or` достаточно, чтобы выполнялось только *одно* из условий.

А как быть, если вам нужно задействовать в блоке `where` оба оператора – `and` и `or`? Рад, что спросили. Необходимо сгруппировать условия с помощью круглых скобок. Следующий запрос составлен так, что в результирующий набор должны попасть только те сотрудники, которые являются старшими операционистами (Head Teller) и начали работать в компании позже 1 января 2002 года, *или* простые операционисты (Teller), начавшие работать после 1 января 2003 года:

```
mysql> SELECT emp_id, fname, lname, start_date, title
    -> FROM employee
    -> WHERE (title = 'Head Teller' AND start_date > '2002-01-01')
    ->   OR (title = 'Teller' AND start_date > '2003-01-01');
+--------+----------+---------+------------+-------------+
| emp_id | fname    | lname   | start_date | title       |
+--------+----------+---------+------------+-------------+
|      6 | Helen    | Fleming | 2004-03-17 | Head Teller |
|      7 | Chris    | Tucker  | 2004-09-15 | Teller      |
|     10 | Paula    | Roberts | 2002-07-27 | Head Teller |
|     12 | Samantha | Jameson | 2003-01-08 | Teller      |
|     15 | Frank    | Portman | 2003-04-01 | Teller      |
+--------+----------+---------+------------+-------------+
5 rows in set (0.00 sec)
```

Для разделения групп условий при использовании различных операторов всегда следует применять круглые скобки, чтобы автор запроса, сервер БД и любой специалист, который позже будет работать с этим кодом, понимали, что происходит.

Блоки group by и having

Все рассмотренные до сих пор запросы извлекали необработанные строки данных, не выполняя над ними никаких действий. Однако иногда вам захочется выявить в данных общие направления, для чего серверу БД придется немного поколдовать над ними, прежде чем предоставить вам результирующий набор. Одним из средств для этого является блок `group by`, предназначенный для группировки данных по значениям столбцов. Например, вместо списка сотрудников и отделов, в которых они числятся, нужен список отделов с числом сотрудников, работающих в каждом из них. С блоком `group by` также можно использовать блок `having`, позволяющий фильтровать данные групп аналогично блоку `where`, позволяющему фильтровать необработанные данные.

Я хотел лишь слегка коснуться этих двух блоков, чтобы в дальнейшем они не были неожиданностью для читателей, но они немного сложнее, чем другие четыре блока выражения `select`. Поэтому прошу дождаться главы 8, где полностью описано, как и когда использовать `group by` и `having`.

Блок order by

В общем случае строки результирующего набора запроса возвращаются в произвольном порядке. Если требуется упорядочить результирующий набор определенным образом, необходимо предписать серверу сортировать результаты с помощью блока `order by`:

> Блок `order by` – это механизм сортировки результирующего набора на основе данных столбцов, или выражений, использующих данные столбцов.

Вот, к примеру, еще один взгляд на приведенный ранее запрос к таблице `account`:

```
mysql> SELECT open_emp_id, product_cd
    -> FROM account;
+-------------+------------+
| open_emp_id | product_cd |
+-------------+------------+
|          10 | CHK        |
|          10 | SAV        |
|          10 | CD         |
|          10 | CHK        |
|          10 | SAV        |
|          13 | CHK        |
|          13 | MM         |
|           1 | CHK        |
|           1 | SAV        |
|           1 | MM         |
|          16 | CHK        |
|           1 | CHK        |
|           1 | CD         |
|          10 | CD         |
|          16 | CHK        |
|          16 | SAV        |
|           1 | CHK        |
|           1 | MM         |
|           1 | CD         |
|          16 | CHK        |
|          16 | BUS        |
|          10 | BUS        |
|          16 | CHK        |
|          13 | SBL        |
+-------------+------------+
24 rows in set (0.00 sec)
```

Если требуется проанализировать данные каждого сотрудника, полезно было бы отсортировать результаты по столбцу `open_emp_id`. Для этого просто добавляем этот столбец в блок `order by`:

```
mysql> SELECT open_emp_id, product_cd
    -> FROM account
    -> ORDER BY open_emp_id;
```

```
+-------------+------------+
| open_emp_id | product_cd |
+-------------+------------+
|           1 | CHK        |
|           1 | SAV        |
|           1 | MM         |
|           1 | CHK        |
|           1 | CD         |
|           1 | CHK        |
|           1 | MM         |
|           1 | CD         |
|          10 | CHK        |
|          10 | SAV        |
|          10 | CD         |
|          10 | CHK        |
|          10 | SAV        |
|          10 | CD         |
|          10 | BUS        |
|          13 | CHK        |
|          13 | MM         |
|          13 | SBL        |
|          16 | CHK        |
|          16 | CHK        |
|          16 | SAV        |
|          16 | CHK        |
|          16 | BUS        |
|          16 | CHK        |
+-------------+------------+
24 rows in set (0.00 sec)
```

Теперь легче увидеть, какие типы счетов были открыты каждым сотрудником. Однако было бы гораздо лучше, если бы типы счетов для каждого отдельного сотрудника выводились в определенном порядке; это осуществляется путем добавления в блок `order by` столбца `product_cd` после `open_emp_id`:

```
mysql> SELECT open_emp_id, product_cd
    -> FROM account
    -> ORDER BY open_emp_id, product_cd;
+-------------+------------+
| open_emp_id | product_cd |
+-------------+------------+
|           1 | CD         |
|           1 | CD         |
|           1 | CHK        |
|           1 | CHK        |
|           1 | CHK        |
|           1 | MM         |
|           1 | MM         |
|           1 | SAV        |
|          10 | BUS        |
```

```
|          10 | CD         |
|          10 | CD         |
|          10 | CHK        |
|          10 | CHK        |
|          10 | SAV        |
|          10 | SAV        |
|          13 | CHK        |
|          13 | MM         |
|          13 | SBL        |
|          16 | BUS        |
|          16 | CHK        |
|          16 | CHK        |
|          16 | CHK        |
|          16 | CHK        |
|          16 | SAV        |
+-------------+------------+
24 rows in set (0.00 sec)
```

Теперь результирующий набор отсортирован сначала по ID сотрудников, а затем по типу счета. Учитывается порядок размещения столбцов в блоке `order by`.

Сортировка по возрастанию и убыванию

При сортировке можно задать порядок *по возрастанию* (*ascending*) или *по убыванию* (*descending*) с помощью ключевых слов `asc` и `desc`. По умолчанию выполняется сортировка по возрастанию, поэтому добавлять придется только ключевое слово `desc` – если требуется сортировка по убыванию. Например, по следующему запросу выводится список всех счетов, отсортированный по доступному остатку, начиная с самого большого:

```
mysql> SELECT account_id, product_cd, open_date, avail_balance
    -> FROM account
    -> ORDER BY avail_balance DESC;
+------------+------------+------------+---------------+
| account_id | product_cd | open_date  | avail_balance |
+------------+------------+------------+---------------+
|         24 | SBL        | 2004-02-22 |      50000.00 |
|         23 | CHK        | 2003-07-30 |      38552.05 |
|         20 | CHK        | 2002-09-30 |      23575.12 |
|         13 | CD         | 2004-12-28 |      10000.00 |
|         22 | BUS        | 2004-03-22 |       9345.55 |
|         18 | MM         | 2004-10-28 |       9345.55 |
|         10 | MM         | 2004-09-30 |       5487.09 |
|         14 | CD         | 2004-01-12 |       5000.00 |
|         15 | CHK        | 2001-05-23 |       3487.19 |
|          3 | CD         | 2004-06-30 |       3000.00 |
|          4 | CHK        | 2001-03-12 |       2258.02 |
|         11 | CHK        | 2004-01-27 |       2237.97 |
|          7 | MM         | 2002-12-15 |       2212.50 |
|         19 | CD         | 2004-06-30 |       1500.00 |
```

```
|          1 | CHK        | 2000-01-15 |       1057.75 |
|          6 | CHK        | 2002-11-23 |       1057.75 |
|          9 | SAV        | 2000-01-15 |        767.77 |
|          8 | CHK        | 2003-09-12 |        534.12 |
|          2 | SAV        | 2000-01-15 |        500.00 |
|         16 | SAV        | 2001-05-23 |        387.99 |
|          5 | SAV        | 2001-03-12 |        200.00 |
|         17 | CHK        | 2003-07-30 |        125.67 |
|         12 | CHK        | 2002-08-24 |        122.37 |
|         21 | BUS        | 2002-10-01 |          0.00 |
+------------+------------+------------+---------------+
24 rows in set (0.01 sec)
```

Сортировка по убыванию обычно применяется в ранжирующих запросах вроде «покажи мне пять самых больших доступных остатков». MySQL включает блок `limit` (предел), позволяющий сортировать данные и затем отбрасывать все, кроме первых *X* строк. Блок `limit` обсуждается в приложении B вместе с другими расширениями, не входящими в стандарт ANSI.

Сортировка с помощью выражений

Сортировать результаты по данным столбца легко и приятно, но иногда может потребоваться сортировка по какому-то признаку, который не хранится в БД и, возможно, отсутствует в запросе. Чтобы справиться с этой ситуацией, можно добавить в блок `order by` выражение. Например, требуется сортировать данные клиентов по последним трем разрядам их федерального ID (это либо номер социальной страховки для физических лиц, либо корпоративный ID для юридических лиц):

```
mysql> SELECT cust_id, cust_type_cd, city, state, fed_id
    -> FROM customer
    -> ORDER BY RIGHT(fed_id, 3);
+---------+--------------+------------+-------+-------------+
| cust_id | cust_type_cd | city       | state | fed_id      |
+---------+--------------+------------+-------+-------------+
|       1 | I            | Lynnfield  | MA    | 111-11-1111 |
|      10 | B            | Salem      | NH    | 04-1111111  |
|       2 | I            | Woburn     | MA    | 222-22-2222 |
|      11 | B            | Wilmington | MA    | 04-2222222  |
|       3 | I            | Quincy     | MA    | 333-33-3333 |
|      12 | B            | Salem      | NH    | 04-3333333  |
|      13 | B            | Quincy     | MA    | 04-4444444  |
|       4 | I            | Waltham    | MA    | 444-44-4444 |
|       5 | I            | Salem      | NH    | 555-55-5555 |
|       6 | I            | Waltham    | MA    | 666-66-6666 |
|       7 | I            | Wilmington | MA    | 777-77-7777 |
|       8 | I            | Salem      | NH    | 888-88-8888 |
|       9 | I            | Newton     | MA    | 999-99-9999 |
+---------+--------------+------------+-------+-------------+
13 rows in set (0.01 sec)
```

В этом запросе используется встроенная функция `right()`, которая извлекает последние три символа значения столбца `fed_id` и сортирует строки на основании этого значения.

Сортировка с помощью числовых заместителей

При сортировке с использованием столбцов, перечисленных в блоке `select`, можно ссылаться на столбцы не по имени, а по их *порядковому номеру*. Например, если требуется выполнить сортировку по второму или пятому столбцу, возвращаемому запросом, можно сделать следующее:

```
mysql> SELECT emp_id, title, start_date, fname, lname
    -> FROM employee
    -> ORDER BY 2, 5;
+--------+--------------------+------------+----------+-----------+
| emp_id | title              | start_date | fname    | lname     |
+--------+--------------------+------------+----------+-----------+
|     13 | Head Teller        | 2000-05-11 | John     | Blake     |
|      6 | Head Teller        | 2004-03-17 | Helen    | Fleming   |
|     16 | Head Teller        | 2001-03-15 | Theresa  | Markham   |
|     10 | Head Teller        | 2002-07-27 | Paula    | Roberts   |
|      5 | Loan Manager       | 2003-11-14 | John     | Gooding   |
|      4 | Operations Manager | 2002-04-24 | Susan    | Hawthorne |
|      1 | President          | 2001-06-22 | Michael  | Smith     |
|     17 | Teller             | 2002-06-29 | Beth     | Fowler    |
|      9 | Teller             | 2002-05-03 | Jane     | Grossman  |
|     12 | Teller             | 2003-01-08 | Samantha | Jameson   |
|     14 | Teller             | 2002-08-09 | Cindy    | Mason     |
|      8 | Teller             | 2002-12-02 | Sarah    | Parker    |
|     15 | Teller             | 2003-04-01 | Frank    | Portman   |
|      7 | Teller             | 2004-09-15 | Chris    | Tucker    |
|     18 | Teller             | 2002-12-12 | Rick     | Tulman    |
|     11 | Teller             | 2000-10-23 | Thomas   | Ziegler   |
|      3 | Treasurer          | 2000-02-09 | Robert   | Tyler     |
|      2 | Vice President     | 2002-09-12 | Susan    | Barker    |
+--------+--------------------+------------+----------+-----------+
18 rows in set (0.03 sec)
```

Скорее всего, вы редко будете использовать эту возможность, поскольку если добавить столбец в блок `select` и не изменить порядковые номера в блоке `order by`, результаты будут непредсказуемыми.

Упражнения

Следующие упражнения разработаны для закрепления понимания выражения `select` и его блоков. Решения приведены в приложении C.

3.1

Извлеките ID, имя и фамилию всех банковских сотрудников. Выполните сортировку по фамилии, а затем по имени.

3.2

Извлеките ID счета, ID клиента и доступный остаток всех счетов, имеющих статус `'ACTIVE'` (активный) и доступный остаток более 2500 долларов.

3.3

Напишите запрос к таблице `account`, возвращающий ID сотрудников, открывших счета (используйте столбец `account.open_emp_id`). Результирующий набор должен включать по одной строке для каждого сотрудника.

3.4

В этом запросе к нескольким наборам данных заполните пробелы (обозначенные как *<число>*) так, чтобы получить результат, приведенный ниже:

```
mysql> SELECT p.product_cd, a.cust_id, a.avail_balance
    -> FROM product p INNER JOIN account <1>
    ->   ON p.product_cd = <2>
    -> WHERE p.<3> = 'ACCOUNT';
+------------+---------+---------------+
| product_cd | cust_id | avail_balance |
+------------+---------+---------------+
| CD         |       1 |       3000.00 |
| CD         |       6 |      10000.00 |
| CD         |       7 |       5000.00 |
| CD         |       9 |       1500.00 |
| CHK        |       1 |       1057.75 |
| CHK        |       2 |       2258.02 |
| CHK        |       3 |       1057.75 |
| CHK        |       4 |        534.12 |
| CHK        |       5 |       2237.97 |
| CHK        |       6 |        122.37 |
| CHK        |       8 |       3487.19 |
| CHK        |       9 |        125.67 |
| CHK        |      10 |      23575.12 |
| CHK        |      12 |      38552.05 |
| MM         |       3 |       2212.50 |
| MM         |       4 |       5487.09 |
| MM         |       9 |       9345.55 |
| SAV        |       1 |        500.00 |
| SAV        |       2 |        200.00 |
| SAV        |       4 |        767.77 |
| SAV        |       8 |        387.99 |
+------------+---------+---------------+
21 rows in set (0.02 sec)
```

4

Фильтрация

Бывают случаи, когда требуется работать со всеми строками таблицы, например:

- Удаление всех данных таблицы для того, чтобы загрузить новые данные из другого источника.
- Изменение всех строк таблицы после добавления нового столбца.
- Извлечение всех строк из таблицы очереди сообщений.

В подобных случаях SQL-выражениям не нужен блок `where`, поскольку нет необходимости исключать из рассмотрения какие-либо строки. Однако чаще всего требуется сужать фокус и работать с подмножеством строк таблицы. Поэтому все SQL-выражения для работы с данными (кроме выражения `insert`) включают необязательный блок `where`, где размещаются всевозможные фильтры для ограничения числа строк, подвергаемых воздействию SQL-выражения. Кроме того, в выражение `select` входит блок `having`, в который могут быть включены условия фильтрации, относящиеся к группам данных. В этой главе изучаются различные типы условий фильтрации, которые могут применяться в блоках `where` выражений `select`, `update` и `delete`.

Оценка условия

Блок `where` может содержать одно или более условий, разделенных операторами `and` и `or`. При использовании только оператора `and` строка будет включена в результирующий набор в случае истинности (`True`) всех условий для нее. Рассмотрим следующий блок `where`:

```
WHERE title = 'Teller' AND start_date < '2003-01-01'
```

Исходя из этих двух условий, из рассмотрения будет исключен любой сотрудник, не являющийся операционистом или работающий в банке

начиная с 2003 года. В данном примере используется только два условия, но если условия в блоке `where` разделены оператором `and`, то независимо от их количества строка попадет в результирующий набор, только если все условия для нее будут истинны.

Если все условия в блоке `where` разделены оператором `or`, то чтобы строка вошла в результирующий набор, должно выполниться (принять значение `true`) хотя бы одно из них. Рассмотрим следующие два условия:

```
WHERE title = 'Teller' OR start_date < '2003-01-01'
```

Теперь есть несколько вариантов условий, по которым строка `employee` может быть включена в результирующий набор:

- Сотрудник является операционистом и был принят на работу до 2003 года.
- Сотрудник является операционистом и был принят на работу после 1 января 2003.
- Сотрудник не является операционистом, но был принят на работу до 2003 года.

В табл. 4.1 показаны возможные результаты вычисления блока `where`, содержащего два условия, разделенных оператором `or`.

Таблица 4.1. Результаты вычисления выражений с двумя условиями, разделенными оператором or

Промежуточный результат	Конечный результат
`WHERE true OR true`	`True`
`WHERE true OR false`	`True`
`WHERE false OR true`	`True`
`WHERE false OR false`	`False`

В предыдущем примере единственный вариант исключения строки из результирующего набора – если сотрудник не является операционистом и был принят на работу начиная с 1 января 2003 года.

Скобки

Если блок включает три или больше условий с использованием как оператора `and`, так и `or`, следует применять круглые скобки. Это сделает намерения запроса понятными и для сервера БД, и для всех, кто будет читать код. Вот блок `where`, расширяющий предыдущий пример. Он проверяет, работает ли сотрудник в банке до сих пор:

```
WHERE end_date IS NULL
  AND (title = 'Teller' OR start_date < '2003-01-01')
```

Теперь имеем три условия. Чтобы строка попала в конечный результирующий набор, первое условие для нее должно быть истинным (`true`), а также истинным должно быть второе *или* третье условие (или оба).

В табл. 4.2 показаны возможные результаты вычисления этого блока where.

Таблица 4.2. Результаты вычисления выражений с тремя условиями, разделенными операторами and и or

Промежуточный результат	Конечный результат
WHERE true AND (true OR true)	True
WHERE true AND (true OR false)	True
WHERE true AND (false OR true)	True
WHERE true AND (false OR false)	False
WHERE false AND (true OR true)	False
WHERE false AND (true OR false)	False
WHERE false AND (false OR true)	False
WHERE false AND (false OR false)	False

Как видите, чем больше условий в блоке where, тем больше комбинаций должен рассмотреть сервер. В данном случае конечный результат true обеспечивают только три из восьми комбинаций.

Оператор not

Надеюсь, предыдущий пример с тремя условиями прост и понятен. Но рассмотрим следующий вариант:

```
WHERE end_date IS NULL
  AND NOT (title = 'Teller' OR start_date < '2003-01-01')
```

Заметили отличие от предыдущего примера? После оператора and во второй строке появился оператор not. Теперь, вместо поиска неуволенных сотрудников, или являющихся операционистами, или начавших работать в банке до 2003 года, выбираются неуволенные сотрудники, которые или не являются операционистами, или начали работу в банке в 2003 и позже. В табл. 4.3 показаны возможные результаты выполнения этого примера.

Таблица 4.3. Результаты вычисления выражений с тремя условиями, разделенными операторами and, or и not

Промежуточный результат	Конечный результат
WHERE true AND NOT (true OR true)	False
WHERE true AND NOT (true OR false)	False
WHERE true AND NOT (false OR true)	False
WHERE true AND NOT (false OR false)	True
WHERE false AND NOT (true OR true)	False
WHERE false AND NOT (true OR false)	False

Промежуточный результат	Конечный результат
`WHERE false AND NOT (false OR true)`	`False`
`WHERE false AND NOT (false OR false)`	`False`

Сервер легко обработает такое выражение, а человеку оценить блок, включающий оператор `not`, обычно трудно. Вот почему он используется нечасто. В данном случае блок `where` можно изменить и записать без оператора `not` следующим образом:

```
WHERE end_date IS NULL
  AND title != 'Teller' AND start_date >= '2003-01-01'
```

Серверу наверняка все равно, а человеку, пожалуй, проще понять этот вариант блока `where`.

Создание условия

Теперь, посмотрев, как сервер вычисляет несколько условий, давайте вернемся назад и посмотрим на то, из чего состоит отдельное условие. Его образуют одно или более *выражений*, попарно объединенных одним или более операторами. Выражением может быть любое из следующего:

- Число
- Столбец таблицы или представления
- Строковый литерал, например `'Teller'`
- Встроенная функция, например `CONCAT('Learning', ' ', 'SQL')`
- Подзапрос
- Список выражений, например `('Teller', 'Head Teller', 'Operations Manager')`

К операторам, используемым в условиях, относятся:

- Операторы сравнения, такие как `=`, `!=`, `<`, `>`, `<>`, `LIKE`, `IN` и `BETWEEN`
- Арифметические операторы, такие как `+`, `-`, `*` и `/`

В следующих разделах показано, как путем сочетания этих выражений и операторов можно создавать различные типы условий.

Типы условий

Есть множество способов отфильтровать ненужные данные. Чтобы включить или исключить те или иные данные, можно вести поиск определенных значений, наборов значений или диапазонов значений. При работе со строковыми данными можно использовать различные методики поиска по шаблону для выявления частичного соответствия. Следующие четыре раздела подробно описывают каждый из этих типов условий.

Условия равенства

Многие из создаваемых или существующих условий фильтрации имеют форму *'столбец = выражение'*:

```
title = 'Teller'
fed_id = '111-11-1111'
amount = 375.25
dept_id = (SELECT dept_id FROM department WHERE name = 'Loans')
```

Такие условия называются *условиями равенства*, потому что они проверяют равенство одного выражения другому. В первых трех примерах столбец сравнивается с литералом (две строки и число), а в четвертом столбец сравнивается со значением, возвращаемым подзапросом. Следующий запрос использует два условия равенства, одно в блоке `on` (условие соединения) и второе в блоке `where` (условие фильтрации):

```
mysql> SELECT pt.name product_type, p.name product
-> FROM product p INNER JOIN product_type pt
->   ON p.product_type_cd = pt.product_type_cd
-> WHERE pt.name = 'Customer Accounts';
+------------------+-----------------------+
| product_type     | product               |
+------------------+-----------------------+
| Customer Accounts | certificate of deposit |
| Customer Accounts | checking account      |
| Customer Accounts | money market account  |
| Customer Accounts | savings account       |
+------------------+-----------------------+
4 rows in set (0.08 sec)
```

По этому запросу выбираются все счета, являющиеся лицевыми счетами (customer accounts).

Условия неравенства

Другой достаточно распространенный тип условия – *условие неравенства*, которое определяет, что два выражения *не* равны. Вот предыдущий запрос, в котором условием фильтрации в блоке `where` является условие неравенства:

```
mysql> SELECT pt.name product_type, p.name product
-> FROM product p INNER JOIN product_type pt
->   ON p.product_type_cd = pt.product_type_cd
-> WHERE pt.name != 'Customer Accounts';
+------------------------------+------------------------+
| product_type                 | product                |
+------------------------------+------------------------+
| Individual and Business Loans | auto loan              |
| Individual and Business Loans | business line of credit |
| Individual and Business Loans | home mortgage          |
| Individual and Business Loans | small business loan    |
```

```
+------------------------------+------------------------+
4 rows in set (0.00 sec)
```

В результате этого запроса выводятся все счета, *не* являющиеся лицевыми счетами. В условиях неравенства можно использовать оператор != или <>.

Изменение данных с помощью условий равенства

Условия равенства/неравенства обычно используются при изменении данных. Например, в банке принято уничтожать строки старых счетов раз в год. Задача состоит в удалении из таблицы account строк с данными о счетах, закрытых в 1999 году. Вот одно из возможных решений:

```
DELETE FROM account
WHERE status = 'CLOSED' AND YEAR(close_date) = 1999;
```

Это выражение включает два условия равенства: одно для выбора только закрытых счетов, а другое – чтобы проверить, были ли эти счета закрыты в 1999 году.

Создавая примеры выражений удаления и обновления, я попытаюсь писать каждое выражение таким образом, чтобы ни одна строка не изменялась. Тогда при их выполнении данные останутся не измененными, и получаемый вами результат выражений select всегда будет соответствовать приведенному в книге.

Поскольку сеансы MySQL по умолчанию находятся в режиме автоматической фиксации (см. главу 12), нельзя откатить (отменить) изменения, внесенные в данные примера, если одно из выражений изменило их. Конечно, вы можете делать с данными, что угодно, даже полностью очистить их и повторно запустить предоставленные мною сценарии, но я постараюсь сохранять их нетронутыми.

Условия вхождения в диапазон

Кроме проверки равенства (или неравенства) одного выражения другому, можно создать условия, проверяющие, попадает ли выражение в определенный диапазон. Этот тип условия широко используется при работе с числовыми или временными данными. Рассмотрим следующий запрос:

```
mysql> SELECT emp_id, fname, lname, start_date
-> FROM employee
-> WHERE start_date < '2003-01-01';
+--------+---------+-----------+------------+
| emp_id | fname   | lname     | start_date |
+--------+---------+-----------+------------+
|      1 | Michael | Smith     | 2001-06-22 |
|      2 | Susan   | Barker    | 2002-09-12 |
|      3 | Robert  | Tyler     | 2000-02-09 |
|      4 | Susan   | Hawthorne | 2002-04-24 |
```

```
|      8 | Sarah   | Parker    | 2002-12-02 |
|      9 | Jane    | Grossman  | 2002-05-03 |
|     10 | Paula   | Roberts   | 2002-07-27 |
|     11 | Thomas  | Ziegler   | 2000-10-23 |
|     13 | John    | Blake     | 2000-05-11 |
|     14 | Cindy   | Mason     | 2002-08-09 |
|     16 | Theresa | Markham   | 2001-03-15 |
|     17 | Beth    | Fowler    | 2002-06-29 |
|     18 | Rick    | Tulman    | 2002-12-12 |
+--------+---------+-----------+------------+
13 rows in set (0.01 sec)
```

Этот запрос выявляет всех сотрудников, нанятых до 2003 года. Кроме верхней границы даты начала работы, можно задать и нижнюю границу:

```
mysql> SELECT emp_id, fname, lname, start_date
-> FROM employee
-> WHERE start_date < '2003-01-01'
->   AND start_date >= '2001-01-01';
+--------+---------+-----------+------------+
| emp_id | fname   | lname     | start_date |
+--------+---------+-----------+------------+
|      1 | Michael | Smith     | 2001-06-22 |
|      2 | Susan   | Barker    | 2002-09-12 |
|      4 | Susan   | Hawthorne | 2002-04-24 |
|      8 | Sarah   | Parker    | 2002-12-02 |
|      9 | Jane    | Grossman  | 2002-05-03 |
|     10 | Paula   | Roberts   | 2002-07-27 |
|     14 | Cindy   | Mason     | 2002-08-09 |
|     16 | Theresa | Markham   | 2001-03-15 |
|     17 | Beth    | Fowler    | 2002-06-29 |
|     18 | Rick    | Tulman    | 2002-12-12 |
+--------+---------+-----------+------------+
10 rows in set (0.01 sec)
```

Эта версия запроса выбирает всех сотрудников, нанятых с 2001 по 2002 год.

Оператор between

Если имеются верхняя и нижняя границы диапазона, вместо двух разных условий можно использовать одно, использующее оператор `between` (между):

```
mysql> SELECT emp_id, fname, lname, start_date
-> FROM employee
-> WHERE start_date BETWEEN '2001-01-01' AND '2003-01-01';
+--------+---------+-----------+------------+
| emp_id | fname   | lname     | start_date |
+--------+---------+-----------+------------+
|      1 | Michael | Smith     | 2001-06-22 |
```

```
|      2 | Susan   | Barker    | 2002-09-12 |
|      4 | Susan   | Hawthorne | 2002-04-24 |
|      8 | Sarah   | Parker    | 2002-12-02 |
|      9 | Jane    | Grossman  | 2002-05-03 |
|     10 | Paula   | Roberts   | 2002-07-27 |
|     14 | Cindy   | Mason     | 2002-08-09 |
|     16 | Theresa | Markham   | 2001-03-15 |
|     17 | Beth    | Fowler    | 2002-06-29 |
|     18 | Rick    | Tulman    | 2002-12-12 |
+--------+---------+-----------+------------+
10 rows in set (0.05 sec)
```

При работе с оператором `between` необходимо помнить пару правил. Первой (после ключевого слова `between`) всегда должна задаваться нижняя граница диапазона, а потом (после `and`) верхняя граница. Вот что происходит, если задать первой верхнюю границу:

```
mysql> SELECT emp_id, fname, lname, start_date
-> FROM employee
-> WHERE start_date BETWEEN '2003-01-01' AND '2001-01-01';
Empty set (0.00 sec)
```

Как видите, не возвращено никаких данных. Это объясняется тем, что фактически из одного заданного условия сервер генерирует два, используя операторы `<=` и `>=`:

```
mysql> SELECT emp_id, fname, lname, start_date
-> FROM employee
-> WHERE start_date >= '2003-01-01'
->   AND start_date <= '2001-01-01';
Empty set (0.00 sec)
```

Поскольку такая дата – одновременно позднее 1 января 2003 года и раньше 1 января 2001 года – не существует, по запросу возвращается пустой набор. Далее следует отметить второй подводный камень при использовании оператора `between`: необходимо помнить, что верхняя и нижняя границы *включаются* в диапазон. В данном случае я хотел, чтобы нижней границей была дата `2001-01-01`, а верхней – `2002-12-31`, а не `2003-01-01`. Даже несмотря на то, что, скорее всего, никто из сотрудников банка не начал работать в первый же день Нового 2003 года, лучше задавать именно то, что требуется.

Как и для дат, можно создавать условия, определяющие диапазон для чисел. Числовые диапазоны довольно просты для понимания, как видно из следующего примера:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
-> FROM account
-> WHERE avail_balance BETWEEN 3000 AND 5000;
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
```

```
|          3 | CD          |        1 |       3000.00 |
|         14 | CD          |        7 |       5000.00 |
|         15 | CHK         |        8 |       3487.19 |
+------------+-------------+----------+---------------+
3 rows in set (0.03 sec)
```

Выбираются все счета, доступный остаток которых составляет от 3000 до 5000 долларов. Еще раз подчеркну, что первой задается нижняя граница.

Строковые диапазоны

Диапазоны дат и чисел легко представить, но можно также создавать условия для поиска диапазона строк, проиллюстрировать которые чуть сложнее. Например, требуется найти клиентов, для которых в определенный диапазон попадает номер социальной страховки. Формат номера социальной страховки – `'XXX-XX-XXXX'`, где `X` – число от `0` до `9`. Требуется найти всех клиентов, номер социальной страховки которых находится между `'500-00-0000'` и `'999-99-9999'`. Вот как может выглядеть такое выражение:

```
mysql> SELECT cust_id, fed_id
-> FROM customer
-> WHERE cust_type_cd = 'I'
->   AND fed_id BETWEEN '500-00-0000' AND '999-99-9999';
+---------+-------------+
| cust_id | fed_id      |
+---------+-------------+
|       5 | 555-55-5555 |
|       6 | 666-66-6666 |
|       7 | 777-77-7777 |
|       8 | 888-88-8888 |
|       9 | 999-99-9999 |
+---------+-------------+
5 rows in set (0.01 sec)
```

Для работы со строковыми диапазонами необходимо знать порядок символов в наборе символов (порядок, в котором сортируются символы в наборе символов, называется *сопоставлением* (*collation*)).

Условия членства

В некоторых случаях выражение ограничивается не одним значением или диапазоном значений, а конечным набором (set) значений. Например, требуется выбрать все счета, кодом типа которых является `'CHK'`, `'SAV'`, `'CD'` или `'MM'`:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
-> FROM account
-> WHERE product_cd = 'CHK' OR product_cd = 'SAV'
->   OR product_cd = 'CD' OR product_cd = 'MM';
```

```
+------------+------------+---------+--------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+--------------+
|          1 | CHK        |       1 |       1057.75 |
|          2 | SAV        |       1 |        500.00 |
|          3 | CD         |       1 |       3000.00 |
|          4 | CHK        |       2 |       2258.02 |
|          5 | SAV        |       2 |        200.00 |
|          6 | CHK        |       3 |       1057.75 |
|          7 | MM         |       3 |       2212.50 |
|          8 | CHK        |       4 |        534.12 |
|          9 | SAV        |       4 |        767.77 |
|         10 | MM         |       4 |       5487.09 |
|         11 | CHK        |       5 |       2237.97 |
|         12 | CHK        |       6 |        122.37 |
|         13 | CD         |       6 |      10000.00 |
|         14 | CD         |       7 |       5000.00 |
|         15 | CHK        |       8 |       3487.19 |
|         16 | SAV        |       8 |        387.99 |
|         17 | CHK        |       9 |        125.67 |
|         18 | MM         |       9 |       9345.55 |
|         19 | CD         |       9 |       1500.00 |
|         20 | CHK        |      10 |      23575.12 |
|         23 | CHK        |      12 |      38552.05 |
+------------+------------+---------+--------------+
21 rows in set (0.02 sec)
```

На создание этого блока `where` (всего четыре условия, разделенных операторами `or`) ушло не слишком много сил и времени. А представьте, если бы набор выражений содержал 10 или 20 элементов? В таких ситуациях можно использовать оператор `in`:

```
SELECT account_id, product_cd, cust_id, avail_balance
FROM account
WHERE product_cd IN ('CHK','SAV','CD','MM');
```

При использовании оператора `in` записывается единственное условие, сколько бы у вас ни было выражений.

Подзапросы

Можно самостоятельно создать набор выражений, например `('CHK', 'SAV','CD','MM')`, но сделать это можно и с помощью подзапроса. Например, у всех четырех типов счетов, используемых в предыдущем запросе, столбец `product_type_cd` имеет значение `'ACCOUNT'`. В следующей версии запроса для извлечения четырех кодов типов счетов вместо явного указания их имен используется подзапрос к таблице `product`:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
->   FROM account
->   WHERE product_cd IN (SELECT product_cd FROM product
->     WHERE product_type_cd = 'ACCOUNT');
```

```
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|          1 | CHK        |       1 |       1057.75 |
|          2 | SAV        |       1 |        500.00 |
|          3 | CD         |       1 |       3000.00 |
|          4 | CHK        |       2 |       2258.02 |
|          5 | SAV        |       2 |        200.00 |
|          6 | CHK        |       3 |       1057.75 |
|          7 | MM         |       3 |       2212.50 |
|          8 | CHK        |       4 |        534.12 |
|          9 | SAV        |       4 |        767.77 |
|         10 | MM         |       4 |       5487.09 |
|         11 | CHK        |       5 |       2237.97 |
|         12 | CHK        |       6 |        122.37 |
|         13 | CD         |       6 |      10000.00 |
|         14 | CD         |       7 |       5000.00 |
|         15 | CHK        |       8 |       3487.19 |
|         16 | SAV        |       8 |        387.99 |
|         17 | CHK        |       9 |        125.67 |
|         18 | MM         |       9 |       9345.55 |
|         19 | CD         |       9 |       1500.00 |
|         20 | CHK        |      10 |      23575.12 |
|         23 | CHK        |      12 |      38552.05 |
+------------+------------+---------+---------------+
21 rows in set (0.03 sec)
```

Подзапрос возвращает набор из четырех значений, а основной запрос проверяет, соответствует ли значение столбца `product_cd` значениям, возвращенным подзапросом.

Оператор not in

Иногда требуется проверить, присутствует ли определенное выражение в наборе выражений, а иногда нужно удостовериться в его *отсутствии*. В таких ситуациях можно использовать оператор `not in` (нет в):

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
->   FROM account
->   WHERE product_cd NOT IN ('CHK','SAV','CD','MM');
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|         21 | BUS        |      10 |          0.00 |
|         22 | BUS        |      11 |       9345.55 |
|         24 | SBL        |      13 |      50000.00 |
+------------+------------+---------+---------------+
3 rows in set (0.02 sec)
```

Этот запрос находит все счета, *не* являющиеся текущими, депозитными, депозитными сертификатами или депозитными счетами денежного рынка.

Условия соответствия

До сих пор были рассмотрены условия, выделяющие определенную строку, диапазон строк или набор строк. Еще один тип условий касается частичного соответствия строк. Например, требуется найти всех сотрудников, фамилия которых начинается с «T». Получить первую букву значения столбца `lname` можно с помощью встроенной функции:

```
mysql> SELECT emp_id, fname, lname
-> FROM employee
-> WHERE LEFT(lname, 1) = 'T';
+--------+--------+--------+
| emp_id | fname  | lname  |
+--------+--------+--------+
|      3 | Robert | Tyler  |
|      7 | Chris  | Tucker |
|     18 | Rick   | Tulman |
+--------+--------+--------+
3 rows in set (0.01 sec)
```

Хотя встроенная функция `left()` выполняет то, что требуется, она не обеспечивает особой гибкости. Вместо нее можно в выражениях поиска можно использовать символы маски, как показано в следующем разделе.

Символы маски

При поиске частичных соответствий строк интерес могут представлять:

- Строки, начинающиеся/заканчивающиеся определенным символом
- Строки, начинающиеся/заканчивающиеся подстрокой
- Строки, содержащие определенный символ в любом месте строки
- Строки, содержащие подстроку в любом месте строки
- Строки определенного формата, независимо от входящих в них отдельных символов

С помощью символов маски, представленных в табл. 4.4, можно построить выражения для поиска этих и многих других частичных строковых соответствий.

Таблица 4.4. Символы маски

Символ маски	Соответствие
_	Точно один символ
%	Любое число символов (в том числе ни одного)

Символ подчеркивания замещает один символ, а символ процента может замещать разное количество символов. При построении условий, использующих выражения поиска, применяется оператор `like` (как):

```
mysql> SELECT lname
```

```
-> FROM employee
-> WHERE lname LIKE '_a%e%';
+-----------+
| lname     |
+-----------+
| Barker    |
| Hawthorne |
| Parker    |
| Jameson   |
+-----------+
4 rows in set (0.00 sec)
```

Выражение поиска в предыдущем примере определяет строки, содержащие «a» во второй позиции, за которым следует «e» в любом другом месте строки (включая последний символ). В табл. 4.5 показано еще несколько выражений поиска и их интерпретации.

Таблица 4.5. Примеры выражений поиска

Выражение поиска	Интерпретация
`F%`	Строки, начинающиеся с «`F`»
`%t`	Строки, заканчивающиеся на «`t`»
`%bas%`	Строки, содержащие подстроку «`bas`»
`__t_`	Строки, состоящие из четырех символов с «`t`» в третьей позиции
`___-__-____`	Строки из 11 символов, где четвертый и седьмой символы – дефисы

Последний пример из табл. 4.5 можно использовать для поиска клиентов, федеральный ID которых соответствует формату, используемому для номеров социальной страховки:

```
mysql> SELECT cust_id, fed_id
-> FROM customer
-> WHERE fed_id LIKE '___-__-____';
+---------+-------------+
| cust_id | fed_id      |
+---------+-------------+
|       1 | 111-11-1111 |
|       2 | 222-22-2222 |
|       3 | 333-33-3333 |
|       4 | 444-44-4444 |
|       5 | 555-55-5555 |
|       6 | 666-66-6666 |
|       7 | 777-77-7777 |
|       8 | 888-88-8888 |
|       9 | 999-99-9999 |
+---------+-------------+
9 rows in set (0.02 sec)
```

Символы маски хороши для простых выражений поиска. Если требуется несколько более сложный поиск, можно использовать несколько выражений поиска, как показано в следующем примере:

```
mysql> SELECT emp_id, fname, lname
-> FROM employee
-> WHERE lname LIKE 'F%' OR lname LIKE 'G%';
+--------+-------+----------+
| emp_id | fname | lname    |
+--------+-------+----------+
|      5 | John  | Gooding  |
|      6 | Helen | Fleming  |
|      9 | Jane  | Grossman |
|     17 | Beth  | Fowler   |
+--------+-------+----------+
4 rows in set (0.00 sec)
```

Этот запрос находит всех сотрудников, фамилия которых начинается с «F» или «G».

Регулярные выражения

Если символы маски не обеспечивают достаточной гибкости, для построения выражений поиска можно использовать регулярные выражения. По существу, регулярные выражения – это мощнейшие выражения поиска. Регулярные выражения хорошо знакомы разработчикам на таких языках программирования, как Perl. Если вам не доводилось использовать их, обратитесь к книге Джеффри Фридла (Jeffrey Friedl) «Mastering Regular Expressions» (O'Reilly). Это слишком объемная тема, чтобы пытаться охватить ее в данной книге.

Вот как выглядел бы предыдущий запрос (найти всех сотрудников с фамилиями, начинающимися с «F» или «G») с использованием реализации регулярных выражений MySQL:

```
mysql> SELECT emp_id, fname, lname
-> FROM employee
-> WHERE lname REGEXP '^[FG]';
+--------+-------+----------+
| emp_id | fname | lname    |
+--------+-------+----------+
|      5 | John  | Gooding  |
|      6 | Helen | Fleming  |
|      9 | Jane  | Grossman |
|     17 | Beth  | Fowler   |
+--------+-------+----------+
4 rows in set (0.00 sec)
```

Оператор `regexp` принимает регулярное выражение (в данном примере `'^[FG]'`) и применяет его к выражению, находящемуся в левой части условия (столбец `lname`). Теперь, с регулярным выражением, запрос содержит всего одно условие, а не два, как это было при использовании символов маски.

Oracle Database 10*g* и SQL Server 2000 тоже поддерживают регулярные выражения. При работе с Oracle Database 10*g* используется функция `regexp_like`, а не оператор `regexp`, показанный в предыдущем примере. А SQL Server допускает использование регулярных выражений с оператором `like`.

NULL: это слово из четырех букв...

Я, сколько мог, оттягивал этот момент, но он настал: пора обратиться к теме, которую обычно встречают с опаской, неуверенностью и трепетом, – значение `null`. `Null` – это отсутствие значения. Например, пока сотрудник не уволен, в его столбце `end_date` таблицы `employee` должно быть записано `null`. В данной ситуации нет значения, которое могло бы быть помещено в столбец `end_date` и имело бы смысл. Однако `null` – коварный тип, поскольку имеет несколько оттенков:

Неприменимый

Например, столбец с ID сотрудника для транзакции, которая выполняется с банкоматом.

Значение еще не известно

Например, если в момент создания строки клиента федеральный ID неизвестен.

Значение не определено

Например, если создается счет для продукта, который еще не был добавлен в БД.

Некоторые теоретики считают, что для каждой из этих (и других) ситуаций следовало бы использовать разные выражения, но по мнению большинства практикующих специалистов при нескольких значениях `null` путаницы было бы гораздо больше.

При работе с `null` необходимо помнить:

- Выражение может *быть* нулевым (`null`), но оно никогда не может быть *равным* нулю.
- Два `null` никогда не равны друг другу.

Проверить выражение на значение `null` можно с помощью оператора `is null`, как показано в следующем примере:

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
-> FROM employee
-> WHERE superior_emp_id IS NULL;
+--------+---------+-------+-----------------+
| emp_id | fname   | lname | superior_emp_id |
+--------+---------+-------+-----------------+
|      1 | Michael | Smith |            NULL |
+--------+---------+-------+-----------------+
1 row in set (0.00 sec)
```

Этот запрос возвращает всех сотрудников, у которых нет начальника (superior). Вот тот же запрос, но вместо `is null` используется `= null`:

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
-> FROM employee
-> WHERE superior_emp_id = NULL;
Empty set (0.01 sec)
```

Как видите, запрос подвергается синтаксическому анализу и выполняется, но не возвращает ни одной строки. Это общая ошибка неопытных SQL-программистов. Сервер БД не предупредит о ней, поэтому при создании условий проверки на `null` следует соблюдать осторожность.

Проверить наличие значения в столбце можно с помощью оператора `is not null`:

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
-> FROM employee
-> WHERE superior_emp_id IS NOT NULL;
+--------+----------+-----------+-----------------+
| emp_id | fname    | lname     | superior_emp_id |
+--------+----------+-----------+-----------------+
|      2 | Susan    | Barker    |               1 |
|      3 | Robert   | Tyler     |               1 |
|      4 | Susan    | Hawthorne |               3 |
|      5 | John     | Gooding   |               4 |
|      6 | Helen    | Fleming   |               4 |
|      7 | Chris    | Tucker    |               6 |
|      8 | Sarah    | Parker    |               6 |
|      9 | Jane     | Grossman  |               6 |
|     10 | Paula    | Roberts   |               4 |
|     11 | Thomas   | Ziegler   |              10 |
|     12 | Samantha | Jameson   |              10 |
|     13 | John     | Blake     |               4 |
|     14 | Cindy    | Mason     |              13 |
|     15 | Frank    | Portman   |              13 |
|     16 | Theresa  | Markham   |               4 |
|     17 | Beth     | Fowler    |              16 |
|     18 | Rick     | Tulman    |              16 |
+--------+----------+-----------+-----------------+
17 rows in set (0.01 sec)
```

Эта версия запроса возвращает остальных 17 сотрудников, у которых, в отличие от Майкла Смита (Michael Smith), есть начальник.

Прежде чем на время отложить рассмотрение `null`, было бы полезным отметить еще одну потенциальную ловушку. Предположим, требуется идентифицировать всех сотрудников, не подчиняющихся Хелен Флеминг (Helen Fleming), ID которой равен 6. Наверняка первым порывом будет сделать следующее:

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
-> FROM employee
-> WHERE superior_emp_id != 6;
```

```
+--------+----------+-----------+-----------------+
| emp_id | fname    | lname     | superior_emp_id |
+--------+----------+-----------+-----------------+
|      2 | Susan    | Barker    |               1 |
|      3 | Robert   | Tyler     |               1 |
|      4 | Susan    | Hawthorne |               3 |
|      5 | John     | Gooding   |               4 |
|      6 | Helen    | Fleming   |               4 |
|     10 | Paula    | Roberts   |               4 |
|     11 | Thomas   | Ziegler   |              10 |
|     12 | Samantha | Jameson   |              10 |
|     13 | John     | Blake     |               4 |
|     14 | Cindy    | Mason     |              13 |
|     15 | Frank    | Portman   |              13 |
|     16 | Theresa  | Markham   |               4 |
|     17 | Beth     | Fowler    |              16 |
|     18 | Rick     | Tulman    |              16 |
+--------+----------+-----------+-----------------+
14 rows in set (0.01 sec)
```

Действительно, эти 14 сотрудников не работают под руководством Хелен Флеминг, но если внимательно посмотреть на данные, можно заметить, что здесь пропущен один сотрудник, также не являющийся подчиненным Хелен. Это Майкл Смит, и в его столбце `superior_emp_id` стоит `null` (потому что он «большая шишка»). Поэтому, чтобы правильно ответить на вопрос, необходимо учитывать вероятность того, что для некоторых строк столбец `superior_emp_id` может иметь значение `null`.

```
mysql> SELECT emp_id, fname, lname, superior_emp_id
-> FROM employee
-> WHERE superior_emp_id != 6 OR superior_emp_id IS NULL;
+--------+----------+-----------+-----------------+
| emp_id | fname    | lname     | superior_emp_id |
+--------+----------+-----------+-----------------+
|      1 | Michael  | Smith     |            NULL |
|      2 | Susan    | Barker    |               1 |
|      3 | Robert   | Tyler     |               1 |
|      4 | Susan    | Hawthorne |               3 |
|      5 | John     | Gooding   |               4 |
|      6 | Helen    | Fleming   |               4 |
|     10 | Paula    | Roberts   |               4 |
|     11 | Thomas   | Ziegler   |              10 |
|     12 | Samantha | Jameson   |              10 |
|     13 | John     | Blake     |               4 |
|     14 | Cindy    | Mason     |              13 |
|     15 | Frank    | Portman   |              13 |
|     16 | Theresa  | Markham   |               4 |
|     17 | Beth     | Fowler    |              16 |
|     18 | Rick     | Tulman    |              16 |
+--------+----------+-----------+-----------------+
15 rows in set (0.01 sec)
```

Теперь результирующий набор включает всех 15 сотрудников, не подчиняющихся Хелен. При работе с малознакомой базой данных не помешает выяснить, какие столбцы таблицы могут содержать `null`; это поможет вам создавать правильные условия фильтрации, чтобы данные не смогли утекать сквозь пальцы.

Упражнения

Следующие упражнения проверят ваше понимание условий фильтрации. Решения ищите в приложении C.

В первых двух упражнениях используются следующие данные о транзакциях:

Txn_id	Txn_date	Account_id	Txn_type_cd	Amount
1	2005-02-22	101	CDT	1000.00
2	2005-02-23	102	CDT	525.75
3	2005-02-24	101	DBT	100.00
4	2005-02-24	103	CDT	55
5	2005-02-25	101	DBT	50
6	2005-02-25	103	DBT	25
7	2005-02-25	102	CDT	125.37
8	2005-02-26	103	DBT	10
9	2005-02-27	101	CDT	75

4.1

Какие ID транзакций возвращают следующие условия фильтрации?

```
txn_date < '2005-02-26' AND (txn_type_cd = 'DBT' OR amount > 100)
```

4.2

Какие ID транзакций возвращают следующие условия фильтрации?

```
account_id IN (101,103) AND NOT (txn_type_cd = 'DBT' OR amount > 100)
```

4.3

Создайте запрос, выбирающий все счета, открытые в 2002 году.

4.4

Создайте запрос, выбирающий всех клиентов-физических лиц, второй буквой фамилии которых является буква `'a'` и есть `'e'` в любой позиции после `'a'`.

5

Запрос к нескольким таблицам

Поскольку реляционные БД предполагают расположение независимых сущностей в разных таблицах, необходим механизм сведения нескольких таблиц воедино в одном запросе. Этот механизм известен как *соединение* (*join*), и данная глава посвящена самому простому и наиболее распространенному соединению – *внутреннему соединению* (*inner join*). Все разнообразные типы соединений представлены в главе 10.

Что такое соединение?

Запросы к одной таблице, конечно, не редкость, но большинство запросов обращены к двум, трем или даже более таблицам. Для иллюстрации давайте рассмотрим описания таблиц `employee` и `department` и затем определим запрос, извлекающий данные из обеих:

```
mysql> DESC employee;
+--------------------+----------------------+------+-----+------------+
| Field              | Type                 | Null | Key | Default    |
+--------------------+----------------------+------+-----+------------+
| emp_id             | smallint(5) unsigned |      | PRI | NULL       |
| fname              | varchar(20)          |      |     |            |
| lname              | varchar(20)          |      |     |            |
| start_date         | date                 |      |     | 0000-00-00 |
| end_date           | date                 | YES  |     | NULL       |
| superior_emp_id    | smallint(5) unsigned | YES  | MUL | NULL       |
| dept_id            | smallint(5) unsigned | YES  | MUL | NULL       |
| title              | varchar(20)          | YES  |     | NULL       |
| assigned_branch_id | smallint(5) unsigned | YES  | MUL | NULL       |
+--------------------+----------------------+------+-----+------------+
9 rows in set (0.11 sec)

mysql> DESC department;
+---------+----------------------+------+-----+---------+
```

```
| Field   | Type                 | Null | Key | Default |
+---------+----------------------+------+-----+---------+
| dept_id | smallint(5) unsigned |      | PRI | NULL    |
| name    | varchar(20)          |      |     |         |
+---------+----------------------+------+-----+---------+
2 rows in set (0.03 sec)
```

Скажем, требуется выбрать имя и фамилию каждого сотрудника, а также название отдела, в котором он работает. Поэтому запрос должен будет извлекать столбцы `employee.fname`, `employee.lname` и `department.name`. Но как можно получить данные двух таблиц одним запросом? Ответ кроется в столбце `employee.dept_id`, в котором хранится ID отдела каждого сотрудника (более формально, столбец `employee.dept_id` является *внешним ключом*, ссылающимся на таблицу `department`). Запрос, который вскоре будет представлен, указывает серверу использовать столбец `employee.dept_id` как *мост* между таблицами `employee` и `department`, обеспечивая таким образом возможность включения столбцов обеих таблиц в результирующий набор запроса. Такой тип операции называется соединением.

Декартово произведение

Начнем с самого простого: поместим таблицы `employee` и `department` в блок `from` запроса и посмотрим, что произойдет. Вот запрос, выбирающий имена и фамилии сотрудников, а также название отдела. Здесь в блоке `from` указаны обе таблицы, разделенные ключевым словом `join`:

```
mysql> SELECT e.fname, e.lname, d.name
    -> FROM employee e JOIN department d;
+----------+-----------+----------------+
| fname    | lname     | name           |
+----------+-----------+----------------+
| Michael  | Smith     | Operations     |
| Susan    | Barker    | Operations     |
| Robert   | Tyler     | Operations     |
| Susan    | Hawthorne | Operations     |
| John     | Gooding   | Operations     |
| Helen    | Fleming   | Operations     |
| Chris    | Tucker    | Operations     |
| Sarah    | Parker    | Operations     |
| Jane     | Grossman  | Operations     |
| Paula    | Roberts   | Operations     |
| Thomas   | Ziegler   | Operations     |
| Samantha | Jameson   | Operations     |
| John     | Blake     | Operations     |
| Cindy    | Mason     | Operations     |
| Frank    | Portman   | Operations     |
| Theresa  | Markham   | Operations     |
| Beth     | Fowler    | Operations     |
| Rick     | Tulman    | Operations     |
```

```
| Michael  | Smith     | Loans          |
| Susan    | Barker    | Loans          |
| Robert   | Tyler     | Loans          |
| Susan    | Hawthorne | Loans          |
| John     | Gooding   | Loans          |
| Helen    | Fleming   | Loans          |
| Chris    | Tucker    | Loans          |
| Sarah    | Parker    | Loans          |
| Jane     | Grossman  | Loans          |
| Paula    | Roberts   | Loans          |
| Thomas   | Ziegler   | Loans          |
| Samantha | Jameson   | Loans          |
| John     | Blake     | Loans          |
| Cindy    | Mason     | Loans          |
| Frank    | Portman   | Loans          |
| Theresa  | Markham   | Loans          |
| Beth     | Fowler    | Loans          |
| Rick     | Tulman    | Loans          |
| Michael  | Smith     | Administration |
| Susan    | Barker    | Administration |
| Robert   | Tyler     | Administration |
| Susan    | Hawthorne | Administration |
| John     | Gooding   | Administration |
| Helen    | Fleming   | Administration |
| Chris    | Tucker    | Administration |
| Sarah    | Parker    | Administration |
| Jane     | Grossman  | Administration |
| Paula    | Roberts   | Administration |
| Thomas   | Ziegler   | Administration |
| Samantha | Jameson   | Administration |
| John     | Blake     | Administration |
| Cindy    | Mason     | Administration |
| Frank    | Portman   | Administration |
| Theresa  | Markham   | Administration |
| Beth     | Fowler    | Administration |
| Rick     | Tulman    | Administration |
+----------+-----------+----------------+
54 rows in set (0.00 sec)
```

Хм... у нас только 18 сотрудников и 3 разных отдела. Но как же получилось, что в результирующем наборе оказалось 54 строки? Приглядевшись, можно заметить, что каждый из 18 сотрудников встречается трижды. При этом все его данные идентичны, кроме названия отдела. Поскольку запрос не определил, *как* должны быть соединены эти две таблицы, сервер БД сгенерировал *Декартово произведение*, т. е. *все возможные* перестановки двух таблиц (18 сотрудников умножить на 3 отдела получается 54 перестановки). Такой тип соединения называют *перекрестным соединением* (*cross join*). Его редко используют (намеренно, по крайней мере). Перекрестные соединения – один из типов соединений, которые будут изучаться в главе 10.

Внутренние соединения

Чтобы изменить предыдущий запрос и получить результирующий набор, включающий только 18 строк (по одной для каждого сотрудника), понадобится описать взаимосвязь двух таблиц. Я уже показал, что связью между двумя таблицами служит столбец `employee.dept_id`, осталось только добавить эту информацию в подблок `on` блока `from`:

```
mysql> SELECT e.fname, e.lname, d.name
    -> FROM employee e JOIN department d
    ->   ON e.dept_id = d.dept_id;
+----------+-----------+----------------+
| fname    | lname     | name           |
+----------+-----------+----------------+
| Susan    | Hawthorne | Operations     |
| Helen    | Fleming   | Operations     |
| Chris    | Tucker    | Operations     |
| Sarah    | Parker    | Operations     |
| Jane     | Grossman  | Operations     |
| Paula    | Roberts   | Operations     |
| Thomas   | Ziegler   | Operations     |
| Samantha | Jameson   | Operations     |
| John     | Blake     | Operations     |
| Cindy    | Mason     | Operations     |
| Frank    | Portman   | Operations     |
| Theresa  | Markham   | Operations     |
| Beth     | Fowler    | Operations     |
| Rick     | Tulman    | Operations     |
| John     | Gooding   | Loans          |
| Michael  | Smith     | Administration |
| Susan    | Barker    | Administration |
| Robert   | Tyler     | Administration |
+----------+-----------+----------------+
18 rows in set (0.00 sec)
```

Теперь благодаря добавлению подблока `on`, предписывающего серверу соединять таблицы `employee` и `department`, прослеживая связь от одной таблицы к другой по столбцу `dept_id`, имеем вместо 54 строк ожидаемые 18. Например, строка Сьюзен Хоторн (Susan Hawthorne) в таблице `employee` в столбце `dept_id` содержит `1` (в примере не показано). Сервер использует это значение для поиска строки в таблице `department`, столбец `dept_id` которой содержит `1`, и извлекает значение `'Operations'` из столбца `name` этой строки.

Если определенное значение столбца `dept_id` присутствует в одной таблице, но его *нет* в другой, соединение строк не происходит, и они не включаются в результирующий набор. Такой тип соединения называют *внутренним соединением* (*inner join*); это наиболее широко используемый тип соединения. Поясню: если в таблице `department` есть четвертая строка для отдела маркетинга, но ни один сотрудник не приписан к нему, отдел маркетинга не попадет в результирующий набор. Анало-

гично, если некоторые сотрудники зарегистрированы в отделе с ID 99, которого нет в таблице department, эти сотрудники не попадут в результирующий набор. Если требуется включить все строки той или иной таблицы независимо от наличия соответствия, можно воспользоваться *внешним соединением* (*outer join*), но мы рассмотрим это в главе 10.

В предыдущем примере в блоке from я не указал тип используемого соединения. Однако если требуется соединить две таблицы путем внутреннего соединения, это следует явно указать в блоке from. Вот тот же пример с добавлением типа соединения (обратите внимание на ключевое слово INNER (внутренний)):

```
mysql> SELECT e.fname, e.lname, d.name
    -> FROM employee e INNER JOIN department d
    ->   ON e.dept_id = d.dept_id;
+----------+-----------+----------------+
| fname    | lname     | name           |
+----------+-----------+----------------+
| Susan    | Hawthorne | Operations     |
| Helen    | Fleming   | Operations     |
| Chris    | Tucker    | Operations     |
| Sarah    | Parker    | Operations     |
| Jane     | Grossman  | Operations     |
| Paula    | Roberts   | Operations     |
| Thomas   | Ziegler   | Operations     |
| Samantha | Jameson   | Operations     |
| John     | Blake     | Operations     |
| Cindy    | Mason     | Operations     |
| Frank    | Portman   | Operations     |
| Theresa  | Markham   | Operations     |
| Beth     | Fowler    | Operations     |
| Rick     | Tulman    | Operations     |
| John     | Gooding   | Loans          |
| Michael  | Smith     | Administration |
| Susan    | Barker    | Administration |
| Robert   | Tyler     | Administration |
+----------+-----------+----------------+
18 rows in set (0.00 sec)
```

Если тип соединения не задан, сервер по умолчанию проведет внутреннее соединение. Однако, как выяснится в главе 10, есть несколько типов соединений, поэтому указание точного типа требуемого соединения должно войти в привычку.

Если имена столбцов, используемых для соединения двух таблиц, совпадают (что имеет место в предыдущем запросе), можно вместо подблока on применить подблок using:

```
mysql> SELECT e.fname, e.lname, d.name
    -> FROM employee e INNER JOIN department d
    ->   USING (dept_id);
+----------+-----------+----------------+
```

```
| fname    | lname     | name           |
+----------+-----------+----------------+
| Susan    | Hawthorne | Operations     |
| Helen    | Fleming   | Operations     |
| Chris    | Tucker    | Operations     |
| Sarah    | Parker    | Operations     |
| Jane     | Grossman  | Operations     |
| Paula    | Roberts   | Operations     |
| Thomas   | Ziegler   | Operations     |
| Samantha | Jameson   | Operations     |
| John     | Blake     | Operations     |
| Cindy    | Mason     | Operations     |
| Frank    | Portman   | Operations     |
| Theresa  | Markham   | Operations     |
| Beth     | Fowler    | Operations     |
| Rick     | Tulman    | Operations     |
| John     | Gooding   | Loans          |
| Michael  | Smith     | Administration |
| Susan    | Barker    | Administration |
| Robert   | Tyler     | Administration |
+----------+-----------+----------------+
18 rows in set (0.01 sec)
```

Поскольку `using` – сокращенная запись, которая может использоваться только в определенной ситуации, во избежание путаницы я всегда предпочитаю подблок `on`.

ANSI-синтаксис соединения

Нотация, используемая в данной книге для соединения таблиц, была введена в версии SQL92 стандарта ANSI SQL. Во всех основных СУБД (Oracle Database, Microsoft SQL Server, MySQL, IBM DB2 Universal Database, Sybase Adaptive Server) принят синтаксис соединения SQL92. Поскольку многие серверы существовали еще до выхода спецификации SQL92, все они включают и старый синтаксис соединения. Например, всем этим серверам был бы понятен такой вариант предыдущего запроса:

```
mysql> SELECT e.fname, e.lname, d.name
    -> FROM employee e, department d
    -> WHERE e.dept_id = d.dept_id;
+----------+-----------+----------------+
| fname    | lname     | name           |
+----------+-----------+----------------+
| Susan    | Hawthorne | Operations     |
| Helen    | Fleming   | Operations     |
| Chris    | Tucker    | Operations     |
| Sarah    | Parker    | Operations     |
| Jane     | Grossman  | Operations     |
| Paula    | Roberts   | Operations     |
| Thomas   | Ziegler   | Operations     |
```

```
| Samantha | Jameson   | Operations     |
| John     | Blake     | Operations     |
| Cindy    | Mason     | Operations     |
| Frank    | Portman   | Operations     |
| Theresa  | Markham   | Operations     |
| Beth     | Fowler    | Operations     |
| Rick     | Tulman    | Operations     |
| John     | Gooding   | Loans          |
| Michael  | Smith     | Administration |
| Susan    | Barker    | Administration |
| Robert   | Tyler     | Administration |
+----------+-----------+----------------+
18 rows in set (0.01 sec)
```

Этот старый метод описания соединений не включает подблок `on`. Таблицы указаны в блоке `from` через запятую, а условия соединения включены в блок `where`. Хотя можно игнорировать синтаксис SQL92 в пользу старого синтаксиса соединений, у синтаксиса ANSI есть следующие преимущества:

- Условия соединения и фильтрации разнесены в два разных блока (подблок `on` и блок `where` соответственно), что упрощает понимание запроса.
- Условия соединения для каждой пары таблиц содержатся в собственном блоке `on`, что уменьшает вероятность ошибочного исключения части соединения.
- Запросы, использующие синтаксис соединения SQL92, портируются на разные серверы БД, тогда как старый синтаксис немного отличается для каждого сервера.

Преимущества синтаксиса соединения SQL92 заметнее в сложных запросах, включающих как условия соединения, так и условия фильтрации. Рассмотрим следующий запрос, по которому возвращаются все счета, открытые опытными операционистами (нанятыми до 2003 года), в настоящее время приписанными к отделению Woburn:

```
mysql> SELECT a.account_id, a.cust_id, a.open_date, a.product_cd
    -> FROM account a, branch b, employee e
    -> WHERE a.open_emp_id = e.emp_id
    ->   AND e.start_date <= '2003-01-01'
    ->   AND e.assigned_branch_id = b.branch_id
    ->   AND (e.title = 'Teller' OR e.title = 'Head Teller')
    ->   AND b.name = 'Woburn Branch';
+------------+---------+------------+------------+
| account_id | cust_id | open_date  | product_cd |
+------------+---------+------------+------------+
|          1 |       1 | 2000-01-15 | CHK        |
|          2 |       1 | 2000-01-15 | SAV        |
|          3 |       1 | 2004-06-30 | CD         |
|          4 |       2 | 2001-03-12 | CHK        |
|          5 |       2 | 2001-03-12 | SAV        |
```

```
|          14 |        7 | 2004-01-12 | CD         |
|          22 |       11 | 2004-03-22 | BUS        |
+-------------+----------+------------+------------+
7 rows in set (0.01 sec)
```

В этом запросе не так просто определить, какие условия блока `where` являются условиями соединения, а какие – условиями фильтрации. Также не вполне очевидно, какой тип соединения используется (для установления типа соединения необходимо внимательно рассмотреть условия соединения в блоке `where` – нет ли там каких-либо специальных символов), и сложно определить, не были ли упущены какие-либо условия соединения. Вот тот же запрос, записанный с использованием синтаксиса соединения SQL92:

```
mysql> SELECT a.account_id, a.cust_id, a.open_date, a.product_cd
    -> FROM account a INNER JOIN employee e
    ->   ON a.open_emp_id = e.emp_id
    ->   INNER JOIN branch b
    ->   ON e.assigned_branch_id = b.branch_id
    -> WHERE e.start_date <= '2003-01-01'
    ->   AND (e.title = 'Teller' OR e.title = 'Head Teller')
    ->   AND b.name = 'Woburn Branch';
+------------+---------+------------+------------+
| account_id | cust_id | open_date  | product_cd |
+------------+---------+------------+------------+
|          1 |       1 | 2000-01-15 | CHK        |
|          2 |       1 | 2000-01-15 | SAV        |
|          3 |       1 | 2004-06-30 | CD         |
|          4 |       2 | 2001-03-12 | CHK        |
|          5 |       2 | 2001-03-12 | SAV        |
|         14 |       7 | 2004-01-12 | CD         |
|         22 |      11 | 2004-03-22 | BUS        |
+------------+---------+------------+------------+
7 rows in set (0.36 sec)
```

Надеюсь, все согласятся, что понятнее версия, использующая синтаксис SQL92.

Соединение трех и более таблиц

Соединение трех таблиц аналогично соединению двух, но есть небольшая хитрость. При соединении двух таблиц имеются две таблицы и один тип соединения в блоке `from`, а также единственный подблок `on`, определяющий, как соединяются таблицы. При соединении трех таблиц присутствуют три таблицы и два типа соединения в блоке `from`, а также два подблока `on`. Вот еще один пример запроса с соединением двух таблиц:

```
mysql> SELECT a.account_id, c.fed_id
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id
```

```
    -> WHERE c.cust_type_cd = 'B';
+------------+------------+
| account_id | fed_id     |
+------------+------------+
|         20 | 04-1111111 |
|         21 | 04-1111111 |
|         22 | 04-2222222 |
|         23 | 04-3333333 |
|         24 | 04-4444444 |
+------------+------------+
5 rows in set (0.06 sec)
```

Этот запрос, возвращающий ID счета и идентификационный номер федерального налога для всех бизнес-счетов, на данный момент должен быть абсолютно понятным. Однако если добавить в запрос таблицу `employee` для получения имени операциониста, открывшего каждый из счетов, он примет следующий вид:

```
mysql> SELECT a.account_id, c.fed_id, e.fname, e.lname
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id
    ->   INNER JOIN employee e
    ->   ON a.open_emp_id = e.emp_id
    -> WHERE c.cust_type_cd = 'B';
+------------+------------+---------+---------+
| account_id | fed_id     | fname   | lname   |
+------------+------------+---------+---------+
|         20 | 04-1111111 | Theresa | Markham |
|         21 | 04-1111111 | Theresa | Markham |
|         22 | 04-2222222 | Paula   | Roberts |
|         23 | 04-3333333 | Theresa | Markham |
|         24 | 04-4444444 | John    | Blake   |
+------------+------------+---------+---------+
5 rows in set (0.03 sec)
```

Теперь имеется три таблицы, два типа соединений и два подблока `on`, перечисленные в блоке `from`. Таким образом, все немного усложняется. На первый взгляд из-за порядка перечисления таблиц может показаться, что таблица `employee` соединяется с таблицей `customer`, поскольку таблица `account` указана первой. Однако если изменить порядок расположения таблиц, результаты будут абсолютно аналогичными:

```
mysql> SELECT a.account_id, c.fed_id, e.fname, e.lname
    -> FROM customer c INNER JOIN account a
    ->   ON a.cust_id = c.cust_id
    ->   INNER JOIN employee e
    ->   ON a.open_emp_id = e.emp_id
    -> WHERE c.cust_type_cd = 'B';
+------------+------------+---------+---------+
| account_id | fed_id     | fname   | lname   |
+------------+------------+---------+---------+
|         20 | 04-1111111 | Theresa | Markham |
```

```
|         21 | 04-1111111 | Theresa | Markham |
|         22 | 04-2222222 | Paula   | Roberts |
|         23 | 04-3333333 | Theresa | Markham |
|         24 | 04-4444444 | John    | Blake   |
+------------+------------+---------+---------+
5 rows in set (0.00 sec)
```

Теперь таблица `customer` стоит первой, за ней идут таблицы `account` и `employee`. Поскольку подблоки `on` не изменились, результаты такие же.

Запрос, использующий три или более таблиц, можно представить как снежный ком, катящийся с горы. Первые две таблицы запускают этот ком, а каждая последующая таблица вносит в него свою лепту. Снежный ком можно рассматривать как *промежуточный результирующий набор* (*intermediate result set*), который подбирает все больше и больше столбцов по мере соединения с последующими таблицами. Поэтому таблица `employee` в действительности была соединена не с таблицей `account`, а с промежуточным результирующим набором, созданным при соединении таблиц `customer` и `account`. (Если интересно, откуда взялся снежный ком: я писал эту главу глубокой новоанглийской зимой – уже выпало 110 дюймов и завтра будет еще. Вот радость-то!)

Применение подзапросов в качестве таблиц

Вы уже посмотрели несколько примеров запросов с тремя таблицами, но хочется особо отметить одну из разновидностей таких запросов. Что делать, если некоторые таблицы формируются подзапросами? Подзапросам посвящена глава 9, но концепция подзапроса в блоке `from` уже была представлена в предыдущей главе. Вот другая версия приведенного ранее запроса (выбор всех счетов, открытых опытными операционистами, в настоящее время работающими в отделении Woburn), в котором таблица `account` соединяется с подзапросами к таблицам `branch` (отделение) и `employee`:

```
1  SELECT a.account_id, a.cust_id, a.open_date, a.product_cd
2  FROM account a INNER JOIN
3    (SELECT emp_id, assigned_branch_id
4      FROM employee
5      WHERE start_date <= '2003-01-01'
6       AND (title = 'Teller' OR title = 'Head Teller')) e
7    ON a.open_emp_id = e.emp_id
8    INNER JOIN
9     (SELECT branch_id
10     FROM branch
11     WHERE name = 'Woburn Branch') b
12   ON e.assigned_branch_id = b.branch_id;
```

Первый подзапрос, начинающийся в строке 3 и имеющий псевдоним `e`, находит всех опытных операционистов. Второй подзапрос, начинающийся в строке 9 и имеющий псевдоним `b`, выбирает ID отделения Woburn. Сначала таблица `account` соединяется с подзапросом по опытным

операционистам с помощью ID сотрудников, а затем результирующая таблица соединяется с подзапросом по отделению Woburn на основе ID филиала. Результаты аналогичны предыдущей версии запроса (попробуйте и убедитесь сами), но на вид запросы очень отличаются друг от друга.

На самом деле здесь нет ничего необычного, но осознать происходящее можно не сразу. Обратите внимание, например, на отсутствие блока `where` в основном запросе. Поскольку все условия фильтрации касаются таблиц `employee` и `branch`, они находятся в подзапросах. В основном запросе никакие условия фильтрации не нужны. Единственный способ проиллюстрировать происходящее – выполнить подзапросы и посмотреть на результирующие наборы. Вот результаты выполнения первого подзапроса к таблице `employee`:

```
mysql> SELECT emp_id, assigned_branch_id
    -> FROM employee
    -> WHERE start_date <= '2003-01-01'
    ->   AND (title = 'Teller' OR title = 'Head Teller');
+--------+--------------------+
| emp_id | assigned_branch_id |
+--------+--------------------+
|      8 |                  1 |
|      9 |                  1 |
|     10 |                  2 |
|     11 |                  2 |
|     13 |                  3 |
|     14 |                  3 |
|     16 |                  4 |
|     17 |                  4 |
|     18 |                  4 |
+--------+--------------------+
9 rows in set (0.03 sec)
```

Таким образом, результирующий набор состоит из набора ID сотрудников и ID соответствующих им отделений. Соединив таблицу `account` посредством столбца `emp_id`, имеем промежуточный результирующий набор, содержащий все строки таблицы `account` и дополнительный столбец с ID отделения, в котором работает сотрудник, открывший каждый из счетов. Вот результаты второго подзапроса к таблице `branch`:

```
mysql> SELECT branch_id
    -> FROM branch
    -> WHERE name = 'Woburn Branch';
+-----------+
| branch_id |
+-----------+
|         2 |
+-----------+
1 row in set (0.02 sec)
```

По этому запросу возвращается одна-единственная строка с одним столбцом: ID отделения Woburn. Эта таблица соединена со столбцом assigned_branch_id промежуточного результирующего набора, что позволяет отсеять из окончательного результирующего набора все счета, открытые несотрудниками отделения Woburn.

Повторное использование таблицы

При соединении нескольких таблиц может обнаружиться, что требуется неоднократно соединять одну и ту же таблицу. В нашем примере БД внешние ключи к таблице branch присутствуют как в таблице account (отделение, в котором был открыт счет), так и в таблице employee (отделение, в котором работает сотрудник). Если в результирующий набор должны войти *оба* отделения, таблицу branch можно включить в блок from дважды, в первый раз соединив ее с таблицей employee, а второй раз – с таблицей account. Это сработает, если каждому экземпляру таблицы branch присвоить свой псевдоним, чтобы сервер знал, на какой экземпляр делается ссылка:

```
mysql> SELECT a.account_id, e.emp_id,
    ->   b_a.name open_branch, b_e.name emp_branch
    -> FROM account a INNER JOIN branch b_a
    ->   ON a.open_branch_id = b_a.branch_id
    ->   INNER JOIN employee e
    ->   ON a.open_emp_id = e.emp_id
    ->   INNER JOIN branch b_e
    ->   ON e.assigned_branch_id = b_e.branch_id
    -> WHERE a.product_cd = 'CHK';
+------------+--------+---------------+---------------+
| account_id | emp_id | open_branch   | emp_branch    |
+------------+--------+---------------+---------------+
|          8 |      1 | Headquarters  | Headquarters  |
|         12 |      1 | Headquarters  | Headquarters  |
|         17 |      1 | Headquarters  | Headquarters  |
|          1 |     10 | Woburn Branch | Woburn Branch |
|          4 |     10 | Woburn Branch | Woburn Branch |
|          6 |     13 | Quincy Branch | Quincy Branch |
|         11 |     16 | So. NH Branch | So. NH Branch |
|         15 |     16 | So. NH Branch | So. NH Branch |
|         20 |     16 | So. NH Branch | So. NH Branch |
|         23 |     16 | So. NH Branch | So. NH Branch |
+------------+--------+---------------+---------------+
10 rows in set (0.07 sec)
```

Этот запрос показывает, кто открыл каждый текущий счет, в каком отделении это произошло и к какому отделению приписан в настоящее время сотрудник, открывший счет. Таблица branch включена дважды под псевдонимами b_a и b_e. Благодаря присваиванию разных псевдонимов каждому экземпляру таблицы branch, сервер сможет определить экземпляр, на который делается ссылка, – соединенный с таблицей

account или с таблицей employee. Таким образом, имеем пример запроса, в котором использование псевдонимов таблиц является *обязательным*.

Рекурсивные соединения

Можно не только несколько раз включать одну и ту же таблицу в один запрос, фактически можно соединить таблицу с самой собой. Поначалу это может показаться странным, но для этого есть веские основания. В таблице employee, например, есть *рекурсивный внешний ключ* (*self-referencing foreign key*). Это означает, что она включает столбец (superior_emp_id), указывающий на первичный ключ в рамках той же таблицы. Этот столбец указывает на начальника сотрудника (если только это не сам босс – тогда столбец имеет значение null). С помощью *рекурсивного соединения* (*self-join*) можно создать запрос, в результате выполнения которого выводится список всех сотрудников с указанием имен их начальников:

```
mysql> SELECT e.fname, e.lname,
    ->   e_mgr.fname mgr_fname, e_mgr.lname mgr_lname
    -> FROM employee e INNER JOIN employee e_mgr
    ->   ON e.superior_emp_id = e_mgr.emp_id;
+----------+-----------+-----------+-----------+
| fname    | lname     | mgr_fname | mgr_lname |
+----------+-----------+-----------+-----------+
| Susan    | Barker    | Michael   | Smith     |
| Robert   | Tyler     | Michael   | Smith     |
| Susan    | Hawthorne | Robert    | Tyler     |
| John     | Gooding   | Susan     | Hawthorne |
| Helen    | Fleming   | Susan     | Hawthorne |
| Chris    | Tucker    | Helen     | Fleming   |
| Sarah    | Parker    | Helen     | Fleming   |
| Jane     | Grossman  | Helen     | Fleming   |
| Paula    | Roberts   | Susan     | Hawthorne |
| Thomas   | Ziegler   | Paula     | Roberts   |
| Samantha | Jameson   | Paula     | Roberts   |
| John     | Blake     | Susan     | Hawthorne |
| Cindy    | Mason     | John      | Blake     |
| Frank    | Portman   | John      | Blake     |
| Theresa  | Markham   | Susan     | Hawthorne |
| Beth     | Fowler    | Theresa   | Markham   |
| Rick     | Tulman    | Theresa   | Markham   |
+----------+-----------+-----------+-----------+
17 rows in set (0.01 sec)
```

Этот запрос включает два экземпляра таблицы employee: из одного (под псевдонимом e) извлекаются имена сотрудников, а из другого (под псевдонимом e_mgr) – имена начальников. Подблок on использует эти псевдонимы для соединения таблицы employee с самой собой посредством внешнего ключа superior_emp_id. Это еще один пример запроса, для ко-

торого псевдонимы таблиц являются обязательными. В противном случае сервер не сможет определить, на кого делается ссылка – на сотрудника или его начальника.

Хотя в таблице employee 18 строк, по запросу было возвращено только 17. У президента банка, Майкла Смита (Michael Smith), нет начальника (его столбец superior_emp_id имеет значение null), поэтому для данной строки соединение не сработало. Чтобы включить Майкла Смита в результирующий набор, необходимо использовать внешнее соединение, которое будет рассмотрено в главе 10.

Сравнение эквисоединений с неэквисоединениями

Все запросы к нескольким таблицам, показанные до сих пор, использовали *эквисоединения* (*equi-joins*). Это означает, что для обеспечения успешности соединения значения двух таблиц должны совпадать. Эквисоединение всегда использует знак равенства, например:

```
ON e.assigned_branch_id = b.branch_id
```

Подавляющее большинство запросов использует эквисоединения, но можно также соединять таблицы посредством диапазонов значений, называемых *неэквисоединениями* (*non-equi-joins*). Вот пример запроса, осуществляющего соединение по диапазону значений:

```
SELECT e.emp_id, e.fname, e.lname, e.start_date
FROM employee e INNER JOIN product p
  ON e.start_date >= p.date_offered
    AND e.start_date <= p.date_retired
WHERE p.name = 'no-fee checking';
```

Этот запрос соединяет две таблицы, между которыми нет взаимосвязей по внешним ключам. Задача – найти всех сотрудников, принятых в банк в то время, когда предлагалась услуга беспроцентного текущего вклада. Таким образом, дата начала работы сотрудника должна находиться между датами начала и конца этой акции.

Также может понадобиться *рекурсивное неэквисоединение* (*self-non-equi-join*), которое означает, что таблица соединяется сама с собой с использованием неэквисоединения. Например, управляющий операциями решил провести шахматный турнир между всеми операционистами банка. Требуется создать список всех пар игроков. Можно попробовать получить список всех операционистов (title = 'Teller'), соединив таблицу employee с самой собой, и выбрать из него все строки с разными значениями emp_id (поскольку игрок не может составить пару с самим собой):

```
mysql> SELECT e1.fname, e1.lname, 'VS' vs, e2.fname, e2.lname
    -> FROM employee e1 INNER JOIN employee e2
```

```
    ->   ON e1.emp_id != e2.emp_id
    -> WHERE e1.title = 'Teller' AND e2.title = 'Teller';
+----------+----------+----+----------+----------+
| fname    | lname    | vs | fname    | lname    |
+----------+----------+----+----------+----------+
| Sarah    | Parker   | VS | Chris    | Tucker   |
| Jane     | Grossman | VS | Chris    | Tucker   |
| Thomas   | Ziegler  | VS | Chris    | Tucker   |
| Samantha | Jameson  | VS | Chris    | Tucker   |
| Cindy    | Mason    | VS | Chris    | Tucker   |
| Frank    | Portman  | VS | Chris    | Tucker   |
| Beth     | Fowler   | VS | Chris    | Tucker   |
| Rick     | Tulman   | VS | Chris    | Tucker   |
| Chris    | Tucker   | VS | Sarah    | Parker   |
| Jane     | Grossman | VS | Sarah    | Parker   |
| Thomas   | Ziegler  | VS | Sarah    | Parker   |
| Samantha | Jameson  | VS | Sarah    | Parker   |
| Cindy    | Mason    | VS | Sarah    | Parker   |
| Frank    | Portman  | VS | Sarah    | Parker   |
| Beth     | Fowler   | VS | Sarah    | Parker   |
| Rick     | Tulman   | VS | Sarah    | Parker   |
...
| Chris    | Tucker   | VS | Rick     | Tulman   |
| Sarah    | Parker   | VS | Rick     | Tulman   |
| Jane     | Grossman | VS | Rick     | Tulman   |
| Thomas   | Ziegler  | VS | Rick     | Tulman   |
| Samantha | Jameson  | VS | Rick     | Tulman   |
| Cindy    | Mason    | VS | Rick     | Tulman   |
| Frank    | Portman  | VS | Rick     | Tulman   |
| Beth     | Fowler   | VS | Rick     | Tulman   |
+----------+----------+----+----------+----------+
72 rows in set (0.01 sec)
```

Мы на правильном пути, но проблема здесь в том, что для каждой пары (например, Сара Паркер (Sarah Parker) против Криса Такера (Chris Tucker)) имеется «обратная» пара (т. е. Крис Такер против Сары Паркер). Один из способов достигнуть желаемого результата – использовать условие соединения `e1.emp_id < e2.emp_id`, чтобы каждый операционист входил в пару только с теми, у кого ID сотрудника больше (можно также использовать `e1.emp_id > e2.emp_id`, если вам так больше нравится):

```
mysql> SELECT e1.fname, e1.lname, 'VS' vs, e2.fname, e2.lname
    -> FROM employee e1 INNER JOIN employee e2
    ->   ON e1.emp_id < e2.emp_id
    -> WHERE e1.title = 'Teller' AND e2.title = 'Teller';
+----------+----------+----+----------+----------+
| fname    | lname    | vs | fname    | lname    |
+----------+----------+----+----------+----------+
| Chris    | Tucker   | VS | Sarah    | Parker   |
| Chris    | Tucker   | VS | Jane     | Grossman |
```

```
| Chris    | Tucker   | VS | Thomas   | Ziegler  |
| Chris    | Tucker   | VS | Samantha | Jameson  |
| Chris    | Tucker   | VS | Cindy    | Mason    |
| Chris    | Tucker   | VS | Frank    | Portman  |
| Chris    | Tucker   | VS | Beth     | Fowler   |
| Chris    | Tucker   | VS | Rick     | Tulman   |
| Sarah    | Parker   | VS | Jane     | Grossman |
| Sarah    | Parker   | VS | Thomas   | Ziegler  |
| Sarah    | Parker   | VS | Samantha | Jameson  |
| Sarah    | Parker   | VS | Cindy    | Mason    |
| Sarah    | Parker   | VS | Frank    | Portman  |
| Sarah    | Parker   | VS | Beth     | Fowler   |
| Sarah    | Parker   | VS | Rick     | Tulman   |
| Jane     | Grossman | VS | Thomas   | Ziegler  |
| Jane     | Grossman | VS | Samantha | Jameson  |
| Jane     | Grossman | VS | Cindy    | Mason    |
| Jane     | Grossman | VS | Frank    | Portman  |
| Jane     | Grossman | VS | Beth     | Fowler   |
| Jane     | Grossman | VS | Rick     | Tulman   |
| Thomas   | Ziegler  | VS | Samantha | Jameson  |
| Thomas   | Ziegler  | VS | Cindy    | Mason    |
| Thomas   | Ziegler  | VS | Frank    | Portman  |
| Thomas   | Ziegler  | VS | Beth     | Fowler   |
| Thomas   | Ziegler  | VS | Rick     | Tulman   |
| Samantha | Jameson  | VS | Cindy    | Mason    |
| Samantha | Jameson  | VS | Frank    | Portman  |
| Samantha | Jameson  | VS | Beth     | Fowler   |
| Samantha | Jameson  | VS | Rick     | Tulman   |
| Cindy    | Mason    | VS | Frank    | Portman  |
| Cindy    | Mason    | VS | Beth     | Fowler   |
| Cindy    | Mason    | VS | Rick     | Tulman   |
| Frank    | Portman  | VS | Beth     | Fowler   |
| Frank    | Portman  | VS | Rick     | Tulman   |
| Beth     | Fowler   | VS | Rick     | Tulman   |
+----------+----------+----+----------+----------+
36 rows in set (0.01 sec)
```

Теперь у нас есть список из 36 пар. Как раз столько, сколько должно быть при наличии девяти участников.

Сравнение условий соединения и условий фильтрации

Теперь мы знаем, что условия соединения относятся к подблоку `on`, тогда как условия фильтрации располагаются в блоке `where`. Однако SQL не налагает жестких ограничений на размещение условий, поэтому создавать запросы следует очень внимательно. Например, следующий запрос соединяет две таблицы с помощью одного блока соединения и одного условия фильтрации в блоке `where`:

```
mysql> SELECT a.account_id, a.product_cd, c.fed_id
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id
    -> WHERE c.cust_type_cd = 'B';
+------------+------------+------------+
| account_id | product_cd | fed_id     |
+------------+------------+------------+
|         20 | CHK        | 04-1111111 |
|         21 | BUS        | 04-1111111 |
|         22 | BUS        | 04-2222222 |
|         23 | CHK        | 04-3333333 |
|         24 | SBL        | 04-4444444 |
+------------+------------+------------+
5 rows in set (0.08 sec)
```

Достаточно просто, но что произойдет, если по ошибке поместить условие фильтрации в подблок `on`, а не в блок `where`?

```
mysql> SELECT a.account_id, a.product_cd, c.fed_id
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id
    ->     AND c.cust_type_cd = 'B';
+------------+------------+------------+
| account_id | product_cd | fed_id     |
+------------+------------+------------+
|         20 | CHK        | 04-1111111 |
|         21 | BUS        | 04-1111111 |
|         22 | BUS        | 04-2222222 |
|         23 | CHK        | 04-3333333 |
|         24 | SBL        | 04-4444444 |
+------------+------------+------------+
5 rows in set (0.00 sec)
```

Как видите, второй вариант, в котором *оба* условия находятся в подблоке `on` и нет блока `where`, обеспечивает аналогичные результаты. А что если оба условия помещены в блок `where`, но блок `from` по-прежнему использует ANSI-синтаксис соединения?

```
mysql> SELECT a.account_id, a.product_cd, c.fed_id
    -> FROM account a INNER JOIN customer c
    -> WHERE a.cust_id = c.cust_id
    ->   AND c.cust_type_cd = 'B';
+------------+------------+------------+
| account_id | product_cd | fed_id     |
+------------+------------+------------+
|         20 | CHK        | 04-1111111 |
|         21 | BUS        | 04-1111111 |
|         22 | BUS        | 04-2222222 |
|         23 | CHK        | 04-3333333 |
|         24 | SBL        | 04-4444444 |
+------------+------------+------------+
5 rows in set (0.00 sec)
```

Сервер MySQL снова сгенерировал тот же результирующий набор. Расположить условия на соответствующих местах, чтобы запрос был правильно понят и обработан, – ваша задача.

Упражнения

Следующие упражнения призваны протестировать понимание внутренних соединений. Решения приведены в приложении C.

5.1

Заполните в следующем запросе пробелы (обозначенные как *<число>*), чтобы получить такие результаты:

```
mysql> SELECT e.emp_id, e.fname, e.lname, b.name
    -> FROM employee e INNER JOIN <1> b
    ->   ON e.assigned_branch_id = b.<2>;
+--------+----------+-----------+---------------+
| emp_id | fname    | lname     | name          |
+--------+----------+-----------+---------------+
|      1 | Michael  | Smith     | Headquarters  |
|      2 | Susan    | Barker    | Headquarters  |
|      3 | Robert   | Tyler     | Headquarters  |
|      4 | Susan    | Hawthorne | Headquarters  |
|      5 | John     | Gooding   | Headquarters  |
|      6 | Helen    | Fleming   | Headquarters  |
|      7 | Chris    | Tucker    | Headquarters  |
|      8 | Sarah    | Parker    | Headquarters  |
|      9 | Jane     | Grossman  | Headquarters  |
|     10 | Paula    | Roberts   | Woburn Branch |
|     11 | Thomas   | Ziegler   | Woburn Branch |
|     12 | Samantha | Jameson   | Woburn Branch |
|     13 | John     | Blake     | Quincy Branch |
|     14 | Cindy    | Mason     | Quincy Branch |
|     15 | Frank    | Portman   | Quincy Branch |
|     16 | Theresa  | Markham   | So. NH Branch |
|     17 | Beth     | Fowler    | So. NH Branch |
|     18 | Rick     | Tulman    | So. NH Branch |
+--------+----------+-----------+---------------+
18 rows in set (0.03 sec)
```

5.2

Напишите запрос, по которому для каждого клиента-физического лица (`customer.cust_type_cd = 'I'`) возвращаются ID счета, федеральный ID (`customer.fed_id`) и тип созданного счета (`product.name`).

5.3

Создайте запрос для выбора всех сотрудников, начальник которых приписан к другому отделу. Извлеките ID, имя и фамилию сотрудника.

6

Работа с множествами

Хотя можно работать и с отдельными строками данных, реляционные БД на самом деле приспособлены для работы с наборами (множествами). Вы уже видели, как можно создавать таблицы посредством запросов или подзапросов, делать их постоянными с помощью выражений `insert` и сводить вместе через соединения. В данной главе будут исследованы комбинации нескольких таблиц с использованием различных операторов работы с множествами.

Основы теории множеств

Во многих странах основы теории множеств включены в программы начального курса математики. Возможно, кое-что на рис. 6.1 покажется вам знакомым.

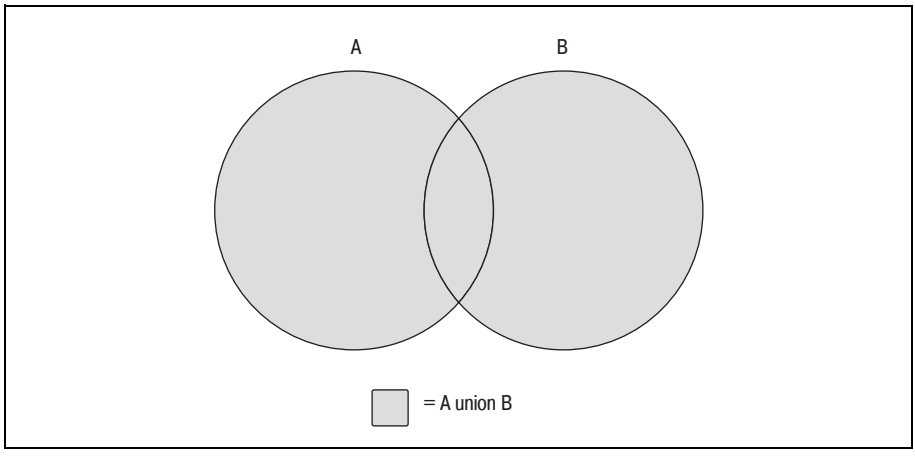


Рис. 6.1. *Операция объединения*

Заштрихованные области на рис. 6.1 представляют *объединение* (*union*) множеств A и B, которое является комбинацией двух множеств (при этом все пересекающиеся области включены только один раз). Что-то припоминаете? Если да, то наконец появился шанс применить эти знания на практике. Если нет, не волнуйтесь, потому что без труда поймете все, взглянув на пару диаграмм.

Представим множества (A и B) в виде кругов; область перекрытия представляет подмножество данных, общих для обоих множеств (рис. 6.1). Поскольку без перекрытий множеств данных теория множеств совершенно неинтересна, я буду использовать такую же диаграмму для иллюстрации всех операций с множествами. Есть другая операция, результат которой – *только* перекрытие двух множеств данных. Эту операцию называют *пересечением* (*intersection*) (рис. 6.2).

Множество данных, получаемое в результате пересечения множеств A и B, – это собственно область перекрытия между двумя множествами. Если два множества не перекрываются, операция пересечения дает пустое множество.

Третья и последняя операция с множествами (рис. 6.3) известна как операция *разности* (*except*). На рис. 6.3 показан результат операции `A except B`, который представляет собой множество A минус все пересечения с множеством B. Если два множества не пересекаются, в результате операции `A except B` будет получено полное множество A.

Применяя эти три операции или их сочетания, можно получать любые нужные результаты. Например, представим, что требуется создать множество, показанное на рис. 6.4.

Искомое множество включает множества A и B *без* области пересечения. Такое множество не может быть получено в результате ни одной из трех представленных ранее операций. Понадобится сначала создать

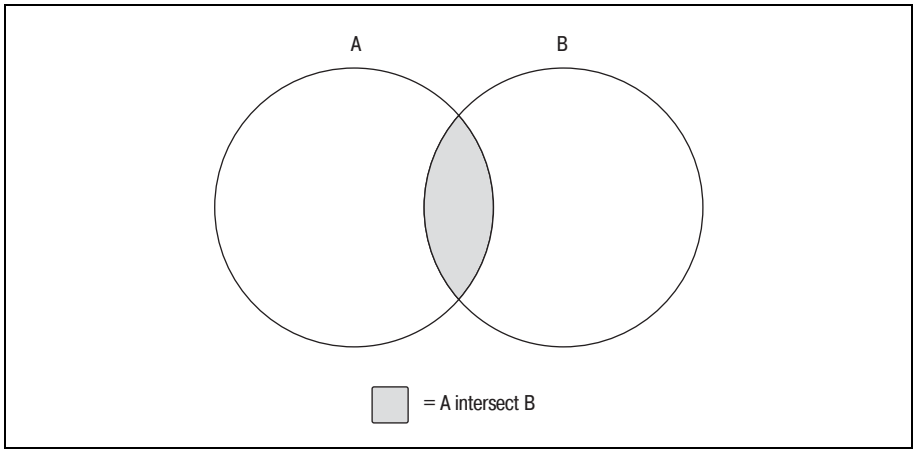


***Рис. 6.2.** Операция пересечения*

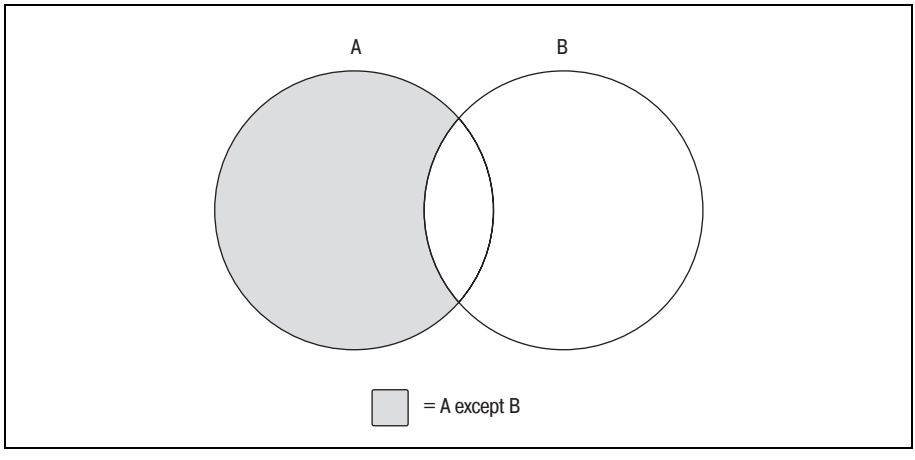


Рис. 6.3. *Операция разности*

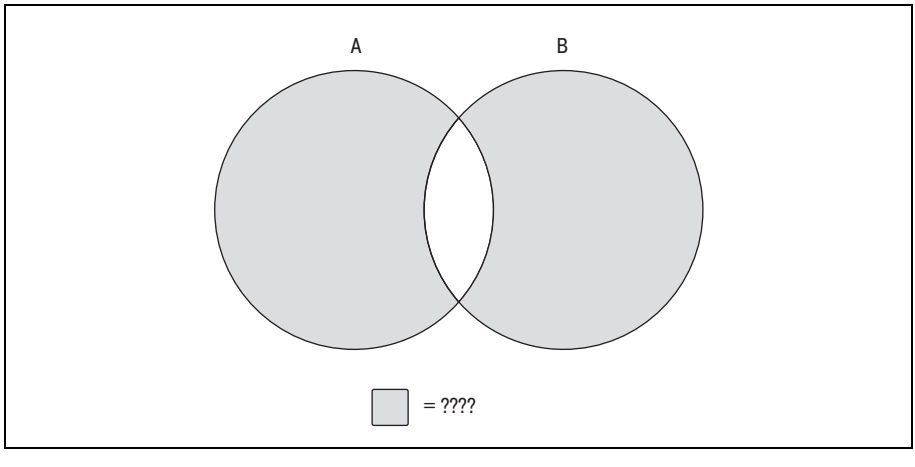


Рис. 6.4. *Загадочное множество данных*

множество данных, объединяющее множества A и B целиком, а затем применить вторую операцию, чтобы удалить область пересечения. Если составное множество описать как `A union B`, а область пересечения – как `A intersect B`, операция, необходимая для формирования представленного на рис. 6.4 множества, выглядела бы так:

```
(A union B) except (A intersect B)
```

Конечно, часто есть несколько способов получения одного и того же результата. Аналогичное множество можно было бы получить с помощью следующей операции:

```
(A except B) union (B except A)
```

Эти концепции, наглядно представленные диаграммами, достаточно просты для понимания. В следующих разделах будет показано, как эти идеи реализуются в реляционных СУБД с помощью SQL-операторов работы с множествами.

Теория множеств на практике

Круги, представляющие множества данных на диаграммах предыдущего раздела, никак не отражают содержимое множеств. Однако при работе с реальными данными необходимо описывать структуру таблиц, если предполагается их комбинирование. Представим, например, что произошло бы при попытке сгенерировать объединение описанных ниже таблиц `product` и `customer`:

```
mysql> DESC product;
+-----------------+-------------+------+-----+---------+-------+
| Field           | Type        | Null | Key | Default | Extra |
+-----------------+-------------+------+-----+---------+-------+
| product_cd      | varchar(10) |      | PRI |         |       |
| name            | varchar(50) |      |     |         |       |
| product_type_cd | varchar(10) |      | MUL |         |       |
| date_offered    | date        | YES  |     | NULL    |       |
| date_retired    | date        | YES  |     | NULL    |       |
+-----------------+-------------+------+-----+---------+-------+
5 rows in set (0.23 sec)

mysql> DESC customer;
+-------------+------------------+------+-----+---------+----------------+
| Field       | Type             | Null | Key | Default | Extra          |
+-------------+------------------+------+-----+---------+----------------+
| cust_id     | int(10) unsigned |      | PRI | NULL    | auto_increment |
| fed_id      | varchar(12)      |      |     |         |                |
| cust_type_cd | enum('I','B')   |      |     | I       |                |
| address     | varchar(30)      | YES  |     | NULL    |                |
| city        | varchar(20)      | YES  |     | NULL    |                |
| state       | varchar(20)      | YES  |     | NULL    |                |
| postal_code | varchar(10)      | YES  |     | NULL    |                |
+-------------+------------------+------+-----+---------+----------------+
7 rows in set (0.04 sec)
```

После комбинирования первый столбец результирующей таблицы был бы комбинацией столбцов `product.product_cd` и `customer.cust_id`, второй – комбинацией столбцов `product.name` и `customer.fed_id` и т. д. Хотя некоторые пары столбцов сочетаются без труда (т. е. два столбца числового типа), неясно, как должны объединяться пары столбцов разного типа, такие как числовой со строковым или строковый с датой. Кроме того, в шестом и седьмом столбцах комбинированной таблицы будут только данные шестого и седьмого столбцов таблицы `customer`, поскольку в таблице `product` всего пять столбцов. Очевидно, что таблицы, подлежащие объединению, должны обладать некоторой общностью.

Поэтому при применении операций с множествами к реальным таблицам необходимо соблюдать такие правила:

- В обеих таблицах должно быть одинаковое число столбцов.
- Типы данных столбцов двух таблиц должны быть одинаковыми (или сервер должен уметь преобразовывать один тип в другой).

Эти правила позволяют уяснить, что представляет собой «перекрытие данных» на практике. Чтобы комбинируемые строки двух таблиц считались одинаковыми, каждая пара столбцов комбинируемых таблиц должна содержать одинаковые строки, числа или даты.

Операции с множествами осуществляются путем помещения *оператора работы с множествами* (*set operator*) между двух выражений `select`, как показано ниже:

```
mysql> SELECT 1 num, 'abc' str
    -> UNION
    -> SELECT 9 num, 'xyz' str;
+-----+-----+
| num | str |
+-----+-----+
|   1 | abc |
|   9 | xyz |
+-----+-----+
2 rows in set (0.02 sec)
```

Каждый запрос формирует таблицу, состоящую из единственной строки с числовым и строковым столбцами. Оператор работы с множествами, в данном случае `union`, указывает серверу БД объединить все строки двух таблиц. Таким образом, конечная таблица включает две строки и два столбца. Такой запрос называют *составным запросом* (*compound query*), потому что он объединяет несколько независимых запросов. Как будет показано позже, если для получения окончательного результата требуется выполнить несколько операций с множествами, составные запросы могут включать *больше* двух запросов.

Операторы работы с множествами

Язык SQL включает три оператора работы с множествами, позволяющие осуществлять всевозможные операции над множествами, уже упомянутые в этой главе. Кроме того, у каждого из этих операторов есть две разновидности: первая включает дублирующие данные, а вторая удаляет их (но необязательно *все*). В следующих разделах даны определения всех операторов и показано их применение.

Оператор union

Операторы `union` (объединить) и `union all` (объединить все) позволяют комбинировать несколько таблиц. Разница в том, что если требуется объединить две таблицы, включая в окончательный результат *все* их

строки, даже дублирующие значения, нужно использовать оператор `union all`. Благодаря оператору `union all` в конечной таблице всегда будет столько строк, сколько во всех исходных таблицах в сумме. Эта операция – самая простая из всех операций работы с множествами (с точки зрения сервера), поскольку серверу не приходится проверять перекрывающиеся данные. Следующий пример демонстрирует применение оператора `union all` для формирования полного множества данных клиентов из двух таблиц подтипов клиентов:

```
mysql> SELECT cust_id, lname name
    -> FROM individual
    -> UNION ALL
    -> SELECT cust_id, name
    -> FROM business;
+---------+------------------------+
| cust_id | name                   |
+---------+------------------------+
|       1 | Hadley                 |
|       2 | Tingley                |
|       3 | Tucker                 |
|       4 | Hayward                |
|       5 | Frasier                |
|       6 | Spencer                |
|       7 | Young                  |
|       8 | Blake                  |
|       9 | Farley                 |
|      10 | Chilton Engineering    |
|      11 | Northeast Cooling Inc. |
|      12 | Superior Auto Body     |
|      13 | AAA Insurance Inc.     |
+---------+------------------------+
13 rows in set (0.04 sec)
```

Запрос возвращает все 13 клиентов: 9 строк поступают из таблицы `individual` (физические лица), а остальные 4 – из таблицы `business` (юридические лица). Таблица `business` включает всего один столбец с названием компании, а в таблице `individual` присутствуют два столбца: имя и фамилия физического лица. В данном случае из таблицы `individual` берется только фамилия.

Проверим, что оператор `union all` не удаляет дублирующие значения. Для этого приведем такой же запрос, как в предыдущем примере, но с дополнительным запросом к таблице `business`:

```
mysql> SELECT cust_id, lname name
    -> FROM individual
    -> UNION ALL
    -> SELECT cust_id, name
    -> FROM business
    -> UNION ALL
    -> SELECT cust_id, name
    -> FROM business;
```

```
+---------+------------------------+
| cust_id | name                   |
+---------+------------------------+
|       1 | Hadley                 |
|       2 | Tingley                |
|       3 | Tucker                 |
|       4 | Hayward                |
|       5 | Frasier                |
|       6 | Spencer                |
|       7 | Young                  |
|       8 | Blake                  |
|       9 | Farley                 |
|      10 | Chilton Engineering    |
|      11 | Northeast Cooling Inc. |
|      12 | Superior Auto Body     |
|      13 | AAA Insurance Inc.     |
|      10 | Chilton Engineering    |
|      11 | Northeast Cooling Inc. |
|      12 | Superior Auto Body     |
|      13 | AAA Insurance Inc.     |
+---------+------------------------+
17 rows in set (0.01 sec)
```

Этот составной запрос включает три выражения `select`, два из которых идентичны. Как видно по результатам, четыре строки из таблицы `business` включены дважды (ID клиентов 10, 11, 12 и 13).

Поскольку вряд ли вы когда-нибудь дважды включите один и тот же запрос в составной запрос, вот другой пример составного запроса, по которому возвращаются дублирующие данные:

```
mysql> SELECT emp_id
    -> FROM employee
    -> WHERE assigned_branch_id = 2
    ->   AND (title = 'Teller' OR title = 'Head Teller')
    -> UNION ALL
    -> SELECT DISTINCT open_emp_id
    -> FROM account
    -> WHERE open_branch_id = 2;
+--------+
| emp_id |
+--------+
|     10 |
|     11 |
|     12 |
|     10 |
+--------+
4 rows in set (0.01 sec)
```

Первый запрос составного выражения выбирает всех операционистов отделения Woburn, а второй возвращает другое множество – операционистов, открывавших счета в отделении Woburn. Из четырех строк ре-

зультирующего набора одна дублируется (ID сотрудника – 10). Если бы потребовалось *исключить* дублирующие строки из составной таблицы, вместо оператора `union all` надо было бы использовать оператор `union`:

```
mysql> SELECT emp_id
    -> FROM employee
    -> WHERE assigned_branch_id = 2
    ->   AND (title = 'Teller' OR title = 'Head Teller')
    -> UNION
    -> SELECT DISTINCT open_emp_id
    -> FROM account
    -> WHERE open_branch_id = 2;
+--------+
| emp_id |
+--------+
|     10 |
|     11 |
|     12 |
+--------+
3 rows in set (0.01 sec)
```

Для данной версии запроса с применением оператора `union all` в результирующий набор включаются только три разные строки, а не четыре (три разные, одна дублирующаяся).

Оператор intersect

Спецификация SQL ANSI включает оператор `intersect` (пересечение), предназначенный для выполнения пересечений. К сожалению, MySQL версии 4.1 не реализует оператор `intersect`. Oracle (но не SQL Server) позволяет использовать `intersect`. Однако поскольку для всех примеров данной книги используется MySQL, результирующие наборы для примеров запросов в данном разделе являются вымышленными и не могут быть получены в MySQL до версии 5.0 включительно. Здесь не показано приглашение MySQL (`mysql>`), потому что эти выражения не выполняются сервером MySQL.

Если два запроса составного запроса возвращают неперекрывающиеся таблицы, пересечением будет пустое множество. Рассмотрим следующий запрос:

```
SELECT emp_id, fname, lname
FROM employee
INTERSECT
SELECT cust_id, fname, lname
FROM individual;
Empty set (0.04 sec)
```

Первый запрос возвращает ID и имя каждого сотрудника, а второй – ID и имя каждого клиента. Это абсолютно неперекрывающиеся множества, поэтому пересечение двух этих множеств и дает пустое множество.

Второй шаг – выявить два запроса, *действительно* имеющих перекрывающиеся данные, и затем применить оператор intersect. Для этого используем тот же запрос, что и для демонстрации разницы между union и union all, только на этот раз возьмем оператор intersect:

```
SELECT emp_id
FROM employee
WHERE assigned_branch_id = 2
  AND (title = 'Teller' OR title = 'Head Teller')
INTERSECT
SELECT DISTINCT open_emp_id
FROM account
WHERE open_branch_id = 2;
+--------+
| emp_id |
+--------+
|     10 |
+--------+
1 row in set (0.01 sec)
```

Пересечение этих двух запросов дает сотрудника с ID равным 10, что является единственным значением, имеющимся в результирующих наборах обоих запросов.

Наряду с оператором intersect, удаляющим все дублирующие строки области перекрытия, спецификация SQL ANSI предлагает оператор intersect all, не удаляющий дубликаты. Единственный сервер БД, в настоящее время реализующий оператор intersect all, – DB2 Universal Server компании IBM.

Оператор except

Спецификация SQL ANSI включает оператор except (разность), предназначенный для выполнения операции разности. Опять же, к сожалению, MySQL версии 4.1 не реализует оператор except, поэтому в данном разделе действуют те же соглашения, что и в предыдущем.

При работе с Oracle Database вам понадобится использовать оператор minus (минус), не совместимый со спецификацией ANSI.

Операция except возвращает первую таблицу за вычетом всех перекрытий со второй таблицей. Вот пример из предыдущего раздела, но с оператором except вместо intersect:

```
SELECT emp_id
FROM employee
WHERE assigned_branch_id = 2
  AND (title = 'Teller' OR title = 'Head Teller')
EXCEPT
SELECT DISTINCT open_emp_id
FROM account
```

```
WHERE open_branch_id = 2;
+--------+
| emp_id |
+--------+
|     11 |
|     12 |
+--------+
2 rows in set (0.01 sec)
```

В этом варианте запроса результирующий набор включает три строки из результирующего набора первого запроса минус сотрудник с ID, равным 10, который присутствует в результирующих наборах обоих запросов. В спецификации SQL ANSI также описан оператор `except all`, но опять же он реализован только в DB2 Universal Server IBM.

В операторе `except all` есть небольшая хитрость. Вот пример, показывающий, как обрабатываются дублирующие данные. Скажем, есть два множества данных, имеющих следующий вид:

Множество A

```
+--------+
| emp_id |
+--------+
|     10 |
|     11 |
|     12 |
|     10 |
|     10 |
+--------+
```

Множество B

```
+--------+
| emp_id |
+--------+
|     10 |
|     10 |
+--------+
```

В результате операции `A except B` получаем следующее:

```
+--------+
| emp_id |
+--------+
|     11 |
|     12 |
+--------+
```

Если изменить операцию и применить `A except all B`, увидим следующее:

```
+--------+
| emp_id |
+--------+
|     10 |
|     11 |
```

```
|     12 |
+--------+
```

Следовательно, разница между этими двумя операциями в том, что `except` удаляет все экземпляры дублирующихся данных из множества A, тогда как `except all` удаляет из множества A только один экземпляр дубликата данных для каждого экземпляра дубликата данных множества B.

Правила операций с множествами

В следующих разделах обозначены некоторые правила, которых необходимо придерживаться при работе с составными запросами.

Результаты сортирующего составного запроса

Если требуется сортировать результаты составного запроса, после последнего входящего в него запроса можно добавить блок `order by`. В блоке `order by` указываются имена столбцов из первого запроса составного запроса. До сих пор в каждом примере главы имена столбцов в обоих запросах составного запроса совпадали, но так делать не обязательно, что и показывает следующий пример:

```
mysql> SELECT emp_id, assigned_branch_id
    -> FROM employee
    -> WHERE title = 'Teller'
    -> UNION
    -> SELECT open_emp_id, open_branch_id
    -> FROM account
    -> WHERE product_cd = 'SAV'
    -> ORDER BY emp_id;
+--------+--------------------+
| emp_id | assigned_branch_id |
+--------+--------------------+
|      1 |                  1 |
|      7 |                  1 |
|      8 |                  1 |
|      9 |                  1 |
|     10 |                  2 |
|     11 |                  2 |
|     12 |                  2 |
|     14 |                  3 |
|     15 |                  3 |
|     16 |                  4 |
|     17 |                  4 |
|     18 |                  4 |
+--------+--------------------+
12 rows in set (0.04 sec)
```

В этом примере в двух запросах заданы разные имена столбцов. Если в блоке `order by` указать имя столбца из второго запроса, будет получена следующая ошибка:

```
mysql> SELECT emp_id, assigned_branch_id
    -> FROM employee
    -> WHERE title = 'Teller'
    -> UNION
    -> SELECT open_emp_id, open_branch_id
    -> FROM account
    -> WHERE product_cd = 'SAV'
    -> ORDER BY open_emp_id;
ERROR 1054 (42S22): Unknown column 'open_emp_id' in 'order clause'
```

Чтобы избежать этой проблемы, рекомендуется в обоих запросах давать столбцам одинаковые псевдонимы.

Старшинство операций с множествами

Если в составном запросе больше двух запросов, использующих разные операторы работы с множествами, то для обеспечения желаемых результатов следует продумать порядок расположения этих запросов в составном выражении. Рассмотрим следующее составное выражение из трех запросов:

```
mysql> SELECT cust_id
    -> FROM account
    -> WHERE product_cd IN ('SAV', 'MM')
    -> UNION ALL
    -> SELECT a.cust_id
    -> FROM account a INNER JOIN branch b
    ->   ON a.open_branch_id = b.branch_id
    -> WHERE b.name = 'Woburn Branch'
    -> UNION
    -> SELECT cust_id
    -> FROM account
    -> WHERE avail_balance BETWEEN 500 AND 2500;
+---------+
| cust_id |
+---------+
|       1 |
|       2 |
|       3 |
|       4 |
|       8 |
|       9 |
|       7 |
|      11 |
|       5 |
+---------+
9 rows in set (0.00 sec)
```

Этот составной запрос включает три запроса, возвращающих набор неуникальных ID клиентов. Первые два запроса разделены оператором `union all`, а второй и третий – оператором `union`. Может показаться, что расположение операторов `union` и `union all` не играет роли, но на самом

деле разница есть. Вот тот же составной запрос, в котором эти операторы поменялись местами:

```
mysql> SELECT cust_id
    -> FROM account
    -> WHERE product_cd IN ('SAV', 'MM')
    -> UNION
    -> SELECT a.cust_id
    -> FROM account a INNER JOIN branch b
    ->   ON a.open_branch_id = b.branch_id
    -> WHERE b.name = 'Woburn Branch'
    -> UNION ALL
    -> SELECT cust_id
    -> FROM account
    -> WHERE avail_balance BETWEEN 500 AND 2500;
+---------+
| cust_id |
+---------+
|       1 |
|       2 |
|       3 |
|       4 |
|       8 |
|       9 |
|       7 |
|      11 |
|       1 |
|       1 |
|       2 |
|       3 |
|       3 |
|       4 |
|       4 |
|       5 |
|       9 |
+---------+
17 rows in set (0.00 sec)
```

При виде результатов становится очевидным, что порядок разных операторов работы с множествами в составном запросе *действительно* имеет значение. В общем, составные запросы из трех или больше запросов оцениваются в порядке сверху вниз, но с учетом следующих пояснений:

- По спецификации SQL ANSI из всех операторов работы с множествами первым выполняется оператор `intersect`.
- Порядок сочетания запросов можно задавать с помощью скобок.

Но поскольку в MySQL еще не реализованы ни оператор `intersect`, ни скобки в составных запросах, для получения нужного результата придется аккуратно расставлять запросы, образующие составной запрос. При использовании другого сервера БД, для переопределения порядка

обработки составных запросов по умолчанию (сверху вниз) запросы, расположенные рядом, можно заключить в скобки:

```
(SELECT cust_id
 FROM account
 WHERE product_cd IN ('SAV', 'MM')
 UNION ALL
 SELECT a.cust_id
 FROM account a INNER JOIN branch b
   ON a.open_branch_id = b.branch_id
 WHERE b.name = 'Woburn Branch')
INTERSECT
(SELECT cust_id
 FROM account
 WHERE avail_balance BETWEEN 500 AND 2500
 EXCEPT
 SELECT cust_id
 FROM account
 WHERE product_cd = 'CD'
  AND avail_balance < 1000);
```

Для этого составного запроса первый и второй запросы комбинируются оператором `union all`, затем третий и четвертый запросы – оператором `except`, и, наконец, для формирования окончательного результирующего набора результаты этих двух операций комбинируются с помощью оператора `intersect`.

Упражнения

Следующие упражнения призваны протестировать понимание операций с множествами. Ответы на эти упражнения приведены в приложении C.

6.1

Имеются множество `A = {L M N O P}` и множество `B = {P Q R S T}`. Какие множества будут получены в результате следующих операций:

- `A union B`
- `A union all B`
- `A intersect B`
- `A except B`

6.2

Напишите составной запрос для выбора имен и фамилий всех клиентов-физических лиц, а также имен и фамилий всех сотрудников.

6.3

Отсортируйте результаты упражнения 6.2 по столбцу `lname`.

7

Создание, преобразование и работа с данными

Как говорилось в предисловии, цель данной книги – показать универсальные методы SQL, применяемые на разных серверах БД. Однако эта глава посвящена созданию, преобразованию и работе со строковыми, числовыми и временными данными, а язык SQL не включает команды, обеспечивающие эти функциональные возможности. Вернее, все эти операции осуществляются встроенными функциями. К тому же, хотя стандарт SQL и описывает некоторые функции, производители БД часто отступают от их спецификаций.

Поэтому в данной главе сначала показаны некоторые общие приемы работы с данными в SQL-выражениях, а потом продемонстрированы отдельные встроенные функции, реализованные в Microsoft SQL Server, Oracle Database и MySQL. Кроме материала, представленного в этой главе, настоятельно рекомендуется приобрести справочное руководство с описанием всех функций, реализованных сервером, с которым вы работаете. Если вы работаете с несколькими серверами, есть ряд справочников, охватывающих несколько серверов, например «SQL in a Nutshell» или «SQL Pocket Guide», оба от издательства O'Reilly.

Строковые данные

При работе со строковыми данными используется один из следующих символьных типов данных:

`CHAR`

: Предназначен для хранения строк фиксированной длины, дополненных пробелами. MySQL допускает значения типа `CHAR` длиной до 255 символов, Oracle Database – до 2000 символов, а SQL Server – до 8000 символов.

`varchar`

Предназначен для хранения строк переменной длины. MySQL допускает в столбце типа `varchar` до 255 символов (65 535 символов для версии 5.0 и выше), Oracle Database (с помощью типа `varchar2`) – до 4000 символов, а SQL Server – до 8000 символов.

`text` (*MySQL и SQL Server*) или `CLOB` (*Character Large Object*; *Oracle Database*)

Позволяют хранить очень большие строки переменной длины (обычно в этом контексте их называют документами). В MySQL есть несколько текстовых типов (`tinytext`, `text`, `mediumtext` и `longtext`) для документов размером до 4 Гбайт. В SQL Server всего один тип `text` для документов размером до 2 Гбайт. Oracle Database включает тип данных `CLOB`, позволяющий хранить колоссальные документы до 128 Тбайт.

В некоторых примерах данного раздела, иллюстрирующих применение этих различных типов, я использую такую таблицу:

```
CREATE TABLE string_tbl
 (char_fld CHAR(30),
  vchar_fld VARCHAR(30),
  text_fld TEXT
 );
```

В следующих двух разделах показано, как создавать строковые данные и работать с ними.

Создание строк

Самый простой способ заполнить символьный столбец – заключить строку в кавычки:

```
mysql> INSERT INTO string_tbl (char_fld, vchar_fld, text_fld)
    -> VALUES ('This is char data',
    ->   'This is varchar data',
    ->   'This is text data');
Query OK, 1 row affected (0.00 sec)
```

При вставке строковых данных в таблицу не забывайте, что если длина строки превышает максимальный размер символьного столбца (или заданный, или допустимый максимум типа данных), сервер или сформирует исключение (Oracle Database), или, в случае MySQL или SQL Server, без лишнего шума усечет строку (MySQL генерирует предупреждение). Чтобы показать, как MySQL поведет себя в такой ситуации, следующее выражение `update` пытается обновить строкой из 46 символов столбец `vchar_fld`, для которого задан максимальный размер в 30 символов:

```
mysql> UPDATE string_tbl
    -> SET vchar_fld = 'This is a piece of extremely long varchar data';
```

```
Query OK, 1 row affected, 1 warning (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 1
```

Столбец изменен, но сформировано следующее предупреждение:

```
mysql> SHOW WARNINGS;
+---------+------+-----------------------------------------------+
| Level   | Code | Message                                       |
+---------+------+-----------------------------------------------+
| Warning | 1265 | Data truncated for column 'vchar_fld' at row 1 |
+---------+------+-----------------------------------------------+
1 row in set (0.00 sec)
```

Если извлечь столбец `vchar_fld`, то получим:

```
mysql> SELECT vchar_fld
    -> FROM string_tbl;
+-------------------------------+
| vchar_fld                     |
+-------------------------------+
| This is a piece of extremely l |
+-------------------------------+
1 row in set (0.05 sec)
```

Как видите, в столбце `vchar_fld` размещены только первые 30 символов 46-символьной строки. Лучший способ избежать усечения строки (или формирования исключений в случае Oracle Database) при работе со столбцами типа `varchar` – задавать достаточно большой верхний предел длины строки, чтобы иметь возможность работать с самыми длинными из предполагаемых для хранения строк (помня о том, что сервер распределяет для хранения строки лишь необходимое количество памяти, т. е. при задании большого верхнего предела для столбцов типа `varchar` память все же не расходуется впустую).

Одинарные кавычки (апострофы)

Поскольку строки разграничиваются одинарными кавычками, необходимо быть внимательными со строками, включающими одинарные кавычки (апострофы). Например, следующую строку вставить не получится, потому что сервер подумает, что апостроф в слове «doesn't» обозначает конец строки:

```
UPDATE string_tbl
SET text_fld = 'This string doesn't work';
```

Чтобы заставить сервер «проигнорировать» апостроф в слове «doesn't», понадобится добавить в строку *знак экранирования символа* (*escape*). Тогда сервер будет воспринимать апостроф как обычный символ строки. Все три сервера обеспечивают возможность сохранить апостроф; для этого надо ввести непосредственно перед апострофом еще один апостроф:

```
mysql> UPDATE string_tbl
    -> SET text_fld = 'This string didn''t work, but it does now';
```

```
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Пользователи Oracle Database и MySQL также могут сохранить одинарную кавычку, разместив прямо перед ней обратный слэш:

```
UPDATE string_tbl SET text_fld =
  'This string didn\'t work, but it does now'
```

При извлечении строки для отображения на экране или в поле сообщения ее внутренние кавычки не требуют какой-либо особой обработки:

```
mysql> SELECT text_fld
    -> FROM string_tbl;
+----------------------------------------+
| text_fld                               |
+----------------------------------------+
| This string didn't work, but it does now |
+----------------------------------------+
1 row in set (0.00 sec)
```

Однако если строка извлекается для помещения в файл, предназначенный для другой программы, возможно, вы захотите вставить в извлеченную строку знак экранирования символа. При работе с MySQL можно использовать встроенную функцию `quote()` (кавычка), которая заключает в кавычки всю строку *и* добавляет знаки экранирования символа к любой одинарной кавычке/апострофу, встречающейся в строке. Вот как выглядит строка, извлеченная с применением функции `quote()`:

```
mysql> SELECT QUOTE(text_fld)
    -> FROM string_tbl;
+-------------------------------------------+
| QUOTE(text_fld)                           |
+-------------------------------------------+
| 'This string didn\'t work, but it does now' |
+-------------------------------------------+
1 row in set (0.04 sec)
```

При извлечении данных с целью экспорта вы, возможно, захотите применить функцию `quote()` ко всем символьным столбцам, сформированным не системой, таким как столбец `customer_notes` (примечания клиента).

Специальные символы

Если приложение предполагается применять в разных странах, строки могут включать символы, которых нет на клавиатуре разработчика. Например, при работе с французским и немецким языками может понадобиться включать символы с диакритическими знаками, такие как e или ö. Серверы SQL Server и MySQL включают встроенную функцию `char()`, позволяющую создавать строки из всех 255 символов набора ASCII (пользователи Oracle Database могут применять функцию

chr()). Для примера следующий фрагмент кода извлекает напечатанную строку и ее эквивалент, собранный из отдельных символов:

```
mysql> SELECT 'abcdefg', CHAR(97,98,99,100,101,102,103);
+---------+-------------------------------+
| abcdefg | CHAR(97,98,99,100,101,102,103) |
+---------+-------------------------------+
| abcdefg | abcdefg                       |
+---------+-------------------------------+
1 row in set (0.01 sec)
```

Таким образом, 97-й символ набора символов ASCII – это буква a. Приведенные выше символы не являются специальными, а вот следующий пример показывает местоположение символов с диакритическими знаками и других специальных символов, таких как знаки валют:

```
mysql> SELECT CHAR(128,129,130,131,132,133,134,135,136,137);
+-----------------------------------------------+
| CHAR(128,129,130,131,132,133,134,135,136,137) |
+-----------------------------------------------+
| Çüéâäàåçêë                                    |
+-----------------------------------------------+
1 row in set (0.01 sec)

mysql> SELECT CHAR(138,139,140,141,142,143,144,145,146,147);
+-----------------------------------------------+
| CHAR(138,139,140,141,142,143,144,145,146,147) |
+-----------------------------------------------+
| èïîìÄÅÉæÆô                                    |
+-----------------------------------------------+
1 row in set (0.01 sec)

mysql> SELECT CHAR(148,149,150,151,152,153,154,155,156,157);
+-----------------------------------------------+
| CHAR(148,149,150,151,152,153,154,155,156,157) |
+-----------------------------------------------+
| öòûùÿ...Ü¢£¥                                  |
+-----------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT CHAR(158,159,160,161,162,163,164,165);
+---------------------------------------+
| CHAR(158,159,160,161,162,163,164,165) |
+---------------------------------------+
| ₧ƒáíóúñÑ                              |
+---------------------------------------+
1 row in set (0.01 sec)
```

В примерах данного раздела используется набор символов latin1. Если сеанс сконфигурирован под другой набор, вы увидите символы, отличные от приведенных здесь. Идея та же, но, чтобы находить определенные символы, вам придется освоить расположение символов своего набора.

Построение строк символ за символом может быть достаточно утомительным, особенно если в строке всего лишь несколько символов с диакритическими знаками. К счастью, можно воспользоваться функцией `concat()` и соединить отдельные строки, часть которых можно ввести с клавиатуры, а другие – сформировать с помощью функции `char()`. Например, следующий фрагмент кода показывает, как построить фразу *danke schön* с помощью функций `concat()` и `char()`:

```
mysql> SELECT CONCAT('danke sch', CHAR(148), 'n');
+-------------------------------------+
| CONCAT('danke sch', CHAR(148), 'n') |
+-------------------------------------+
| danke schön                         |
+-------------------------------------+
1 row in set (0.00 sec)
```

Пользователи Oracle Database вместо функции `concat()` могут применять оператор конкатенации (`||`):

```
SELECT 'danke sch' || CHR(148) || 'n'
    FROM dual;
```

В SQL Server нет функции `concat()`, поэтому придется использовать оператор конкатенации (`+`):

```
SELECT 'danke sch' + CHAR(148) + 'n'
```

Найти ASCII-эквивалент нужного символа можно с помощью функции `ascii()`, принимающей самый левый символ строки и возвращающей его номер:

```
mysql> SELECT ASCII('ö');
+------------+
| ASCII('ö') |
+------------+
|        148 |
+------------+
1 row in set (0.00 sec)
```

Функции `char()`, `ascii()` и `concat()` (как и операторы конкатенации) позволяют работать с любым романским языком, даже если клавиатура не включает символы с диакритическими знаками или спецсимволы.

Работа со строками

Каждый сервер БД включает множество встроенных функций для работы со строками. В данном разделе будут рассмотрены строковые функции двух типов: возвращающие числа и строки. Однако прежде чем начать, возвратим данные таблицы `string_tbl` к исходному состоянию:

```
mysql> DELETE FROM string_tbl;
Query OK, 1 row affected (0.02 sec)

mysql> INSERT INTO string_tbl (char_fld, vchar_fld, text_fld)
```

```
    -> VALUES ('This string is 28 characters',
    ->   'This string is 28 characters',
    ->   'This string is 28 characters');
Query OK, 1 row affected (0.00 sec)
```

Строковые функции, возвращающие числа

Одна из наиболее широко используемых строковых функций, возвращающих числа, – функция `length()` (длина), которая возвращает число символов в строке (пользователям SQL Server придется использовать функцию `len()`). В следующем запросе функция `length()` применяется к каждому столбцу таблицы `string_tbl`:

```
mysql> SELECT LENGTH(char_fld) char_length,
    ->   LENGTH(vchar_fld) varchar_length,
    ->   LENGTH(text_fld) text_length
    -> FROM string_tbl;
+-------------+----------------+-------------+
| char_length | varchar_length | text_length |
+-------------+----------------+-------------+
|          28 |             28 |          28 |
+-------------+----------------+-------------+
1 row in set (0.00 sec)
```

Длины столбцов `varchar` и `text` вполне ожидаемы, но предполагалось, что столбец `char` содержит строки длиной 30 символов, – я ведь говорил, что строки, хранящиеся в столбцах типа `char`, дополняются пробелами справа. Но сервер MySQL удаляет пробелы в конце строки при извлечении данных типа `char`, поэтому для всех строковых функций, независимо от типа столбца, хранящего строки, получены аналогичные результаты.

Кроме определения длины строки может потребоваться найти местоположение подстроки в строке. Например, если надо определить, где в столбце `vchar_fld` располагается строка «characters», можно воспользоваться функцией `position()` (положение), как показано ниже:

```
mysql> SELECT POSITION('characters' IN vchar_fld)
    -> FROM string_tbl;
+-------------------------------------+
| POSITION('characters' IN vchar_fld) |
+-------------------------------------+
|                                  19 |
+-------------------------------------+
1 row in set (0.12 sec)
```

Если не получается найти подстроку, функция `position()` возвращает 0.

Программисты на таких языках, как C или C++, в которых первый элемент массива имеет порядковый номер 0, при работе с базами данных должны помнить, что порядковый номер первого символа строки равен 1. Если функция `position()` возвращает значение 0, это указывает на то, что подстрока не найдена, а не на то, что подстрока обнаружена в строке на первой позиции.

Если требуется начать поиск не с первого символа целевой строки, необходимо использовать функцию `locate()`, аналогичную функции `position()` за тем исключением, что допускает третий необязательный параметр, предназначенный для задания стартовой позиции поиска. И еще функция `locate()` является собственной функцией производителей БД, тогда как `position()` – часть стандарта SQL:2003. Вот пример запроса позиции строки `'is'`, начинающего поиск с пятого символа столбца `vchar_fld`:

```
mysql> SELECT LOCATE('is', vchar_fld, 5)
    -> FROM string_tbl;
+-----------------------------+
| LOCATE('is', vchar_fld, 5) |
+-----------------------------+
|                          13 |
+-----------------------------+
1 row in set (0.02 sec)
```

В Oracle Database нет функции `position()` или `locate()`, но есть функция `instr()`, которая воспроизводит функцию `position()`, если задано два аргумента, и функцию `locate()`, если задано три аргумента. В SQL Server тоже нет функции `position()` или `locate()`, но есть функция `charindx()`, которая также принимает два или три аргумента аналогично функции `instr()` Oracle.

Еще одна функция, принимающая строки в качестве аргументов и возвращающая числа, – функция сравнения строк `strcmp()`. `Strcmp()`, которая реализована только в MySQL и не имеет аналогов в Oracle Database или SQL Server. Она принимает в качестве аргументов две строки и возвращает одно из следующих значений:

`-1` если первая строка в порядке сортировки расположена до второй строки

`0` если строки идентичны

`1` если первая строка в порядке сортировки расположена после второй строки

Чтобы проиллюстрировать работу этой функции, сначала покажем с помощью запроса порядок сортировки пяти строк, а затем проведем сравнение строк с помощью функции `strcmp()`. Вот пять строк, которые будут вставлены в таблицу `string_tbl`:

```
mysql> DELETE FROM string_tbl;
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('abcd');
Query OK, 1 row affected (0.03 sec)

mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('xyz');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('QRSTUV');
Query OK, 1 row affected (0.00 sec)
```

```
mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('qrstuv');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO string_tbl(vchar_fld) VALUES ('12345');
Query OK, 1 row affected (0.00 sec)
```

Вот эти пять строк в порядке сортировки:

```
mysql> SELECT vchar_fld
    -> FROM string_tbl
    -> ORDER BY vchar_fld;
+-----------+
| vchar_fld |
+-----------+
| 12345     |
| abcd      |
| QRSTUV    |
| qrstuv    |
| xyz       |
+-----------+
5 rows in set (0.00 sec)
```

Следующий запрос проводит шесть сравнений пяти разных строк:

```
mysql> SELECT STRCMP('12345','12345') 12345_12345,
    ->   STRCMP('abcd','xyz') abcd_xyz,
    ->   STRCMP('abcd','QRSTUV') abcd_QRSTUV,
    ->   STRCMP('qrstuv','QRSTUV') qrstuv_QRSTUV,
    ->   STRCMP('12345','xyz') 12345_xyz,
    ->   STRCMP('xyz','qrstuv') xyz_qrstuv;
+-------------+----------+-------------+---------------+-----------+------------+
| 12345_12345 | abcd_xyz | abcd_QRSTUV | qrstuv_QRSTUV | 12345_xyz | xyz_qrstuv |
+-------------+----------+-------------+---------------+-----------+------------+
|           0 |       -1 |          -1 |             0 |        -1 |          1 |
+-------------+----------+-------------+---------------+-----------+------------+
1 row in set (0.00 sec)
```

В результате первого сравнения получаем вполне ожидаемое значение `0`, поскольку строка сравнивалась сама с собой. Четвертое сравнение также дает `0`, что немного неожиданно, поскольку строки состоят из одних и тех же букв, но в одной строке все буквы в верхнем регистре, а в другой – в нижнем. Причиной такого результата является то, что функция `strcmp()` MySQL не чувствительна к регистру, о чем надо помнить при ее использовании. Остальные четыре сравнения дают или `-1`, или `1` в зависимости от порядка расположения строк в порядке сортировки. Например, `strcmp('abcd','xyz')` дает `-1`, поскольку строка `'abcd'` идет перед строкой `'xyz'`.

Наряду с функцией `strcmp()` MySQL позволяет использовать в блоке `select` операторы `like` и `regexp` для сравнения строк. Результатом таких сравнений будет `1` (для `true`) и `0` (для `false`). Следовательно, эти операторы позволяют создавать выражения, возвращающие число, подобно функциям, описанным в этом разделе. Вот пример использования оператора `like`:

```
mysql> SELECT name, name LIKE '%ns' ends_in_ns
    -> FROM department;
+----------------+------------+
| name           | ends_in_ns |
+----------------+------------+
| Operations     |          1 |
| Loans          |          1 |
| Administration |          0 |
+----------------+------------+
3 rows in set (0.25 sec)
```

В этом примере выбираются все названия отделов. Также есть выражение, возвращающее 1, если название отдела заканчивается на «ns» или 0 в противном случае. Для поиска совпадений по более сложному шаблону можно использовать оператор regexp, как показано ниже:

```
mysql> SELECT cust_id, cust_type_cd, fed_id,
    ->   fed_id REGEXP '.{3}-.{2}-.{4}' is_ss_no_format
    -> FROM customer;
+---------+--------------+-------------+-----------------+
| cust_id | cust_type_cd | fed_id      | is_ss_no_format |
+---------+--------------+-------------+-----------------+
|       1 | I            | 111-11-1111 |               1 |
|       2 | I            | 222-22-2222 |               1 |
|       3 | I            | 333-33-3333 |               1 |
|       4 | I            | 444-44-4444 |               1 |
|       5 | I            | 555-55-5555 |               1 |
|       6 | I            | 666-66-6666 |               1 |
|       7 | I            | 777-77-7777 |               1 |
|       8 | I            | 888-88-8888 |               1 |
|       9 | I            | 999-99-9999 |               1 |
|      10 | B            | 04-1111111  |               0 |
|      11 | B            | 04-2222222  |               0 |
|      12 | B            | 04-3333333  |               0 |
|      13 | B            | 04-4444444  |               0 |
+---------+--------------+-------------+-----------------+
13 rows in set (0.00 sec)
```

Четвертый столбец этого запроса возвращает 1, если значение в столбце fed_id соответствует формату номера социальной страховки.

Пользователи SQL Server и Oracle Database могут получить аналогичные результаты с помощью выражений case, описанных в главе 11.

Строковые функции, возвращающие строки

В некоторых случаях требуется изменить имеющиеся строки – удалить/добавить текстовый фрагмент. Для выполнения этих задач у каждого сервера БД есть множество функций. Прежде чем начать, еще раз возвратим данные таблицы string_tbl в исходное состояние:

```
mysql> DELETE FROM string_tbl;
Query OK, 5 rows affected (0.00 sec)

mysql> INSERT INTO string_tbl (text_fld)
    -> VALUES ('This string was 29 characters');
Query OK, 1 row affected (0.01 sec)
```

Ранее в этой главе было описано построение слов, включающих символы с диакритическими знаками, с помощью функции `concat()`. Функция `concat()` полезна и во многих других ситуациях, например, если требуется добавить в конец хранящейся строки дополнительные символы. В следующем примере строка в столбце `text_fld` изменяется путем добавления в ее конец дополнительной фразы:

```
mysql> UPDATE string_tbl
    -> SET text_fld = CONCAT(text_fld, ', but now it is longer');
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Теперь столбец `text_fld` содержит следующую строку:

```
mysql> SELECT text_fld
    -> FROM string_tbl;
+-----------------------------------------------------+
| text_fld                                            |
+-----------------------------------------------------+
| This string was 29 characters, but now it is longer |
+-----------------------------------------------------+
1 row in set (0.00 sec)
```

Таким образом, как и все функции, возвращающие строку, `concat()` можно использовать для замещения данных, хранящихся в столбце символьного типа.

Другое традиционное применение функции `concat()` – построение строки из отдельных частей данных. Например, следующий запрос формирует строку примечания для каждого операциониста банка:

```
mysql> SELECT CONCAT(fname, ' ', lname, ' has been a ',
    ->   title, ' since ', start_date) emp_narrative
    -> FROM employee
    -> WHERE title = 'Teller' OR title = 'Head Teller';
+-----------------------------------------------------------+
| emp_narrative                                             |
+-----------------------------------------------------------+
| Helen Fleming has been a Head Teller since 2004-03-17     |
| Chris Tucker has been a Teller since 2004-09-15           |
| Sarah Parker has been a Teller since 2002-12-02           |
| Jane Grossman has been a Teller since 2002-05-03          |
| Paula Roberts has been a Head Teller since 2002-07-27     |
| Thomas Ziegler has been a Teller since 2000-10-23         |
| Samantha Jameson has been a Teller since 2003-01-08       |
| John Blake has been a Head Teller since 2000-05-11        |
| Cindy Mason has been a Teller since 2002-08-09            |
```

```
| Frank Portman has been a Teller since 2003-04-01        |
| Theresa Markham has been a Head Teller since 2001-03-15 |
| Beth Fowler has been a Teller since 2002-06-29          |
| Rick Tulman has been a Teller since 2002-12-12          |
+---------------------------------------------------------+
13 rows in set (0.12 sec)
```

Функция `concat()` может обрабатывать любое выражение, возвращающее строку, и даже преобразует числа и даты в строковый формат, о чем свидетельствует столбец дат (`start_date`), используемый как аргумент. Хотя Oracle Database включает функцию `concat()`, она может принимать только строковые аргументы, поэтому в Oracle предыдущий запрос работать не будет. В этом случае придется использовать оператор конкатенации (`||`), а не вызов функции:

```
SELECT fname || ' ' || lname || ' has been a ' ||
  title || ' since ' || start_date emp_narrative
FROM employee
WHERE title = 'Teller' OR title = 'Head Teller';
```

В SQL Server нет функции `concat()`, поэтому используется такой же подход, что и в предыдущем примере, только с применением оператора конкатенации SQL Server (`+`, а не `||`).

Функция `concat()` полезна для добавления символов в начало или конец строки, но также позволяет ввести или заменить символы в *середине* строки. Все три сервера БД предоставляют специальные функции для этого, но все они разные, поэтому сначала рассмотрим функцию MySQL, а затем перейдем к функциям двух других серверов.

MySQL включает функцию `insert()`, которая принимает четыре аргумента: исходную строку, начальное положение, число символов, требующих замены, и замещающую строку. В зависимости от значения третьего аргумента функция выполняет вставку либо замену символов строки. Если третий аргумент равен нулю, то замещающая строка вставляется со сдвигом всех последующих символов вправо, например:

```
mysql> SELECT INSERT('goodbye world', 9, 0, 'cruel ') string;
+---------------------+
| string              |
+---------------------+
| goodbye cruel world |
+---------------------+
1 row in set (0.00 sec)
```

В этом примере все символы, начиная с девятого, сдвигаются вправо, и вставляется строка `'cruel '`. Если третий аргумент больше нуля, то замещающая строка замещает указанное количество символов, например:

```
mysql> SELECT INSERT('goodbye world', 1, 7, 'hello') string;
+-------------+
```

```
| string      |
+-------------+
| hello world |
+-------------+
1 row in set (0.00 sec)
```

В этом примере первые семь символов замещаются строкой `'hello'`. В Oracle Database нет единой функции, обладающей гибкостью `insert()` MySQL, но в Oracle есть функция `replace()`, замещающая одну подстроку другой. Вот предыдущий пример, переработанный с использованием `replace()`:

```
SELECT REPLACE('goodbye world', 'goodbye', 'hello')
FROM dual;
```

Все экземпляры строки `'goodbye'` будут замещены строкой `'hello'`. В результате получаем строку `'hello world'`. Функция заместит *все* экземпляры искомой строки замещающей строкой – будьте внимательны, чтобы не получить больше замещений, чем задумано.

В SQL Server есть и функция `replace()` с той же функциональностью, что и в Oracle, а также функция `stuff()` (заполнить), функциональные возможности которой аналогичны функции `insert()` MySQL. Вот пример:

```
SELECT STUFF('hello world', 1, 5, 'goodbye cruel')
```

Во время выполнения этого запроса удаляются пять символов, начиная с первой позиции, и на их место вставляется строка `'goodbye cruel'`. В результате получаем строку `'goodbye cruel world'`.

Кроме вставки символов в строку может понадобиться *извлечь* из строки подстроку. Для этого все три сервера включают функцию `substring()` (подстрока) (правда, в Oracle Database эта функция называется `substr()`). Она извлекает указанное число символов, начиная с заданной позиции. В следующем примере из строки извлекается пять символов, начиная с девятой позиции:

```
mysql> SELECT SUBSTRING('goodbye cruel world', 9, 5);
+---------------------------------------+
| SUBSTRING('goodbye cruel world', 9, 5) |
+---------------------------------------+
| cruel                                 |
+---------------------------------------+
1 row in set (0.00 sec)
```

Кроме упомянутых, все три сервера включают множество других встроенных функций для работы со строковыми данными. Хотя назначение многих из них сугубо специальное, например формирование строкового эквивалента восьмеричных или шестнадцатеричных чисел, есть и функции общего назначения, например удаляющие или добавляющие пробелы в конце текстовой строки. Более подробную информацию можно получить в справочном руководстве по SQL для кон-

кретного сервера или универсальном справочнике по SQL, например «SQL in a Nutshell» (O'Reilly).

Числовые данные

В отличие от строковых данных (и временных, как вы вскоре увидите) числовые данные довольно просты. Число можно ввести с клавиатуры, извлечь из другого столбца или сформировать с помощью вычисления. Для вычислений доступны все обычные арифметические операторы (+, -, *, /), а для задания порядка вычислений – скобки:

```
mysql> SELECT (37 * 59) / (78 - (8 * 6));
+----------------------------+
| (37 * 59) / (78 - (8 * 6)) |
+----------------------------+
|                      72.77 |
+----------------------------+
1 row in set (0.00 sec)
```

Как упоминалось в главе 2, основная проблема при хранении числовых данных – возможное округление (иногда очень грубое) при превышении предела, заданного для числового столбца. Например, число 999,99 при сохранении в столбце, определенном как `float(3,1)`, будет округлено до 99,9.

Выполнение арифметических операций

Большинство встроенных числовых функций предназначено для выполнения определенных арифметических операций, таких как вычисление квадратного корня числа. В табл. 7.1 перечислены некоторые распространенные числовые функции, принимающие один числовой аргумент и возвращающие число.

Таблица 7.1. Одноаргументные числовые функции

Функция	Описание
`Acos(`*x*`)`	Вычисляет арккосинус *x*
`Asin(`*x*`)`	Вычисляет арксинус *x*
`Atan(`*x*`)`	Вычисляет арктангенс *x*
`Cos(`*x*`)`	Вычисляет косинус *x*
`Cot(`*x*`)`	Вычисляет котангенс *x*
`Exp(`*x*`)`	Вычисляет e^x
`Ln(`*x*`)`	Вычисляет натуральный логарифм *x*
`Sin(`*x*`)`	Вычисляет синус *x*
`Sqrt(`*x*`)`	Вычисляет квадратный корень из *x*
`Tan(`*x*`)`	Вычисляет тангенс *x*

Эти функции осуществляют очень специальные задачи. Не будем приводить здесь примеры для них (если читатель не узнаёт функцию по названию или описанию, то, скорее всего, она ему не нужна). Однако другие числовые функции, используемые в вычислениях, чуть более гибки и заслуживают некоторого пояснения.

Например, оператор `modulo`, вычисляющий остаток от деления одного числа на другое, реализован в MySQL и Oracle Database функцией `mod()`. В следующем примере вычисляется остаток от деления `10` на `4`:

```
mysql> SELECT MOD(10,4);
+-----------+
| MOD(10,4) |
+-----------+
|         2 |
+-----------+
1 row in set (0.02 sec)
```

Обычно функция `mod()` используется с целыми аргументами, но в MySQL 4.1.7 и более поздних версий допустимы и вещественные аргументы:

```
mysql> SELECT MOD(22.75, 5);
+---------------+
| MOD(22.75, 5) |
+---------------+
|          2.75 |
+---------------+
1 row in set (0.02 sec)
```

В SQL Server нет функции `mod()`. Вместо нее для нахождения остатка используется оператор `%`. Следовательно, выражение `10 % 4` дает в результате значение `2`.

Другая числовая функция, принимающая два числовых аргумента, – функция `pow()` (в Oracle Database или SQL Server – `power()`), которая возвращает первое число в степени, равной второму числу, например:

```
mysql> SELECT POW(2,8);
+----------+
| POW(2,8) |
+----------+
|      256 |
+----------+
1 row in set (0.03 sec)
```

Таким образом, `pow(2,8)` – эквивалент MySQL для записи 2^8. Поскольку память компьютера распределена блоками по 2^x байт, с помощью функции `pow()` может быть удобно определять точное число байт в памяти определенного объема:

```
mysql> SELECT POW(2,10) kilobyte, POW(2,20) megabyte,
    ->   POW(2,30) gigabyte, POW(2,40) terabyte;
```

```
+----------+----------+------------+---------------+
| kilobyte | megabyte | gigabyte   | terabyte      |
+----------+----------+------------+---------------+
|     1024 |  1048576 | 1073741824 | 1099511627776 |
+----------+----------+------------+---------------+
1 row in set (0.00 sec)
```

Не знаю как вам, но мне проще запомнить гигабайт как 2^{30} байт, а не как число 1 073 741 824.

Управление точностью числовых данных

Числа с плавающей точкой не всегда обязаны взаимодействовать или отображаться полностью. Например, можно хранить данные о денежных операциях с точностью до шести десятичных разрядов, но при отображении округлять их до сотых. Для ограничения точности чисел с плавающей точкой предназначены четыре функции – `ceil()`, `floor()`, `round()` и `truncate()`. Все три сервера включают эти функции, только Oracle Database использует `trunc()` вместо `truncate()`, а SQL Server – `ceiling()` вместо `ceil()`.

Функции `ceil()` (потолок) и `floor()` (пол) предназначены для округления вверх или вниз до ближайшего целого, как показано в следующем примере:

```
mysql> SELECT CEIL(72.445), FLOOR(72.445);
+--------------+---------------+
| CEIL(72.445) | FLOOR(72.445) |
+--------------+---------------+
|           73 |            72 |
+--------------+---------------+
1 row in set (0.06 sec)
```

Как видите, любое число в диапазоне между 72 и 73 округляется до 73 (функция `ceil()`) или до 72 (функция `floor()`). Необходимо помнить, что `ceil()` округлит до 73, даже если десятичная часть числа очень мала, и `floor()` округлит до 72, даже если десятичная часть достаточно велика:

```
mysql> SELECT CEIL(72.000000001), FLOOR(72.999999999);
+--------------------+---------------------+
| CEIL(72.000000001) | FLOOR(72.999999999) |
+--------------------+---------------------+
|                 73 |                  72 |
+--------------------+---------------------+
1 row in set (0.00 sec)
```

Если предыдущие функции предлагают округления, слишком грубые для приложения, можно использовать функцию `round()` (округлить). Она округляет в большую или меньшую сторону от *середины* промежутка между двумя целыми, например:

```
mysql> SELECT ROUND(72.49999), ROUND(72.5), ROUND(72.50001);
+-----------------+-------------+-----------------+
| ROUND(72.49999) | ROUND(72.5) | ROUND(72.50001) |
+-----------------+-------------+-----------------+
|              72 |          72 |              73 |
+-----------------+-------------+-----------------+
1 row in set (0.00 sec)
```

При использовании функции round() любое число, десятичная часть которого относится к верхней половине диапазона между двумя целыми, округляется с увеличением, а если его десятичная часть относится к нижней половине диапазона, то выполняется округление с уменьшением.

Чаще всего требуется не округлять число до ближайшего целого, а сохранить, по крайней мере, несколько разрядов его десятичной части. Функция round() допускает необязательный второй аргумент, задающий число разрядов справа от десятичной точки, до которого проводится округление. Следующий пример показывает, как можно использовать второй аргумент для округления числа 72,0909 до первого, второго и третьего десятичного знака:

```
mysql> SELECT ROUND(72.0909, 1), ROUND(72.0909, 2), ROUND(72.0909, 3);
+------------------+------------------+------------------+
| ROUND(72.0909, 1) | ROUND(72.0909, 2) | ROUND(72.0909, 3) |
+------------------+------------------+------------------+
|             72.1 |            72.09 |           72.091 |
+------------------+------------------+------------------+
1 row in set (0.00 sec)
```

Как и функция round(), функция truncate() допускает необязательный второй аргумент, задающий число разрядов справа от десятичной точки, при этом truncate() просто отбрасывает лишние разряды без округления. Пример показывает, как было бы усечено число 72,0909 до одного, двух и трех десятичных знаков:

```
mysql> SELECT TRUNCATE(72.0909, 1), TRUNCATE(72.0909, 2),
    ->   TRUNCATE(72.0909, 3);
+----------------------+----------------------+----------------------+
| TRUNCATE(72.0909, 1) | TRUNCATE(72.0909, 2) | TRUNCATE(72.0909, 3) |
+----------------------+----------------------+----------------------+
|                 72.0 |                72.09 |               72.090 |
+----------------------+----------------------+----------------------+
1 row in set (0.00 sec)
```

В SQL Server нет функции truncate(). Ее роль играет функция round(), допуская третий необязательный аргумент; если он присутствует и отличен от нуля, выполняется усечение, а не округление числа.

Обе функции, truncate() и round(), также допускают отрицательное значение второго аргумента, означающее усечение или округление числа

слева от десятичной точки. На первый взгляд эта возможность может показаться странной, но для ее наличия есть веские основания. Например, есть продукт, закупка которого возможна только в количестве, пропорциональном десяти. Если покупатель закажет 17 единиц, то изменить заказанное количество можно одним из следующих способов:

```
mysql> SELECT ROUND(17, -1), TRUNCATE(17, -1);
+---------------+------------------+
| ROUND(17, -1) | TRUNCATE(17, -1) |
+---------------+------------------+
|            20 |               10 |
+---------------+------------------+
1 row in set (0.00 sec)
```

Если рассматриваемый продукт – канцелярские кнопки, то, скорее всего, итоговая сумма не сильно зависит от того, продано 10 или 20 штук при запрошенных 17; однако если речь идет о часах Rolex, для процветания бизнеса лучше будет округлять.

Обработка данных со знаком

При работе с числовыми столбцами, допускающими отрицательные значения (в главе 2 было показано, как сделать столбец беззнаковым, т. е. допускающим только положительные числа), могут быть полезными несколько числовых функций. Скажем, требуется составить отчет о текущем состоянии всех банковских счетов. Следующий запрос возвращает три столбца, помогающих сформировать отчет:

```
mysql> SELECT account_id, SIGN(avail_balance), ABS(avail_balance)
    -> FROM account;
+------------+---------------------+--------------------+
| account_id | SIGN(avail_balance) | ABS(avail_balance) |
+------------+---------------------+--------------------+
|          1 |                   1 |            1057.75 |
|          2 |                   1 |             500.00 |
|          3 |                   1 |            3000.00 |
|          4 |                   1 |            2258.02 |
|          5 |                   1 |             200.00 |
| ...                                                   |
|         19 |                   1 |            1500.00 |
|         20 |                   1 |           23575.12 |
|         21 |                   0 |               0.00 |
|         22 |                   1 |            9345.55 |
|         23 |                   1 |           38552.05 |
|         24 |                   1 |           50000.00 |
+------------+---------------------+--------------------+
24 rows in set (0.00 sec)
```

Второй столбец использует функцию `sign()` (знак), возвращающую: `-1`, если баланс счета отрицателен, `0`, если баланс нулевой, и `1`, если баланс положительный. С помощью функции `abs()` в третьем столбце возвращается абсолютное значение баланса.

Временные данные

Из трех типов данных, обсуждаемых в этой главе (символьные, числовые и временные), временные данные – наиболее сложные с точки зрения создания и обработки. Сложность временных данных отчасти обусловлена бесконечным множеством способов описания дат и времени. Например, дату написания этого абзаца можно записать любым из следующих способов:

- суббота, 19 марта 2005
- 3/19/2005 2:14:56 P.M. EST
- 3/19/2005 19:14:56 GMT
- 0782005 (Юлианский формат)
- Звездная дата [-4] 82213.47 14:14:56 (формат фильма «Звездный путь»)

Хотя некоторые записи отличаются только форматированием, основная сложность заключается в избранной системе отсчета, что рассматривается в следующем разделе.

Часовые пояса

Поскольку всюду на планете люди считают полднем наивысшую точку подъема солнца над горизонтом, никто и не пытался ввести одни универсальные часы на всех. Вместо этого мир был разделен на 24 воображаемые секции, названные *часовыми поясами* (*time zones*). В рамках одного часового пояса все придерживаются текущего времени, а в другом поясе люди живут по другому времени. Все выглядит достаточно просто, но одни географические регионы переводят свое время на час дважды в год (реализуя нак называемое *декретное время* (*Daylight Savings Time*)), а другие – нет. Таким образом, разница во времени между двумя точками планеты может первые полгода составлять четыре часа, а вторые – пять часов. Даже в рамках одного часового пояса одни регионы могут принимать, а другие не принимать декретное время. Поэтому в одном часовом поясе полгода время может совпадать, а полгода отличаться на час.

Компьютерная эра обострила эту проблему, хотя различия часовых поясов известны со времен великих географических открытий. Чтобы обеспечить общую точку отсчета для хронометрирования, мореплаватели XV столетия устанавливали свои часы по Гринвичу (Англия). Это время назвали *временем по Гринвичу* (*Greenwich Mean Time*), или GMT. Все остальные часовые пояса можно описать разностью между GMT и местным временем. Например, часовой пояс восточных штатов Америки, известный как *восточное поясное время* (*Eastern Standard Time*), можно описать как GMT-5:00 (на пять часов раньше GMT).

Сегодня используется разновидность GMT – *универсальное глобальное время* (*coordinated universal time*), или UTC, отсчитываемое по атом-

ным часам (или точнее среднее время 200 атомных часов, размещенных в 50 точках по всему миру, которое называют *всемирным временем* (*universal time*)). И SQL Server, и MySQL предоставляют функции, возвращающие текущее время UTC (`getutcdate()` для SQL Server и `utc_timestamp()` для MySQL).

Большинство серверов БД по умолчанию используют настройки часового пояса сервера, на котором размещены, и предоставляют инструменты для изменения часового пояса в случае необходимости. Например БД, предназначенная для хранения фондовых операций со всего света, обычно конфигурируется на использование времени UTC, тогда как БД для хранения операций конкретного предприятия розничной торговли может использовать часовой пояс сервера.

MySQL придерживается двух разных настроек часового пояса – глобальный часовой пояс и сеансовый часовой пояс, который может отличаться для каждого зарегистрированного пользователя. Следующий запрос позволяет увидеть обе настройки:

```
mysql> SELECT @@global.time_zone, @@session.time_zone;
+--------------------+---------------------+
| @@global.time_zone | @@session.time_zone |
+--------------------+---------------------+
| SYSTEM             | SYSTEM              |
+--------------------+---------------------+
1 row in set (0.00 sec)
```

Значение `system` (система) сообщает, что сервер использует настройку часового пояса сервера, на котором установлена БД.

Если пользователь находится в Цюрихе (Швейцария) и по сети открывает сеанс на сервере MySQL, расположенном в Нью-Йорке, вероятно, он захочет изменить часовой пояс для своего сеанса и может сделать это посредством следующей команды:

```
mysql> SET time_zone = 'Europe/Zurich';
Query OK, 0 rows affected (0.18 sec)
```

Если снова проверить настройки часового пояса, увидим следующее:

```
mysql> SELECT @@global.time_zone, @@session.time_zone;
+--------------------+---------------------+
| @@global.time_zone | @@session.time_zone |
+--------------------+---------------------+
| SYSTEM             | Europe/Zurich       |
+--------------------+---------------------+
1 row in set (0.00 sec)
```

Все даты, отображаемые в сеансе, теперь будут соответствовать Цюрихскому времени.

Пользователи Oracle Database могут изменить настройку часового пояса для сеанса следующей командой:

```
ALTER SESSION TIMEZONE = 'Europe/Zurich'
```

Создание временных данных

Временные данные могут быть сформированы любым из следующих средств:

- Копирование даты из имеющегося столбца типа `date`, `datetime` или `time`
- Выполнение встроенной функции, возвращающей значение типа `date`, `datetime` или `time`
- Создание строкового представления временных данных, которое потом преобразовывается сервером

Для применения последнего метода необходимо понимать различные компоненты, используемые при форматировании дат.

Строковые представления временных данных

Напомним самые популярные компоненты дат:

Таблица 7.2. Компоненты формата даты

Компонент	Описание	Диапазон
YYYY	Год, включая столетие	от 1000 до 9999
MM	Месяц	от 01 (январь) до 12 (декабрь)
DD	День	от 01 до 31
HH	Час	от 00 до 23
HHH	Часы (прошедшие)	от –838 до 838
MI	Минута	от 00 до 59
SS	Секунда	от 00 до 59

Чтобы создать строку, которая может быть интерпретирована сервером как тип `date`, `datetime` или `time`, необходимо свести различные компоненты вместе в порядке, показанном в табл. 7.3.

Таблица 7.3. Обязательные компоненты дат

Тип	Формат по умолчанию
`Date`	YYYY-MM-DD
`Datetime`	YYYY-MM-DD HH:MI:SS
`Timestamp`	YYYY-MM-DD HH:MI:SS
`Time`	HHH:MI:SS

Загрузка данных часового пояса MySQL

Если сервер MySQL выполняется на платформе Windows, то прежде чем настраивать глобальные или сеансовые часовые пояса, пользователю необходимо загрузить данные часовых поясов вручную. Для этого надо сделать следующее:

1. Скачать данные часового пояса по адресу *http://dev.mysql.com/downloads/timezones.html*.
2. Остановить сервер MySQL.
3. Извлечь файлы из загруженного zip-файла (в моем случае этот файл назывался *timezone-2004e.zip*) и поместить его в подкаталог */data/mysql* каталога установки MySQL (полный путь для моей установки был */Program Files/MySQL/MySQL Server 4.1/data/mysql*).
4. Вновь запустить сервер MySQL.

Чтобы посмотреть данные часового пояса, необходимо перейти к базе данных `mysql` с помощью команды `use mysql` и выполнить следующий запрос:

```
mysql> SELECT name FROM time_zone_name;
+----------------------------------+
| name                             |
+----------------------------------+
| Africa/Abidjan                   |
| Africa/Accra                     |
| Africa/Addis_Ababa               |
| Africa/Algiers                   |
| Africa/Asmera                    |
| Africa/Bamako                    |
| Africa/Bangui                    |
| Africa/Banjul                    |
| Africa/Bissau                    |
| Africa/Blantyre                  |
| Africa/Brazzaville               |
| Africa/Bujumbura                 |
...
| US/Alaska                        |
| US/Aleutian                      |
| US/Arizona                       |
| US/Central                       |
| US/East-Indiana                  |
| US/Eastern                       |
| US/Hawaii                        |
| US/Indiana-Starke                |
| US/Michigan                      |
| US/Mountain                      |
| US/Pacific                       |
```

```
| US/Samoa                          |
| UTC                               |
| W-SU                              |
| WET                               |
| Zulu                              |
+-----------------------------------+
545 rows in set (0.01 sec)
```

Чтобы изменить настройки своего часового пояса, надо выбрать одно из имен из предыдущего запроса, наиболее соответствующее вашему местоположению.

Таким образом, чтобы заполнить столбец типа `datetime` показаниями времени 3:30 после полудня 27 марта 2005 года, понадобиться создать следующую строку:

```
'2005-03-27 15:30:00'
```

Если сервер ожидает значение `datetime`, как при обновлении столбца типа `datetime` или при вызове встроенной функции, принимающей аргумент `datetime`, можно предоставить правильно форматированную строку с необходимыми компонентами даты, и сервер выполнит преобразование сам. Вот, например, выражение для корректировки даты банковской операции:

```
UPDATE transaction
SET txn_date = '2005-03-27 15:30:00'
WHERE txn_id = 99999;
```

Сервер определяет, что предоставленная строка блока `set` должна быть значением типа `datetime`, поскольку строка используется для заполнения столбца типа `datetime`. Следовательно, сервер попытается преобразовать эту строку, разбирая ее на шесть компонентов (год, месяц, день, час, минута, секунда), включенные в формат `datetime` по умолчанию.

Преобразования строки в дату

Если сервер *не* ожидает значение типа `datetime`, необходимо указать ему преобразовать строку в тип `datetime`. Вот, например, простой запрос, возвращающий значение типа `datetime` с помощью функции `cast()` (привести):

```
mysql> SELECT CAST('2005-03-27 15:30:00' AS DATETIME);
+----------------------------------------+
| CAST('2005-03-27 15:30:00' AS DATETIME) |
+----------------------------------------+
| 2005-03-27 15:30:00                    |
+----------------------------------------+
1 row in set (0.00 sec)
```

Функция cast() будет рассмотрена в конце данной главы. Хотя этот пример демонстрирует построение значений типа datetime, аналогичная логика применятся и к типам date и time. Следующий запрос использует функцию cast() для формирования значения типа date и значения типа time:

```
mysql> SELECT CAST('2005-03-27' AS DATE) date_field,
    ->   CAST('108:17:57' AS TIME) time_field;
+------------+------------+
| date_field | time_field |
+------------+------------+
| 2005-03-27 | 108:17:57  |
+------------+------------+
1 row in set (0.00 sec)
```

Конечно, можно явно преобразовывать строки, даже когда сервер ожидает значение date, datetime или time, а не полагаться на неявное преобразование, выполняемое сервером.

При явном или неявном преобразовании строк во временные значения все компоненты даты должны быть предоставлены в требуемом порядке. Некоторые серверы очень строги относительно формата даты, но сервер MySQL довольно мягок в отношении разделителя компонентов. Например, MySQL примет все нижеприведенные строки как допустимые представления времени 3:30 дня 27 марта 2005 года:

```
'2005-03-27 15:30:00'
'2005/03/27 15:30:00'
'2005,03,27,15,30,00'
'20050327153000'
```

Хотя это и обеспечивает немногим большую гибкость для вас, возможна ситуация, в которой требуется сформировать временное значение *без* стандартных компонентов даты. В следующем разделе будут представлены встроенные функции, гораздо более гибкие, чем функция cast().

Функции для создания дат

Если требуется сгенерировать временные данные из строки, и форма строки не позволяет использовать функцию cast(), можно обратиться к встроенной функции, позволяющей предоставить вместе со строкой даты строку форматирования. MySQL включает для этой цели функцию str_to_date(). Например, для обновления столбца date из файла извлекается строка 'March 27, 2005'. Строка не соответствует требуемому формату YYYY-MM-DD, но вместо того чтобы переформатировать ее, делая пригодной для применения функции cast(), можно воспользоваться функцией str_to_date():

```
UPDATE individual
SET birth_date = STR_TO_DATE('March 27, 2005', '%M %d, %Y')
WHERE cust_id = 9999;
```

Второй аргумент в вызове str_to_date() определяет формат строки даты. В данном случае это название месяца (%M), число (%d) и четырехзначное число, обозначающее год (%Y). Есть более 30 общепринятых компонентов форматирования. В табл. 7.4 приведено около десятка наиболее широко используемых компонентов.

Таблица 7.4. Компоненты форматирования даты

Компонент форматирования	Описание
%M	Название месяца (от January до December)
%m	Номер месяца (от 01 до 12)
%d	Число (от 01 до 31)
%j	День года (от 001 до 366)
%W	Дни недели (от Sunday до Saturday)
%Y	Год, четырехзначное число
%y	Год, двузначное число
%H	Час (от 00 до 23)
%h	Час (от 01 до 12)
%i	Минуты (от 00 до 59)
%s	Секунды (от 00 до 59)
%f	Микросекунды (от 000000 до 999999)
%p	A.M. или P.M.

Функция str_to_date() возвращает значение типа datetime, date или time в зависимости от содержимого форматирующей строки. Например, если форматирующая строка включает только %H, %i и %s, будет возвращено значение типа time.

В распоряжении пользователей Oracle Database имеется функция to_date(), с которой можно работать так же, как с функцией MySQL str_to_date().

При формировании *текущей* даты/времени создавать строку не требуется – следующие встроенные функции организуют доступ к системным часам и возвратят текущую дату и/или время в виде строки:

```
mysql> SELECT CURRENT_DATE( ), CURRENT_TIME( ), CURRENT_TIMESTAMP( );
+---------------+---------------+---------------------+
| CURRENT_DATE( ) | CURRENT_TIME( ) | CURRENT_TIMESTAMP( ) |
+---------------+---------------+---------------------+
| 2005-03-20    | 22:15:56      | 2005-03-20 22:15:56 |
+---------------+---------------+---------------------+
1 row in set (0.00 sec)
```

Возвращаемые этими функциями значения имеют формат по умолчанию для возвращаемого временного типа. В Oracle Database есть функ-

ции current_date() и current_timestamp(), но нет функции current_time(). SQL Server включает только функцию current_timestamp().

Работа с временными данными

В данном разделе рассматриваются встроенные функции, принимающие аргументы даты и возвращающие даты, строки или числа.

Временные функции, возвращающие даты

Многие встроенные временные функции принимают в качестве аргумента одну дату и возвращают другую. Например, функция MySQL date_add() позволяет добавить любой интервал (т. е. дни, месяцы, года) к заданной дате, чтобы получить другую дату. Вот пример, демонстрирующий, как добавить к текущей дате пять дней:

```
mysql> SELECT DATE_ADD(CURRENT_DATE(  ), INTERVAL 5 DAY);
+-----------------------------------------+
| DATE_ADD(CURRENT_DATE(  ), INTERVAL 5 DAY) |
+-----------------------------------------+
| 2005-03-26                              |
+-----------------------------------------+
1 row in set (0.00 sec)
```

Второй аргумент заключает в себе три элемента: ключевое слово interval (интервал), требуемое количество и тип интервала. В табл. 7.5 приведены некоторые широко используемые типы интервалов.

Таблица 7.5. Общепринятые типы интервалов

Интервал	Описание
Second	Количество секунд
Minute	Количество минут
Hour	Количество часов
Day	Количество дней
Month	Количество месяцев
Year	Количество лет
Minute_second	Количества минут и секунд, разделенные двоеточием
Hour_second	Количества часов, минут и секунд, разделенные двоеточием
Year_month	Количества лет и месяцев, разделенные дефмсом

Первые шесть типов, перечисленные в табл. 7.5, довольно просты, а последние три требуют немного более подробного объяснения, поскольку содержат по несколько элементов. Например, если оказалось, что операция с ID 9999 на самом деле имела место на 3 часа 27 минут и 11 секунд позже того значения, которое было отправлено в таблицу Transaction, исправить это можно следующим образом:

```
UPDATE transaction
SET txn_date = DATE_ADD(txn_date, INTERVAL '3:27:11' HOUR_SECOND)
WHERE txn_id = 9999;
```

В этом примере функция берет значение столбца txn_date, добавляет к нему 3 часа 27 минут и 11 секунд и изменяет столбец txn_date, вставляя в него результирующее значение.

Или если в отделе кадров обнаруживают, что сотрудник с ID 4789 по записанным данным моложе, чем на самом деле, можно добавить к дате его рождения, скажем, 9 лет и 11 месяцев:

```
UPDATE employee
SET birth_date = DATE_ADD(birth_date, INTERVAL '9-11' YEAR_MONTH)
WHERE emp_id = 4789;
```

Для пользователей SQL Server предыдущий пример можно было бы реализовать с помощью функции dateadd():

```
UPDATE employee
SET birth_date =
  DATEADD(MONTH, 119, birth_date)
WHERE emp_id = 4789
```

В SQL Server нет комбинированных интервалов (т. е. year_month), поэтому 9 лет 11 месяцев были преобразованы в 119 месяцев.

Пользователи Oracle Database могут для данного примера применить функцию add_months():

```
UPDATE employee
SET birth_date = ADD_MONTHS(birth_date, 119)
WHERE emp_id = 4789;
```

Иногда требуется добавить интервал времени к определенной дате, при этом известна конечная дата выполнения, но неизвестно, сколько дней осталось до этой даты. Например, клиент банка регистрируется в сетевой банковской системе и планирует перевод на конец месяца. Вместо того чтобы писать какой-то код, определяющий текущий месяц и вычисляющий количество дней в этом месяце, можно вызвать функцию last_day() (последний день), которая сделает всю работу (и MySQL, и Oracle Database включают функцию last_day(); в SQL Server сопоставимой функции нет). Если клиент запрашивает перевод 25 марта 2005 года, последний день марта можно найти следующим образом:

```
mysql> SELECT LAST_DAY('2005-03-25');
+------------------------+
| LAST_DAY('2005-03-25') |
+------------------------+
| 2005-03-31             |
+------------------------+
1 row in set (0.04 sec)
```

Независимо от того, предоставляется ли значение типа `date` или `datetime`, функция `last_day()` всегда возвращает значение типа `date`. Хотя, может быть, и не заметно, что эта функция существенно экономит время, но если требуется найти последний день февраля и для этого выяснить, високосный этот год или нет, логика вычисления может быть довольно сложной.

Еще одна временная функция, возвращающая дату, преобразует значение типа `datetime` из одного временного пояса в другой. Для этого MySQL включает функцию `convert_tz()`, а Oracle Database – функцию `new_time()` (новое время). Например, если требуется преобразовать текущее местное время в UTC, можно сделать следующее:

```
mysql> SELECT CURRENT_TIMESTAMP(  ) current_est,
    ->   CONVERT_TZ(CURRENT_TIMESTAMP(  ), 'US/Eastern', 'UTC') current_utc;
+---------------------+---------------------+
| current_est         | current_utc         |
+---------------------+---------------------+
| 2005-04-18 21:23:25 | 2005-04-19 01:23:25 |
+---------------------+---------------------+
1 row in set (0.50 sec)
```

Эта функция очень полезна при получении дат из других часовых поясов, отличных от того, в котором хранится база данных.

Временные функции, возвращающие строки

Большинство временных функций, возвращающих строковые значения, используются для получения отдельной части даты или времени. Например, MySQL включает функцию `dayname()` (название дня), определяющую, на какой день недели приходится определенная дата:

```
mysql> SELECT DAYNAME('2005-03-22');
+-----------------------+
| DAYNAME('2005-03-22') |
+-----------------------+
| Tuesday               |
+-----------------------+
1 row in set (0.12 sec)
```

В MySQL много таких функций, предназначенных для извлечения информации из значений дат, но я рекомендую пользоваться функцией `extract()` (извлечь), поскольку проще запомнить несколько разновидностей одной функции, чем десяток разных функций. Кроме того, `extract()` является частью стандарта SQL:2003 и была реализована не только в MySQL, но и в Oracle Database.

Для определения интересующего элемента даты функция `extract()` использует те же типы интервалов, что и функция `date_add()` (см. табл. 7.5). Например, если из значения типа `datetime` требуется извлечь только год, можно поступить так:

```
mysql> SELECT EXTRACT(YEAR FROM '2005-03-22 22:19:05');
+-----------------------------------------+
| EXTRACT(YEAR FROM '2005-03-22 22:19:05') |
+-----------------------------------------+
|                                    2005 |
+-----------------------------------------+
1 row in set (0.02 sec)
```

В SQL Server нет реализации `extract()`, но есть функция `datepart()` (часть даты). Вот как можно извлечь год из значения `datetime` с помощью `datepart()`:

```
SELECT DATEPART(YEAR, GETDATE(  ))
```

Временные функции, возвращающие числа

Ранее в этой главе была представлена функция, используемая для добавления заданного интервала к значению даты и формирующая, таким образом, другую дату. Другая распространенная операция при работе с датами – определение количества интервалов (дней, недель, лет) *между* двумя датами. MySQL включает предназначенную для этого функцию `datediff()`, которая возвращает количество полных дней между двумя датами. Например, чтобы узнать, сколько дней будут продолжаться школьные каникулы этим летом, можно сделать так:

```
mysql> SELECT DATEDIFF('2005-09-05', '2005-06-22');
+--------------------------------------+
| DATEDIFF('2005-09-05', '2005-06-22') |
+--------------------------------------+
|                                   75 |
+--------------------------------------+
1 row in set (0.00 sec)
```

Итак, до благополучного возвращения детей в школу мне предстоит 75-дневная пытка ядовитым плющом, комариными укусами и разбитыми коленками. Функция `datediff()` в своих аргументах не учитывает время дня. Даже если включить время, задавая для первой даты одну секунду до полуночи и для второй даты одну секунду после полуночи, эти данные никак не отразятся на вычислениях:

```
mysql> SELECT DATEDIFF('2005-09-05 23:59:59', '2005-06-22 00:00:01');
+-------------------------------------------------------+
| DATEDIFF('2005-09-05 23:59:59', '2005-06-22 00:00:01') |
+-------------------------------------------------------+
|                                                    75 |
+-------------------------------------------------------+
1 row in set (0.00 sec)
```

Если переставить аргументы, поместив первой более раннюю дату, `datediff()` вернет отрицательное число:

```
mysql> SELECT DATEDIFF('2005-06-22', '2005-09-05');
```

```
+-------------------------------------+
| DATEDIFF('2005-06-22', '2005-09-05') |
+-------------------------------------+
|                                 -75 |
+-------------------------------------+
```

В SQL Server тоже есть функция `datediff()`, но более гибкая, чем в MySQL. В ней можно задавать тип интервала (т. е. год, месяц, день, час), а не только вычислять количество дней между двумя датами. Вот как был бы выполнен предыдущий пример для SQL Server:

```
SELECT DATEDIFF(DAY, '2005-06-22',
  '2005-09-05')
```

Oracle Database позволяет определять число дней между двумя датами простым вычитанием одной даты из другой.

Функции преобразования

Ранее в этой главе было показано, как использовать функцию `cast()` для преобразования строки в значение типа `datetime`. Хотя у всех серверов БД есть собственные функции преобразования данных из одного типа в другой, я рекомендую использовать функцию `cast()`, включенную в стандарт SQL:2003 и реализованную в MySQL, Oracle Database и Microsoft SQL Server.

Применяя функцию `cast()`, необходимо задать значение или выражение, ключевое слово `as` и тип, в который должно быть преобразовано значение. Вот пример преобразования строки в целое:

```
mysql> SELECT CAST('1456328' AS SIGNED INTEGER);
+-----------------------------------+
| CAST('1456328' AS SIGNED INTEGER) |
+-----------------------------------+
|                           1456328 |
+-----------------------------------+
1 row in set (0.01 sec)
```

При преобразовании строки в число функция `cast()` попытается преобразовать всю строку слева направо. Если в строке встретиться любой нечисловой символ, преобразование будет остановлено без формирования ошибки. Рассмотрим следующий пример:

```
mysql> SELECT CAST('999ABC111' AS UNSIGNED INTEGER);
+---------------------------------------+
| CAST('999ABC111' AS UNSIGNED INTEGER) |
+---------------------------------------+
|                                   999 |
+---------------------------------------+
1 row in set (0.00 sec)
```

В данном случае преобразуются первые три цифры в строке, все остальные символы строки отбрасываются. В результате получаем значение `999`.

Для преобразования строки в значение типа `date`, `time` или `datetime` понадобится придерживаться форматов по умолчанию для каждого типа, поскольку передать строку формата в функцию `cast()` невозможно. Если формат строки даты не соответствует применяемому по умолчанию (т. е. YYYY-MM-DD HH:MI:SS для типов `datetime`), придется прибегнуть к другой функции, например к функции MySQL `str_to_date()`, описанной ранее в этой главе.

Упражнения

Эти упражнения позволяют проверить понимание читателем некоторых встроенных функций, упомянутых в данной главе. Ответы приведены в приложении C.

7.1

Напишите запрос, возвращающий с 17-го по 25-й символы строки `'Please find the substring in this string'` (Пожалуйста, найдите подстроку в этой строке).

7.2

Напишите запрос, возвращающий абсолютную величину и знак (`-1`, `0` или `1`) числа -25,768 23. Также возвратите число, округленное до сотых.

7.3

Напишите запрос, возвращающий только значение месяца текущей даты.

8

Группировка и агрегаты

Обычно данные хранятся с самым низким уровнем детализации, который может понадобиться какому-либо пользователю базы данных. Если Чаку (Chuck) для бухгалтерского учета требуется посмотреть операции одного клиента, в БД должна быть таблица, хранящая операции отдельного клиента. Однако это не означает, что все пользователи должны работать с данными в том виде, в каком они хранятся в БД. Основное внимание уделено группировке и агрегированию данных, которые обеспечивают пользователям возможность работать с данными на более высоком уровне детализации, чем тот, с которым они хранятся в БД.

Принципы группировки

Иногда необходимо выявить в данных некоторые тенденции, что потребует от сервера некоторой подготовки данных, прежде чем можно будет получить искомые результаты. Например, вы отвечаете за операции в банке и хотели бы выяснить, сколько счетов открывает каждый операционист банка. Можно было бы создать простой запрос для просмотра необработанных данных:

```
mysql> SELECT open_emp_id
    -> FROM account;
+-------------+
| open_emp_id |
+-------------+
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
```

```
|          10 |
|          10 |
|          10 |
|          10 |
|          10 |
|          10 |
|          10 |
|          13 |
|          13 |
|          13 |
|          16 |
|          16 |
|          16 |
|          16 |
|          16 |
|          16 |
+-------------+
24 rows in set (0.01 sec)
```

В таблице `account` всего 24 строки, поэтому относительно просто увидеть, что счета открывались четырьмя сотрудниками и что сотрудник с ID 16 открыл шесть счетов. Но для банка с десятками сотрудников и тысячами открываемых счетов этот подход оказался бы очень утомительным и подверженным ошибкам.

Вместо этого можно с помощью блока `group by` (группировать по) попросить сервер БД сгруппировать данные. Вот тот же запрос, но с применением блока `group by` для группировки данных о счетах по ID сотрудника:

```
mysql> SELECT open_emp_id
    -> FROM account
    -> GROUP BY open_emp_id;
+-------------+
| open_emp_id |
+-------------+
|           1 |
|          10 |
|          13 |
|          16 |
+-------------+
4 rows in set (0.00 sec)
```

Результирующий набор содержит по одной строке для каждого отдельного значения столбца `open_emp_id`, что в результате дает четыре, а не 24 строки. Этот результирующий набор получился меньшим, потому что каждый из четырех сотрудников открыл больше одного счета. Чтобы увидеть, сколько счетов открыл каждый сотрудник, в блоке `select` можно подсчитать количество строк в каждой группе с помощью *агрегатной функции* (*aggregate function*):

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
```

```
    -> GROUP BY open_emp_id;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|           1 |        8 |
|          10 |        7 |
|          13 |        3 |
|          16 |        6 |
+-------------+----------+
4 rows in set (0.00 sec)
```

Агрегатная функция `count()` подсчитывает количество строк в каждой группе, а звездочка предписывает серверу сосчитать все строки в группе. Сочетание блока `group by` и функции обобщения `count()` позволяет формировать именно те данные, которые требуются для ответа на прикладной вопрос, без необходимости просматривать необработанные данные.

При группировке может понадобиться отфильтровать из результирующего набора ненужные данные, опираясь на информацию групп данных, а не необработанных данных. Блок `group by` выполняется *после* вычисления блока `where`, поэтому условия фильтрации нельзя добавлять в блок `where`. Вот, например, попытка отфильтровать всех сотрудников, открывших меньше пяти счетов:

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> WHERE COUNT(*) > 4
    -> GROUP BY open_emp_id, product_cd;
ERROR 1111 (HY000): Invalid use of group function
```

Агрегатную функцию `count(*)` нельзя использовать в блоке `where`, потому что на момент вычисления блока `where` группы еще не сформированы. Вместо этого можно поместить условия фильтрации группы в блок `having`. Вот пример того же запроса с блоком `having`:

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id
    -> HAVING COUNT(*) > 4;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|           1 |        8 |
|          10 |        7 |
|          16 |        6 |
+-------------+----------+
3 rows in set (0.00 sec)
```

Группы, содержащие меньше пяти элементов, были отфильтрованы с помощью блока `having`, и теперь результирующий набор включает только сотрудников, открывших пять или более счетов.

Агрегатные функции

Агрегатные функции осуществляют определенную операцию над всеми строками группы. Хотя у всех серверов БД есть собственные наборы специализированных агрегатных функций, большинством из них реализованы следующие общие агрегатные функции:

`Max()`

Возвращает максимальное значение из набора.

`Min()`

Возвращает минимальное значение из набора.

`Avg()`

Возвращает среднее значение набора.

`Sum()`

Возвращает сумму значений из набора.

`Count()`

Возвращает количество значений в наборе.

Вот запрос, использующий все обычные агрегатные функции для анализа доступных остатков (available balance) всех текущих счетов:

```
mysql> SELECT MAX(avail_balance) max_balance,
    ->   MIN(avail_balance) min_balance,
    ->   AVG(avail_balance) avg_balance,
    ->   SUM(avail_balance) tot_balance,
    ->   COUNT(*) num_accounts
    -> FROM account
    -> WHERE product_cd = 'CHK';
+-------------+-------------+-------------+-------------+--------------+
| max_balance | min_balance | avg_balance | tot_balance | num_accounts |
+-------------+-------------+-------------+-------------+--------------+
|   385527.05 |      122.37 | 7300.800985 |    73008.01 |           10 |
+-------------+-------------+-------------+-------------+--------------+
1 row in set (0.09 sec)
```

Результаты этого запроса сообщают о том, что из десяти текущих счетов таблицы `account` максимальный остаток составляет 38 552,05 долларов, минимальный остаток – 122,37 долларов, средний остаток – 7 300,80 долларов, а общий остаток (баланс) по всем десяти счетам – 73 008,01 долларов. Надеюсь, теперь роль данных агрегатных функций вам ясна; возможности применения этих функций подробно рассмотрены в следующих разделах.

Сравнение неявных и явных групп

В предыдущем примере все значения, возвращаемые по запросу, формируются агрегатной функцией, а сами агрегатные функции применяются к группе строк, определенной условием фильтрации `product_cd =`

'CHK'. Поскольку блок group by отсутствует, имеется единственная *неявная* группа (все возвращенные запросом строки).

Однако в большинстве случаев потребуется извлекать и другие столбцы, а не только сформированные агрегатными функциями. Что если, к примеру, заставить предыдущий запрос выполнить эти же пять агрегатных функций для каждого типа счетов, а не только для текущих счетов? Для такого запроса пришлось бы извлекать столбец product_cd в дополнение к столбцам, сформированным пятью агрегатными функциями:

```
SELECT product_cd,
  MAX(avail_balance) max_balance,
  MIN(avail_balance) min_balance,
  AVG(avail_balance) avg_balance,
  SUM(avail_balance) tot_balance,
  COUNT(*) num_accounts
FROM account;
```

Однако если попытаться выполнить этот запрос, будет получена следующая ошибка:

```
ERROR 1140 (42000): Mixing of GROUP columns (MIN(),MAX(),COUNT(  ),...)
with no GROUP columns is illegal if there is no GROUP BY clause
```

Хотя для разработчика очевидно, что он хочет применить агрегатные функции к множеству счетов каждого типа, выявленного в таблице account, этот запрос дает сбой, потому что не был *явно* задан способ группировки данных. Следовательно, необходимо добавить блок group by и определить в нем группу строк, к которой следует применять агрегатные функции:

```
mysql> SELECT product_cd,
    ->   MAX(avail_balance) max_balance,
    ->   MIN(avail_balance) min_balance,
    ->   AVG(avail_balance) avg_balance,
    ->   SUM(avail_balance) tot_balance,
    ->   COUNT(*) num_accts
    -> FROM account
    -> GROUP BY product_cd;
+------------+-------------+-------------+-------------+-------------+-----------+
| product_cd | max_balance | min_balance | avg_balance | tot_balance | num_accts |
+------------+-------------+-------------+-------------+-------------+-----------+
| BUS        |     9345.55 |        0.00 | 4672.774902 |     9345.55 |         2 |
| CD         |    10000.00 |     1500.00 | 4875.000000 |    19500.00 |         4 |
| CHK        |    38552.05 |      122.37 | 7300.800985 |    73008.01 |        10 |
| MM         |     9345.55 |     2212.50 | 5681.713216 |    17045.14 |         3 |
| SAV        |      767.77 |      200.00 |  463.940002 |     1855.76 |         4 |
| SBL        |    50000.00 |    50000.00 | 50000.000000 |    50000.00 |         1 |
+------------+-------------+-------------+-------------+-------------+-----------+
6 rows in set (0.00 sec)
```

Если есть блок `group by`, сервер знает, что сначала надо сгруппировать строки с одинаковым значением в столбце `product_cd`, а затем применить пять агрегатных функций к каждой из шести групп.

Подсчет уникальных значений

При использовании функции `count()` для определения числа членов в каждой группе существует выбор: или пересчитать *все* члены группы, или посчитать только *уникальные* (*distinct*) значения столбца из всех членов группы. Рассмотрим, например, следующие данные, которыми представлены сотрудники, ответственные за открытие каждого счета:

```
mysql> SELECT account_id, open_emp_id
    -> FROM account
    -> ORDER BY open_emp_id;
+------------+-------------+
| account_id | open_emp_id |
+------------+-------------+
|          8 |           1 |
|          9 |           1 |
|         10 |           1 |
|         12 |           1 |
|         13 |           1 |
|         17 |           1 |
|         18 |           1 |
|         19 |           1 |
|          1 |          10 |
|          2 |          10 |
|          3 |          10 |
|          4 |          10 |
|          5 |          10 |
|         14 |          10 |
|         22 |          10 |
|          6 |          13 |
|          7 |          13 |
|         24 |          13 |
|         11 |          16 |
|         15 |          16 |
|         16 |          16 |
|         20 |          16 |
|         21 |          16 |
|         23 |          16 |
+------------+-------------+
24 rows in set (0.00 sec)
```

Как видите, все множество счетов было открыто четырьмя разными сотрудниками (с ID = 1, 10, 13 и 16). Допустим, хочется подсчитать число открывших счета сотрудников – не вручную, а с помощью запроса. Если к столбцу `open_emp_id` применить функцию `count()`, увидим следующие результаты:

```
mysql> SELECT COUNT(open_emp_id)
    -> FROM account;
+--------------------+
| COUNT(open_emp_id) |
+--------------------+
|                 24 |
+--------------------+
1 row in set (0.00 sec)
```

В этом случае столбец open_emp_id задан как столбец, который должен быть пересчитан. При этом полученный результат ничем не отличается от результата выполнения функции count(*). Если требуется подсчитать *уникальные* значения в группе, а не просто пересчитать число строк в ней, нужно указать ключевое слово distinct:

```
mysql> SELECT COUNT(DISTINCT open_emp_id)
    -> FROM account;
+-----------------------------+
| COUNT(DISTINCT open_emp_id) |
+-----------------------------+
|                           4 |
+-----------------------------+
1 row in set (0.00 sec)
```

Следовательно, если задано ключевое слово distinct, функция count() проверит значение столбца для каждого члена группы, а не просто подсчитает число значений в ней.

Использование выражений

В качестве аргументов агрегатных функций вы можете использовать не только столбцы, но и созданные вами выражения. Например, требуется найти максимальное значение отложенных вкладов по всем счетам, которое вычисляется путем вычитания доступного остатка из отложенного остатка. Сделать это можно посредством следующего запроса:

```
mysql> SELECT MAX(pending_balance - avail_balance) max_uncleared
    -> FROM account;
+---------------+
| max_uncleared |
+---------------+
|        660.00 |
+---------------+
1 row in set (0.00 sec)
```

В данном примере используется довольно простое выражение, но применяемые в качестве аргументов агрегатных функций выражения могут быть настолько сложными, насколько это нужно, и возвращать число, строку или дату. В главе 11 будет показано, как с помощью выражения case и агрегатных функций можно управлять попаданием или непопаданием конкретной строки под действие агрегатной функции.

Обработка значений Null

При агрегировании, да и при вычислении любого численного выражения, всегда следует учитывать влияние значения `null` на результат вычисления. Для иллюстрации создадим простую таблицу для хранения числовых данных и заполним ее набором {1, 3, 5}:

```
mysql> CREATE TABLE number_tbl
    ->  (val SMALLINT);
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO number_tbl VALUES (1);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO number_tbl VALUES (3);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO number_tbl VALUES (5);
Query OK, 1 row affected (0.00 sec)
```

Рассмотрим следующий запрос, применяющий пять агрегатных функций к этому набору чисел:

```
mysql> SELECT COUNT(*) num_rows,
    ->   COUNT(val) num_vals,
    ->   SUM(val) total,
    ->   MAX(val) max_val,
    ->   AVG(val) avg_val
    -> FROM number_tbl;
+----------+----------+-------+---------+---------+
| num_rows | num_vals | total | max_val | avg_val |
+----------+----------+-------+---------+---------+
|        3 |        3 |     9 |       5 |       3 |
+----------+----------+-------+---------+---------+
1 row in set (0.00 sec)
```

Как и следовало ожидать, результаты таковы: и `count(*)`, и `count(val)` возвращают значение `3`, `sum(val)` — значение `9`, `max(val)` — `5`, а `avg(val)` — `3`. Теперь добавим в таблицу `number_tbl` значение `null` и выполним запрос еще раз:

```
mysql> INSERT INTO number_tbl VALUES (NULL);
Query OK, 1 row affected (0.01 sec)

mysql> SELECT COUNT(*) num_rows,
    ->   COUNT(val) num_vals,
    ->   SUM(val) total,
    ->   MAX(val) max_val,
    ->   AVG(val) avg_val
    -> FROM number_tbl;
+----------+----------+-------+---------+---------+
| num_rows | num_vals | total | max_val | avg_val |
+----------+----------+-------+---------+---------+
|        4 |        3 |     9 |       5 |       3 |
```

```
+----------+----------+-------+---------+---------+
1 row in set (0.00 sec)
```

Даже при добавлении в таблицу значения null функции sum(), max() и avg() возвращают те же значения; это означает, что они игнорируют все встречающиеся значения null. Функция count(*) теперь возвращает значение 4, что является правильным, поскольку в таблице number_tbl четыре строки, тогда как функция count(val) по-прежнему возвращает значение 3. Разница в том, что функция count(*) считает строки и поэтому не подвержена влиянию значений null, содержащихся в строке. А вот функция count(val) считает *значения* в столбце val, пропуская все встречающиеся значения null.

Формирование групп

Мало кого заинтересуют необработанные данные; тем, кто занимается анализом, потребуются обработанные данные, приведенные к виду, наиболее соответствующему их нуждам. Среди обычных примеров манипуляций с данными можно назвать:

- Формирование общих показателей для географического региона, например общий объем продаж по Европе.
- Выявление экстремумов, например лучший продавец 2005 года.
- Определение повторяемости, например число новых счетов, открытых в каждом отделении.

Чтобы ответить на запросы подобных типов, вам потребуется попросить сервер БД сгруппировать строки по одному или более столбцам или выражениям. Как уже было показано в нескольких примерах, механизмом группировки данных в рамках запроса является блок group by. В этом разделе рассматриваются группировка данных по одному или более столбцам, группировка данных с помощью выражений и формирование обобщений в рамках группы.

Группировка по одному столбцу

Формирование группы по одному столбцу – самый простой и наиболее распространенный тип группировки. Например, если требуется найти общие остатки (total balance) для всех типов счетов, нужно всего лишь провести группировку по столбцу account.product_cd:

```
mysql> SELECT product_cd, SUM(avail_balance) prod_balance
    -> FROM account
    -> GROUP BY product_cd;
+------------+--------------+
| product_cd | prod_balance |
+------------+--------------+
| BUS        |      9345.55 |
| CD         |     19500.00 |
```

```
| CHK          |     73008.01 |
| MM           |     17045.14 |
| SAV          |      1855.76 |
| SBL          |     50000.00 |
+--------------+--------------+
6 rows in set (0.00 sec)
```

Этот запрос формирует шесть групп, по одной для каждого типа счетов, и затем суммирует доступные остатки по всем строкам в каждой группе.

Группировка по нескольким столбцам

В некоторых случаях может понадобиться сформировать группы, охватывающие *более* одного столбца. Развивая предыдущий пример, представим, что требуется найти общие остатки не только по каждому типу счетов, но и по отделениям (например: каков общий остаток для всех текущих счетов, открытых в отделении Woburn?). Следующий пример демонстрирует, как это может быть реализовано:

```
mysql> SELECT product_cd, open_branch_id,
    ->   SUM(avail_balance) tot_balance
    -> FROM account
    -> GROUP BY product_cd, open_branch_id;
+--------------+----------------+-------------+
| product_cd | open_branch_id | tot_balance |
+--------------+----------------+-------------+
| BUS          |              2 |     9345.55 |
| BUS          |              4 |        0.00 |
| CD           |              1 |    11500.00 |
| CD           |              2 |     8000.00 |
| CHK          |              1 |      782.16 |
| CHK          |              2 |     3315.77 |
| CHK          |              3 |     1057.75 |
| CHK          |              4 |    67852.33 |
| MM           |              1 |    14832.64 |
| MM           |              3 |     2212.50 |
| SAV          |              1 |      767.77 |
| SAV          |              2 |      700.00 |
| SAV          |              4 |      387.99 |
| SBL          |              3 |    50000.00 |
+--------------+----------------+-------------+
14 rows in set (0.00 sec)
```

Этот вариант запроса формирует 14 групп, по одной для каждого обнаруженного в таблице `account` сочетания типа счетов и отделения. Столбец `open_branch_id` добавлен в блок `select`, а также введен в блок `group by`, поскольку он извлекается из таблицы, а не формируется агрегатной функцией.

Группировка посредством выражений

Кроме столбцов группировку данных можно выполнить на основании значений, сгенерированных выражениями. Рассмотрим запрос, который группирует сотрудников по году начала их работы в банке:

```
mysql> SELECT EXTRACT(YEAR FROM start_date) year,
    ->   COUNT(*) how_many
    -> FROM employee
    -> GROUP BY EXTRACT(YEAR FROM start_date);
+------+----------+
| year | how_many |
+------+----------+
| 2000 |        3 |
| 2001 |        2 |
| 2002 |        8 |
| 2003 |        3 |
| 2004 |        2 |
+------+----------+
5 rows in set (0.00 sec)
```

Этот запрос для группировки строк таблицы `employee` использует довольно простое выражение, которое с помощью функции `extract()` из всей даты извлекает только значение года.

Формирование обобщений

Ранее в этой главе в разделе «Группировка по нескольким столбцам» был показан пример формирования общих остатков счетов по каждому типу счетов и отделению. Однако допустим, что кроме общих остатков для каждого сочетания тип счетов/отделение требуется получить и общие остатки по каждому отдельному типу счетов. Можно было бы выполнить дополнительный запрос и объединить результаты, или загрузить результаты запроса в электронную таблицу, или создать сценарий на Perl или Java-программу, или применить какой-либо другой механизм для получения данных и проведения дополнительных вычислений. Но все-таки лучше всего использовать вариант `with rollup` (с обобщением), заставив выполнить всю эту работу сервер БД. Вот измененный запрос, использующий `with rollup` в блоке `group by`:

```
mysql> SELECT product_cd, open_branch_id,
    ->   SUM(avail_balance) tot_balance
    -> FROM account
    -> GROUP BY product_cd, open_branch_id WITH ROLLUP;
+------------+----------------+-------------+
| product_cd | open_branch_id | tot_balance |
+------------+----------------+-------------+
| BUS        |              2 |     9345.55 |
| BUS        |              4 |        0.00 |
| BUS        |           NULL |     9345.55 |
| CD         |              1 |    11500.00 |
```

```
| CD          |              2 |     8000.00 |
| CD          |           NULL |    19500.00 |
| CHK         |              1 |      782.16 |
| CHK         |              2 |     3315.77 |
| CHK         |              3 |     1057.75 |
| CHK         |              4 |    67852.33 |
| CHK         |           NULL |    73008.01 |
| MM          |              1 |    14832.64 |
| MM          |              3 |     2212.50 |
| MM          |           NULL |    17045.14 |
| SAV         |              1 |      767.77 |
| SAV         |              2 |      700.00 |
| SAV         |              4 |      387.99 |
| SAV         |           NULL |     1855.76 |
| SBL         |              3 |    50000.00 |
| SBL         |           NULL |    50000.00 |
| NULL        |           NULL |   170754.46 |
+-------------+----------------+-------------+
21 rows in set (0.02 sec)
```

Теперь имеется семь дополнительных результатов, по одному для каждого из шести разных типов счетов, и один – общая сумма (для всех типов счетов). Для шести обобщений по типам счетов столбец `open_branch_id` содержит значение `null`, поскольку обобщение осуществляется по всем отделениям. Например, взглянув на строку #3 результата, можно заметить, что всего по счетам BUS во всех отделениях внесено 9 345,55 долларов. Строка итоговой суммы в обоих столбцах, `product_cd` и `open_branch_id`, содержит значение `null`. Последняя строка выходных данных показывает общую сумму 170 754,46 долларов для всех типов счетов и всех отделений.

При работе с Oracle Database для выполнения обобщения применяется немного отличающийся синтаксис. Блок `group by` из предыдущего запроса при использовании в Oracle выглядел бы так:

```
GROUP BY ROLLUP(product_cd, open_branch_id)
```

Преимущество этого синтаксиса в том, что он позволяет выполнять обобщения для подмножества столбцов в блоке `group by`. Например, если группировка осуществляется по столбцам `a`, `b` и `c`, можно было бы указать, что сервер должен проводить обобщения только для `b` и `c`:

```
GROUP BY a, ROLLUP(b, c)
```

Если кроме суммы по типам счетов требуется подсчитать сумму по каждому отделению, можно использовать вариант `with cube`, который формирует строки суммы для *всех* возможных сочетаний группирующих столбцов. К сожалению, `with cube` недоступен в MySQL версии 4.1, но есть в SQL Server и Oracle Database. Вот пример использования `with cube` (я убрал приглашение `mysql>`, чтобы показать, что этот запрос пока нельзя осуществить в MySQL):

```
SELECT product_cd, open_branch_id,
  SUM(avail_balance) tot_balance
FROM account
GROUP BY product_cd, open_branch_id WITH CUBE;
+------------+----------------+-------------+
| product_cd | open_branch_id | tot_balance |
+------------+----------------+-------------+
| NULL       |           NULL |   170754.46 |
| NULL       |              1 |    27882.57 |
| NULL       |              2 |    21361.32 |
| NULL       |              3 |    53270.25 |
| NULL       |              4 |    68240.32 |
| BUS        |              2 |     9345.55 |
| BUS        |              4 |        0.00 |
| BUS        |           NULL |     9345.55 |
| CD         |              1 |    11500.00 |
| CD         |              2 |     8000.00 |
| CD         |           NULL |    19500.00 |
| CHK        |              1 |      782.16 |
| CHK        |              2 |     3315.77 |
| CHK        |              3 |     1057.75 |
| CHK        |              4 |    67852.33 |
| CHK        |           NULL |    73008.01 |
| MM         |              1 |    14832.64 |
| MM         |              3 |     2212.50 |
| MM         |           NULL |    17045.14 |
| SAV        |              1 |      767.77 |
| SAV        |              2 |      700.00 |
| SAV        |              4 |      387.99 |
| SAV        |           NULL |     1855.76 |
| SBL        |              3 |    50000.00 |
| SBL        |           NULL |    50000.00 |
+------------+----------------+-------------+
25 rows in set (0.02 sec)
```

Применение `with cube` дает на четыре строки больше, чем версия запроса с `with rollup`, по одной для каждого из четырех ID отделений. Как и в случае с `with rollup`, значения `null` в столбце `product_cd` обозначают то, что производится суммирование по отделениям.

При работе с Oracle Database для указания на операцию `cube` также применяется немного отличающийся синтаксис. Блок `group by` из предыдущего запроса для Oracle выглядел бы так:

```
GROUP BY CUBE(product_cd, open_branch_id)
```

Условия групповой фильтрации

В главе 4 были представлены различные типы условий фильтрации и показано, как их можно использовать в блоке `where`. При группировке данных тоже можно применять условия фильтрации к данным *после*

формирования групп. Этот тип условий фильтрации должен располагаться в блоке having. Рассмотрим следующий пример:

```
mysql> SELECT product_cd, SUM(avail_balance) prod_balance
    -> FROM account
    -> WHERE status = 'ACTIVE'
    -> GROUP BY product_cd
    -> HAVING SUM(avail_balance) >= 10000;
+------------+--------------+
| product_cd | prod_balance |
+------------+--------------+
| CD         |     19500.00 |
| CHK        |     73008.01 |
| MM         |     17045.14 |
| SBL        |     50000.00 |
+------------+--------------+
4 rows in set (0.00 sec)
```

В этом запросе два условия фильтрации: одно в блоке where (отсеиваются неактивные счета), а второе в блоке having (отсеиваются счета всех типов с общим доступным остатком меньше 10 000 долларов). Таким образом, один из фильтров воздействует на данные *до* группировки, а другой – *после* создания групп. Если по ошибке оба фильтра помещены в блок where, возникает следующая ошибка:

```
mysql> SELECT product_cd, SUM(avail_balance) prod_balance
    -> FROM account
    -> WHERE status = 'ACTIVE'
    ->   AND SUM(avail_balance) > 10000
    -> GROUP BY product_cd;
ERROR 1111 (HY000): Invalid use of group function
```

Данный запрос дал сбой, потому что агрегатную функцию нельзя включать в блок where. Причина в том, что фильтры блока обрабатываются *до* выполнения группировки, поэтому серверу еще не доступны какие-либо действия над группами.

При введении фильтров в запрос, включающий блок group by, необходимо тщательно продумать, к чему применяется фильтр – к необработанным данным (тогда он относится к блоку where) или к сгруппированным данным (в этом случае он относится к блоку having).

Однако в блок having можно включить агрегатные функции, *не* перечисленные в блоке select, как показано ниже:

```
mysql> SELECT product_cd, SUM(avail_balance) prod_balance
    -> FROM account
    -> WHERE status = 'ACTIVE'
    -> GROUP BY product_cd
    -> HAVING MIN(avail_balance) >= 1000
    ->   AND MAX(avail_balance) <= 10000;
```

```
+------------+--------------+
| product_cd | prod_balance |
+------------+--------------+
| MM         |     17045.14 |
+------------+--------------+
1 row in set (0.01 sec)
```

Этот запрос формирует общие остатки для каждого типа счетов, но условие фильтрации блока `having` исключает все группы, минимальный остаток которых меньше 1000 долларов или максимальный остаток которых больше 10 000 долларов.

Упражнения

Проработайте следующие упражнения, чтобы протестировать понимание группировки и агрегатных функций SQL. Ответы приведены в приложении C.

8.1

Создайте запрос для подсчета числа строк в таблице `account`.

8.2

Измените свой запрос из упражнения 8.1 для подсчета числа счетов, имеющихся у каждого клиента. Для каждого клиента выведите ID клиента и количество счетов.

8.3

Измените запрос из упражнения 8.2 так, чтобы в результирующий набор были включены только клиенты, имеющие не менее двух счетов.

8.4 (дополнительно)

Найдите общий доступный остаток по типу счетов и отделению, где на каждый тип и отделение приходится более одного счета. Результаты должны быть упорядочены по общему остатку (от наибольшего к наименьшему).

9

Подзапросы

Подзапросы – мощный инструмент, который можно использовать во всех четырех SQL-выражениях для работы с данными. В этой главе подробно рассматриваются многие варианты применения подзапроса.

Что такое подзапрос?

Подзапрос (*subquery*) – это запрос, содержащийся в другом SQL-выражении (далее я называю его *содержащим выражением* (*containing statement*)). Подзапрос всегда заключен в круглые скобки и обычно выполняется до содержащего выражения. Как и любой другой запрос, подзапрос возвращает таблицу, которая может состоять из:

- Одной строки с одним столбцом
- Нескольких строк с одним столбцом
- Нескольких строк и столбцов

Тип возвращаемой подзапросом таблицы определяет, как можно ее использовать и какие операторы можно применять в содержащем выражении для взаимодействия с этой таблицей. По завершении выполнения содержащего выражения таблицы, возвращенные любым подзапросом, выгружаются из памяти. Таким образом, подзапрос действует как временная таблица, *областью видимости* которой является *выражение* (т. е. после завершения выполнения выражения сервер высвобождает всю память, отведенную под результаты подзапроса).

Предыдущие главы уже содержали несколько примеров подзапросов. Для начала приведем простой пример:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE account_id = (SELECT MAX(account_id) FROM account);
+------------+------------+---------+---------------+
```

```
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|         24 | SBL        |      13 |      50000.00 |
+------------+------------+---------+---------------+
1 row in set (0.65 sec)
```

В этом примере подзапрос возвращает максимальное значение столбца `account_id` таблицы `account`. Затем содержащее выражение возвращает данные по этому счету. Если возникают какие-нибудь вопросы по поводу того, что делает подзапрос, можно выполнить его отдельно (без скобок) и посмотреть, что он возвращает. Вот подзапрос из предыдущего примера:

```
mysql> SELECT MAX(account_id) FROM account;
+-----------------+
| MAX(account_id) |
+-----------------+
|              24 |
+-----------------+
1 row in set (0.00 sec)
```

Итак, подзапрос возвращает одну строку и один столбец. Это позволяет использовать его как одно из выражений в условии равенства (если бы подзапрос возвращал две или более строк, он мог бы *сравниваться* с чем-то, но не мог бы быть *равным* чему-то; более подробно об этом позже). В этом случае можно взять значение, возвращаемое подзапросом, и подставить его в правую часть условия фильтрации в основном запросе:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account a
    -> WHERE account_id = 24;
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|         24 | SBL        |      13 |      50000.00 |
+------------+------------+---------+---------------+
1 row in set (0.02 sec)
```

Здесь удобно использовать подзапрос, потому что он позволяет извлекать информацию о счете с наибольшим порядковым номером одним запросом. В противном случае пришлось бы с помощью одного запроса получать максимальный `account_id` и затем писать второй запрос для выбора необходимых данных из таблицы `account`. Как вы увидите, подзапросы полезны и во многих других ситуациях и могут стать одним из самых мощных инструментов в вашем наборе SQL-инструментов.

Типы подзапросов

Наряду с различиями, отмеченными ранее относительно типа возвращаемой подзапросом таблицы (одна строка/один столбец, одна стро-

ка/несколько столбцов или несколько столбцов), есть и другой показатель, по которому можно дифференцировать подзапросы. Некоторые подзапросы полностью самодостаточны (называются *несвязанными подзапросами* (*noncorrelated subqueries*)), тогда как другие ссылаются на столбцы содержащего выражения (называются *связанными подзапросами* (*correlated subqueries*)). В нескольких следующих разделах рассмотрены эти два типа подзапросов и приведены разные операторы, позволяющие взаимодействовать с ними.

Несвязанные подзапросы

Приведенный ранее в этой главе пример является несвязанным подзапросом. Он может выполняться самостоятельно и не использует ничего из содержащего выражения. Большинство подзапросов являются несвязанными. Только выражения `update` или `delete` часто используют связанные подзапросы (более подробно об этом позже). Упомянутый пример не только является несвязанным, но и возвращает таблицу, состоящую всего из одной строки и одного столбца. Такой тип подзапроса называется *скалярным подзапросом* (*scalar subquery*), и его можно помещать в любую часть условия, использующего обычные операторы (=, <>, <, >, <=, >=). Следующие примеры показывают применение скалярного подзапроса в условии неравенства:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE open_emp_id <> (SELECT e.emp_id
    ->   FROM employee e INNER JOIN branch b
    ->     ON e.assigned_branch_id = b.branch_id
    ->   WHERE e.title = 'Head Teller' AND b.city = 'Woburn');
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|          6 | CHK        |       3 |       1057.75 |
|          7 | MM         |       3 |       2212.50 |
|          8 | CHK        |       4 |        534.12 |
|          9 | SAV        |       4 |        767.77 |
|         10 | MM         |       4 |       5487.09 |
|         11 | CHK        |       5 |       2237.97 |
|         12 | CHK        |       6 |        122.37 |
|         13 | CD         |       6 |      10000.00 |
|         15 | CHK        |       8 |       3487.19 |
|         16 | SAV        |       8 |        387.99 |
|         17 | CHK        |       9 |        125.67 |
|         18 | MM         |       9 |       9345.55 |
|         19 | CD         |       9 |       1500.00 |
|         20 | CHK        |      10 |      23575.12 |
|         21 | BUS        |      10 |          0.00 |
|         23 | CHK        |      12 |      38552.05 |
|         24 | SBL        |      13 |      50000.00 |
```

```
+------------+------------+---------+---------------+
17 rows in set (0.00 sec)
```

Этот запрос возвращает данные по всем счетам, которые были открыты операционистом отделения Woburn, который *не* является старшим (подзапрос написан в предположении, что в отделении только один старший операционист). Подзапрос в этом примере немного сложнее, чем в предыдущем, – он соединяет две таблицы и включает два условия фильтрации. Подзапросы могут быть простыми или сложными настолько, насколько требуется. Они могут использовать любые из всех доступных блоков запроса (`select`, `from`, `where`, `group by`, `having`, `order by`).

Если при использовании в условии равенства подзапрос возвращает более одной строки, будет сформирована ошибка. Например, если предыдущий запрос изменить так, чтобы по подзапросу возвращались *все* операционисты отделения Woburn, а не только старший, будет получена следующая ошибка:

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE open_emp_id <> (SELECT e.emp_id
    ->   FROM employee e INNER JOIN branch b
    ->     ON e.assigned_branch_id = b.branch_id
    ->   WHERE e.title = 'Teller' AND b.city = 'Woburn');
ERROR 1242 (21000): Subquery returns more than 1 row
```

Если выполнить только подзапрос, результаты будут такими:

```
mysql> SELECT e.emp_id
    -> FROM employee e INNER JOIN branch b
    ->   ON e.assigned_branch_id = b.branch_id
    -> WHERE e.title = 'Teller' AND b.city = 'Woburn';
+--------+
| emp_id |
+--------+
|     11 |
|     12 |
+--------+
2 rows in set (0.02 sec)
```

Причина сбоя основного запроса в том, что выражение (`open_emp_id`) не может быть приравнено набору выражений (`emp_id` 11 и 12). Другими словами, единичный элемент не может приравниваться множеству. В следующем разделе вы увидите, как решить эту проблему с помощью другого оператора.

Подзапросы, возвращающие несколько строк и один столбец

Если подзапрос возвращает более одной строки, его нельзя использовать как одну из частей условия равенства, что и было продемонстри-

ровано предыдущим примером. Однако есть четыре дополнительных оператора, позволяющие строить условия с подзапросами этих типов.

Оператор in

Нельзя *приравнять* одно значение набору значений, но можно проверить *наличие* этого значения *в* наборе. Следующий пример, хотя и без подзапроса, показывает, как можно создать условие, использующее оператор `in` (в) для поиска значения в наборе значений:

```
mysql> SELECT branch_id, name, city
    -> FROM branch
    -> WHERE name IN ('Headquarters', 'Quincy Branch');
+-----------+---------------+---------+
| branch_id | name          | city    |
+-----------+---------------+---------+
|         1 | Headquarters  | Waltham |
|         3 | Quincy Branch | Quincy  |
+-----------+---------------+---------+
2 rows in set (0.03 sec)
```

В левой части условия – выражение (столбец `name`), а в правой части – набор строк. Оператор `in` проверяет, нет ли в столбце `name` одной из заданных строк. Если есть, условие выполнено, и строка добавляется в результирующий набор. Такие же результаты можно получить и с помощью двух условий равенства:

```
mysql> SELECT branch_id, name, city
    -> FROM branch
    -> WHERE name = 'Headquarters' OR name = 'Quincy Branch';
+-----------+---------------+---------+
| branch_id | name          | city    |
+-----------+---------------+---------+
|         1 | Headquarters  | Waltham |
|         3 | Quincy Branch | Quincy  |
+-----------+---------------+---------+
2 rows in set (0.01 sec)
```

Для набора только из двух выражений такой подход кажется рациональным, но если в наборе десятки (или сотни, тысячи и т. д.) значений, очевидный выбор – одно условие с оператором `in`.

Даже иногда создавая вручную наборы строк, дат или чисел для использования в одной из частей условия, вы все же предпочтете формировать набор при выполнении запроса посредством подзапроса, возвращающего одну или более строк. Следующий запрос использует оператор `in` и подзапрос в правой части условия фильтрации для того, чтобы выявить руководящий состав банка:

```
mysql> SELECT emp_id, fname, lname, title
    -> FROM employee
    -> WHERE emp_id IN (SELECT superior_emp_id
    ->   FROM employee);
```

```
+--------+---------+-----------+--------------------+
| emp_id | fname   | lname     | title              |
+--------+---------+-----------+--------------------+
|      1 | Michael | Smith     | President          |
|      3 | Robert  | Tyler     | Treasurer          |
|      4 | Susan   | Hawthorne | Operations Manager |
|      6 | Helen   | Fleming   | Head Teller        |
|     10 | Paula   | Roberts   | Head Teller        |
|     13 | John    | Blake     | Head Teller        |
|     16 | Theresa | Markham   | Head Teller        |
+--------+---------+-----------+--------------------+
7 rows in set (0.01 sec)
```

Подзапрос возвращает ID всех сотрудников, имеющих кого-то в подчинении, а основной запрос извлекает для этих сотрудников четыре столбца таблицы `employee`. Вот результаты подзапроса:

```
mysql> SELECT superior_emp_id
    -> FROM employee;
+-----------------+
| superior_emp_id |
+-----------------+
|            NULL |
|               1 |
|               1 |
|               3 |
|               4 |
|               4 |
|               4 |
|               4 |
|               4 |
|               6 |
|               6 |
|               6 |
|              10 |
|              10 |
|              13 |
|              13 |
|              16 |
|              16 |
+-----------------+
18 rows in set (0.00 sec)
```

Как видите, ID некоторых сотрудников встречаются по нескольку раз, поскольку у них более одного подчиненного. Это не оказывает негативного влияния на результаты основного запроса, потому что совершенно неважно, сколько раз встречается ID сотрудника в результирующем наборе подзапроса. Конечно, если вас беспокоят дублирующие значения в возвращаемой подзапросом таблице, можно добавить в блок `select` подзапроса ключевое слово `distinct`, но это никак не отразится на результирующем наборе основного запроса.

Можно проверять не только наличие значения в наборе значений, но и его отсутствие. Делается это с помощью оператора `not in` (нет в). Вот другой вариант предыдущего запроса с оператором `not in` вместо `in`:

```
mysql> SELECT emp_id, fname, lname, title
    -> FROM employee
    -> WHERE emp_id NOT IN (SELECT superior_emp_id
    ->   FROM employee
    ->   WHERE superior_emp_id IS NOT NULL);
+--------+----------+----------+----------------+
| emp_id | fname    | lname    | title          |
+--------+----------+----------+----------------+
|      2 | Susan    | Barker   | Vice President |
|      5 | John     | Gooding  | Loan Manager   |
|      7 | Chris    | Tucker   | Teller         |
|      8 | Sarah    | Parker   | Teller         |
|      9 | Jane     | Grossman | Teller         |
|     11 | Thomas   | Ziegler  | Teller         |
|     12 | Samantha | Jameson  | Teller         |
|     14 | Cindy    | Mason    | Teller         |
|     15 | Frank    | Portman  | Teller         |
|     17 | Beth     | Fowler   | Teller         |
|     18 | Rick     | Tulman   | Teller         |
+--------+----------+----------+----------------+
11 rows in set (0.00 sec)
```

Этот запрос находит всех сотрудников, которые никем *не* руководят. Здесь потребовалось добавить в подзапрос условие фильтрации, чтобы гарантировать отсутствие значений `null` в возвращаемой подзапросом таблице. В следующем разделе объясняется, почему в данном случае понадобился этот фильтр.

Оператор all

Оператор `in` используется для поиска выражения в наборе выражений, а оператор `all` (все) позволяет проводить сравнение одиночного значения с каждым значением набора. Для построения такого условия, помимо оператора `all`, понадобится один из операторов сравнения (`=`, `<>`, `<`, `>` и т. д.). Например, следующий запрос находит всех сотрудников, ID которых не равен ни одному из ID руководителей:

```
mysql> SELECT emp_id, fname, lname, title
    -> FROM employee
    -> WHERE emp_id <> ALL (SELECT superior_emp_id
    ->   FROM employee
    ->   WHERE superior_emp_id IS NOT NULL);
+--------+----------+----------+----------------+
| emp_id | fname    | lname    | title          |
+--------+----------+----------+----------------+
|      2 | Susan    | Barker   | Vice President |
|      5 | John     | Gooding  | Loan Manager   |
```

```
|      7 | Chris    | Tucker   | Teller         |
|      8 | Sarah    | Parker   | Teller         |
|      9 | Jane     | Grossman | Teller         |
|     11 | Thomas   | Ziegler  | Teller         |
|     12 | Samantha | Jameson  | Teller         |
|     14 | Cindy    | Mason    | Teller         |
|     15 | Frank    | Portman  | Teller         |
|     17 | Beth     | Fowler   | Teller         |
|     18 | Rick     | Tulman   | Teller         |
+--------+----------+----------+----------------+
11 rows in set (0.05 sec)
```

И опять подзапрос возвращает набор ID сотрудников, имеющих подчиненных. Основной запрос возвращает данные для всех сотрудников, ID которых не равны ни одному возвращенному подзапросом ID. Иначе говоря, запрос находит всех сотрудников-«неруководителей». Если этот подход кажется вам несколько топорным, вы не одиноки; многие предпочли бы построить запрос по-другому, обойдясь без оператора `all`. Например, результаты этого запроса аналогичны последнему примеру с оператором `not in` из предыдущего раздела. Дело вкуса, но, думаю, что этим многим версия с `not in` просто кажется более понятной.

Сравнивать значения с набором значений с помощью операторов `not in` или `<> all` нужно аккуратно, убедившись, что в наборе нет значения `null`. Сервер приравнивает значение из левой части выражения к каждому члену набора, и любая попытка приравнять значение к `null` дает в результате `unknown`. Таким образом, следующий запрос возвратит пустой набор:

```
mysql> SELECT emp_id, fname, lname, title
    -> FROM employee
    -> WHERE emp_id NOT IN (1, 2, NULL);
Empty set (0.00 sec)
```

Бывают случаи, когда оператор `all` чуть более естественен. Следующий пример использует `all` для поиска счетов, доступный остаток которых меньше, чем на любом из счетов Фрэнка Такера (Frank Tucker):

```
mysql> SELECT account_id, cust_id, product_cd, avail_balance
    -> FROM account
    -> WHERE avail_balance < ALL (SELECT a.avail_balance
    ->   FROM account a INNER JOIN individual i
    ->     ON a.cust_id = i.cust_id
    ->   WHERE i.fname = 'Frank' AND i.lname = 'Tucker');
+------------+---------+------------+---------------+
| account_id | cust_id | product_cd | avail_balance |
+------------+---------+------------+---------------+
|          2 |       1 | SAV        |        500.00 |
|          5 |       2 | SAV        |        200.00 |
|          8 |       4 | CHK        |        534.12 |
|          9 |       4 | SAV        |        767.77 |
|         12 |       6 | CHK        |        122.37 |
```

```
|          16 |        8 | SAV         |         387.99 |
|          17 |        9 | CHK         |         125.67 |
|          21 |       10 | BUS         |           0.00 |
+-------------+----------+-------------+----------------+
8 rows in set (0.01 sec)
```

Вот таблица, возвращенная подзапросом. Она включает доступные остатки всех счетов Фрэнка:

```
mysql> SELECT a.avail_balance
    -> FROM account a INNER JOIN individual i
    ->   ON a.cust_id = i.cust_id
    -> WHERE i.fname = 'Frank' AND i.lname = 'Tucker';
+---------------+
| avail_balance |
+---------------+
|       1057.75 |
|       2212.50 |
+---------------+
2 rows in set (0.01 sec)
```

У Фрэнка два счета, минимальный остаток – 1057,75 долларов. Основной запрос находит все счета, остаток на которых меньше, чем на любом из счетов Фрэнка. Таким образом, результирующий набор включает все счета, остаток на которых меньше 1057,75 долларов.

Оператор any

Как и оператор `all`, оператор `any` (любой) обеспечивает возможность сравнивать значение с элементами набора значений. Однако, в отличие от `all`, условие, использующее оператор `any`, истинно (`true`), если есть хотя бы одно совпадение, тогда как при использовании оператора `all` требуется, чтобы условие выполнялось для *всех* элементов набора. Например, требуется найти все счета, доступный остаток которых больше, чем на *любом* из счетов Фрэнка Такера:

```
mysql> SELECT account_id, cust_id, product_cd, avail_balance
    -> FROM account
    -> WHERE avail_balance > ANY (SELECT a.avail_balance
    ->   FROM account a INNER JOIN individual i
    ->     ON a.cust_id = i.cust_id
    ->   WHERE i.fname = 'Frank' AND i.lname = 'Tucker');
+------------+---------+------------+---------------+
| account_id | cust_id | product_cd | avail_balance |
+------------+---------+------------+---------------+
|          3 |       1 | CD         |       3000.00 |
|          4 |       2 | CHK        |       2258.02 |
|          7 |       3 | MM         |       2212.50 |
|         10 |       4 | MM         |       5487.09 |
|         11 |       5 | CHK        |       2237.97 |
|         13 |       6 | CD         |      10000.00 |
|         14 |       7 | CD         |       5000.00 |
```

```
|          15 |        8 | CHK          |        3487.19 |
|          18 |        9 | MM           |        9345.55 |
|          19 |        9 | CD           |        1500.00 |
|          20 |       10 | CHK          |       23575.12 |
|          22 |       11 | BUS          |        9345.55 |
|          23 |       12 | CHK          |       38552.05 |
|          24 |       13 | SBL          |       50000.00 |
+-------------+----------+--------------+----------------+
14 rows in set (0.01 sec)
```

У Фрэнка два счета с остатками 1057,75 и 2212,50 долларов. Чтобы остаток был больше, чем на *любом* из этих двух счетов, на счете должно быть, по крайней мере, 1057,75 долларов.

Операторы `= any` и `in` эквивалентны, хотя многие предпочитают оператор `in`.

Подзапросы, возвращающие несколько столбцов

До сих пор все примеры подзапросов в данной главе возвращали один столбец и одну или более строк. Однако в определенных ситуациях можно использовать подзапросы, возвращающие два или более столбцов. Чтобы лучше разобраться в подзапросах, возвращающих несколько столбцов, полезно сначала взглянуть на пример использования нескольких подзапросов, возвращающих один столбец:

```
mysql> SELECT account_id, product_cd, cust_id
    -> FROM account
    -> WHERE open_branch_id = (SELECT branch_id
    ->   FROM branch
    ->   WHERE name = 'Woburn Branch')
    ->   AND open_emp_id IN (SELECT emp_id
    ->   FROM employee
    ->   WHERE title = 'Teller' OR title = 'Head Teller');
+------------+------------+---------+
| account_id | product_cd | cust_id |
+------------+------------+---------+
|          1 | CHK        |       1 |
|          2 | SAV        |       1 |
|          3 | CD         |       1 |
|          4 | CHK        |       2 |
|          5 | SAV        |       2 |
|         14 | CD         |       7 |
|         22 | BUS        |      11 |
+------------+------------+---------+
7 rows in set (0.00 sec)
```

Чтобы выявить ID отделения Woburn и ID всех банковских операционистов, этот запрос использует два подзапроса. Затем содержащий запрос использует эту информацию для выбора всех текущих счетов, от-

крытых старшим операционистом в отделении Woburn. Но в таблице employee есть информация об отделении, в котором числится каждый сотрудник, поэтому те же результаты можно получить путем сравнения столбцов account.open_branch_id и account.open_emp_id с единственным подзапросом к таблицам employee и branch. Для этого в условии фильтрации следует указать в круглых скобках имена обоих столбцов таблицы account в том же порядке, в каком они возвращаются подзапросом:

```
mysql> SELECT account_id, product_cd, cust_id
    -> FROM account
    -> WHERE (open_branch_id, open_emp_id) IN
    ->  (SELECT b.branch_id, e.emp_id
    ->   FROM branch b INNER JOIN employee e
    ->     ON b.branch_id = e.assigned_branch_id
    ->   WHERE b.name = 'Woburn Branch'
    ->     AND (e.title = 'Teller' OR e.title = 'Head Teller'));
+------------+------------+---------+
| account_id | product_cd | cust_id |
+------------+------------+---------+
|          1 | CHK        |       1 |
|          2 | SAV        |       1 |
|          3 | CD         |       1 |
|          4 | CHK        |       2 |
|          5 | SAV        |       2 |
|         14 | CD         |       7 |
|         22 | BUS        |      11 |
+------------+------------+---------+
7 rows in set (0.00 sec)
```

Эта версия запроса делает то же самое, что и предыдущий пример, но с помощью всего одного подзапроса, который возвращает два столбца, а не двух подзапросов, возвращающих по одному столбцу.

Конечно, можно было бы переписать предыдущий пример, просто соединив три таблицы, без этой возни с подзапросами. Но при изучении SQL полезно увидеть несколько путей достижения одного результата. Вот еще один пример, требующий применения подзапроса. Скажем, от клиентов поступило несколько жалоб, связанных с неверными значениями в столбцах доступного остатка и отложенного остатка (pending balance) таблицы account. Задача – найти все счета, остатки на которых не соответствуют суммам по операциям для этого счета. Вот частичное решение проблемы:

```
SELECT 'ALERT! : Account #1 Has Incorrect Balance!'
FROM account
WHERE (avail_balance, pending_balance) <>
 (SELECT SUM(<expression to generate available balance>),
    SUM(<expression to generate pending balance>)
  FROM transaction
  WHERE account_id = 1)
  AND account_id = 1;
```

Как видите, здесь нет выражений, суммирующих операции для вычисления доступного и отложенного остатков, но обещаю, что в главе 11, когда мы научимся создавать выражения `case`, все будет доработано. Но даже в таком виде запрос достаточно полон, чтобы увидеть, что подзапрос генерирует две суммы из таблицы `transaction`, которые потом сравниваются со столбцами `avail_balance` и `pending_balance` таблицы `account`. И подзапрос, и основной запрос включают условие фильтрации `account_id = 1`. Таким образом, запрос в его теперешней форме будет проверять только по одному счету за раз. В следующем разделе мы научимся создавать более общую форму запроса, которая будет проверять *все* счета за одно выполнение.

Связанные подзапросы

Все приведенные до сих пор запросы не зависели от своих содержащих выражений, т. е. могли выполняться самостоятельно и представлять свои результаты для проверки. *Связанный подзапрос* (*correlated subquery*), напротив, *зависит* от содержащего выражения, из которого он ссылается на один или более столбцов. В отличие от несвязанного подзапроса, который выполняется непосредственно перед выполнением содержащего выражения, связанный подзапрос выполняется по разу для каждой строки-кандидата (это строки, которые предположительно могут быть включены в окончательные результаты). Например, следующий запрос использует связанный подзапрос для подсчета количества счетов у каждого клиента. Затем основной запрос выбирает тех клиентов, у которых ровно по два счета:

```
mysql> SELECT c.cust_id, c.cust_type_cd, c.city
    -> FROM customer c
    -> WHERE 2 = (SELECT COUNT(*)
    ->   FROM account a
    ->   WHERE a.cust_id = c.cust_id);
+---------+--------------+---------+
| cust_id | cust_type_cd | city    |
+---------+--------------+---------+
|       2 | I            | Woburn  |
|       3 | I            | Quincy  |
|       6 | I            | Waltham |
|       8 | I            | Salem   |
|      10 | B            | Salem   |
+---------+--------------+---------+
5 rows in set (0.01 sec)
```

Ссылка на `c.cust_id` в самом конце подзапроса – это то, что делает этот подзапрос связанным. Чтобы подзапрос мог выполняться, основной запрос должен поставлять значения для `c.cust_id`. В данном случае основной запрос извлекает из таблицы `customer` все 13 строк и выполняет по одному подзапросу для всех клиентов, передавая в него соответст-

вующий ID клиента при каждом выполнении. Если подзапрос возвращает значение 2, условие фильтрации выполняется и строка добавляется в результирующий набор.

Помимо условий равенства связанные подзапросы можно применять в условиях других типов, таких как условие вхождения в диапазон, проиллюстрированное ниже:

```
mysql> SELECT c.cust_id, c.cust_type_cd, c.city
    -> FROM customer c
    -> WHERE (SELECT SUM(a.avail_balance)
    ->     FROM account a
    ->     WHERE a.cust_id = c.cust_id)
    ->   BETWEEN 5000 AND 10000;
+---------+--------------+------------+
| cust_id | cust_type_cd | city       |
+---------+--------------+------------+
|       4 | I            | Waltham    |
|       7 | I            | Wilmington |
|      11 | B            | Wilmington |
+---------+--------------+------------+
3 rows in set (0.02 sec)
```

Этот вариант приведенного ранее запроса находит всех клиентов, чей общий доступный остаток по всем счетам находится в диапазоне от 5000 до 10 000 долларов. И снова связанный подзапрос выполняется 13 раз (по разу для каждой строки), и каждое выполнение подзапроса возвращает общий остаток по счетам данного клиента.

В конце предыдущего раздела было продемонстрировано, как проверять доступный и отложенный остатки счета по транзакциям, зарегистрированным по данному счету, и я обещал показать, как изменить пример для обработки всех счетов за одно выполнение. Вот тот пример:

```
SELECT 'ALERT! : Account #1 Has Incorrect Balance!'
FROM account
WHERE (avail_balance, pending_balance) <>
 (SELECT SUM(<expression to generate available balance>),
    SUM(<expression to generate pending balance>)
  FROM transaction
  WHERE account_id = 1)
  AND account_id = 1;
```

При использовании связанного подзапроса вместо несвязанного основной запрос может выполняться всего лишь один раз, а подзапрос будет выполняться для каждого счета. Вот обновленная версия:

```
SELECT CONCAT('ALERT! : Account #', a.account_id,
  ' Has Incorrect Balance!')
FROM account a
WHERE (a.avail_balance, a.pending_balance) <>
 (SELECT SUM(<expression to generate available balance>),
    SUM(<expression to generate pending balance>)
```

```
    FROM transaction t
    WHERE t.account_id = a.account_id);
```

Подзапрос теперь включает условие фильтрации, связывающее ID счета транзакции и ID счета из основного запроса. Изменился и блок `select` – теперь вместо жестко запрограммированного значения `1` он путем конкатенации формирует предупреждение, включающее ID счета.

Оператор exists

Связанные подзапросы часто используются в условиях равенства и вхождения в диапазон, но самый распространенный оператор, применяемый в условиях со связанными подзапросами, – это оператор `exists` (существует). Оператор `exists` применяется, если требуется показать, что связь есть, а количество связей при этом не имеет значения. Например, следующий запрос находит все счета, для которых транзакция была выполнена в определенный день, без учета количества транзакций:

```
SELECT a.account_id, a.product_cd, a.cust_id, a.avail_balance
FROM account a
WHERE EXISTS (SELECT 1
  FROM transaction t
  WHERE t.account_id = a.account_id
    AND t.txn_date = '2005-01-22');
```

При использовании оператора `exists` подзапрос может возвращать ни одной, одну или много строк, а условие просто проверяет, возвращены ли в результате выполнения подзапроса строки (все равно сколько). Если взглянуть на блок `select` подзапроса, можно увидеть, что он состоит из единственного литерала (`1`); для условия основного запроса имеет значение только число возвращенных строк, а что именно было возвращено подзапросом – не важно. Подзапрос может возвращать все, что вам вздумается, как показывает следующий пример:

```
SELECT a.account_id, a.product_cd, a.cust_id, a.avail_balance
FROM account a
WHERE EXISTS (SELECT t.txn_id, 'hello', 3.1415927
  FROM transaction t
  WHERE t.account_id = a.account_id
    AND t.txn_date = '2005-01-22');
```

Но все же при использовании `exists` принято задавать `select 1` или `select *`.

Для поиска подзапросов, не возвращающих строки, можно использовать и оператор `not exists`:

```
mysql> SELECT a.account_id, a.product_cd, a.cust_id
    -> FROM account a
    -> WHERE NOT EXISTS (SELECT 1
    ->   FROM business b
    ->   WHERE b.cust_id = a.cust_id);
```

```
+------------+------------+---------+
| account_id | product_cd | cust_id |
+------------+------------+---------+
|          1 | CHK        |       1 |
|          2 | SAV        |       1 |
|          3 | CD         |       1 |
|          4 | CHK        |       2 |
|          5 | SAV        |       2 |
|          6 | CHK        |       3 |
|          7 | MM         |       3 |
|          8 | CHK        |       4 |
|          9 | SAV        |       4 |
|         10 | MM         |       4 |
|         11 | CHK        |       5 |
|         12 | CHK        |       6 |
|         13 | CD         |       6 |
|         14 | CD         |       7 |
|         15 | CHK        |       8 |
|         16 | SAV        |       8 |
|         17 | CHK        |       9 |
|         18 | MM         |       9 |
|         19 | CD         |       9 |
+------------+------------+---------+
19 rows in set (0.04 sec)
```

Этот запрос выявляет всех клиентов, ID которых нет в таблице `business`, – окольный путь для поиска всех клиентов-физических лиц.

Манипулирование данными с помощью связанных подзапросов

До сих пор в этой главе в качестве примеров приводились только выражения `select`, но это не значит, что в других SQL-выражениях подзапросы не используются. Они также широко задействуются в выражениях `update`, `delete` и `insert`, а связанные подзапросы часто применяются в выражениях `update` и `delete`. Вот пример связанного подзапроса, с помощью которого изменяется столбец `last_activity_date` таблицы `account`:

```
UPDATE account a
SET a.last_activity_date =
 (SELECT MAX(t.txn_date)
  FROM transaction t
  WHERE t.account_id = a.account_id);
```

Это выражение корректирует все строки таблицы `account` (поскольку блока `where` нет), выбирая дату последней операции для каждого счета. Хотя кажется разумным ожидать, что для каждого счета будет существовать, по крайней мере, одна связанная с ним операция, но лучше проверить наличие такой операции, прежде чем пытаться обновить столбец `last_activity_date`. В противном случае в столбце появится

значение null, поскольку по подзапросу не будет возвращено ни одной строки. Вот другой вариант выражения update, на этот раз использующий блок where со вторым связанным подзапросом:

```
UPDATE account a
SET a.last_activity_date =
 (SELECT MAX(t.txn_date)
  FROM transaction t
  WHERE t.account_id = a.account_id)
WHERE EXISTS (SELECT 1
  FROM transaction t
  WHERE t.account_id = a.account_id);
```

Эти два связанных подзапроса идентичны, за исключением блоков select. Однако подзапрос блока set выполняется, только если условие блока where выражения update истинно (true) (т. е. для счета была найдена, по крайней мере, одна операция). Таким образом данные столбца last_activity_date защищены от перезаписи значением null.

Связанные подзапросы обычны и в выражениях delete. Например, в конце каждого месяца запускается сценарий, уничтожающий ненужные данные. Этот сценарий может включать следующее выражение, которое удаляет из таблицы department данные, не имеющие дочерних строк в таблице employee:

```
DELETE FROM department
WHERE NOT EXISTS (SELECT 1
  FROM employee
  WHERE employee.dept_id = department.dept_id);
```

При использовании связанных подзапросов в выражениях delete в MySQL необходимо помнить, что псевдонимы таблиц не допускаются ни в коем случае. Вот почему в этом подзапросе приходилось использовать полное имя таблицы. Для большинства других серверов БД можно было бы снабдить таблицы department и employee псевдонимами:

```
DELETE FROM department d
WHERE NOT EXISTS (SELECT 1
  FROM employee e
  WHERE e.dept_id = d.dept_id);
```

Использование подзапросов

Теперь, когда вы знакомы с разными типами подзапросов и разными операторами, с помощью которых можно взаимодействовать с таблицами, возвращенными подзапросами, пора рассмотреть множество способов использования подзапросов для построения мощных SQL-выражений. В следующих трех разделах будет продемонстрировано, как подзапросы могут участвовать в построении специальных таблиц, создании условий и формировании столбцов значений в результирующих наборах.

Подзапросы как источники данных

Ранее в главе 3 говорилось, что в блоке `from` выражения `select` указываются *таблицы*, которые будут использоваться запросом. Поскольку подзапрос формирует таблицу, содержащую строки и столбцы данных, абсолютно допустимо включать подзапросы в блок `from`. Хотя на первый взгляд это может показаться любопытной возможностью без особых преимуществ, но использование подзапросов в качестве таблиц – один из самых мощных инструментов, доступных при написании запросов. Вот простой пример:

```
mysql> SELECT d.dept_id, d.name, e_cnt.how_many num_employees
    -> FROM department d INNER JOIN
    ->  (SELECT dept_id, COUNT(*) how_many
    ->   FROM employee
    ->   GROUP BY dept_id) e_cnt
    ->   ON d.dept_id = e_cnt.dept_id;
+---------+----------------+---------------+
| dept_id | name           | num_employees |
+---------+----------------+---------------+
|       1 | Operations     |            14 |
|       2 | Loans          |             1 |
|       3 | Administration |             3 |
+---------+----------------+---------------+
3 rows in set (0.04 sec)
```

В этом примере подзапрос формирует список ID отделов с указанием количества сотрудников, приписанных к каждому отделу. Вот таблица, сформированная подзапросом:

```
mysql> SELECT dept_id, COUNT(*) how_many
    -> FROM employee
    -> GROUP BY dept_id;
+---------+----------+
| dept_id | how_many |
+---------+----------+
|       1 |       14 |
|       2 |        1 |
|       3 |        3 |
+---------+----------+
3 rows in set (0.00 sec)
```

Подзапрос назван `e_cnt` и соединен с таблицей `department` по столбцу `dept_id`. После этого основной запрос извлекает ID отдела и имя из таблицы `department`, а также количество сотрудников из подзапроса `e_cnt`.

Подзапросы, используемые для формирования таблиц, должны быть несвязанными; они выполняются первыми, и полученные таблицы поддерживаются в памяти, пока основной запрос завершает выполнение. Подзапросы предлагают необычайную гибкость при написании запросов. С их помощью можно выходить далеко за пределы набора доступных таблиц, создавать практически любое представление необ-

ходимых данных и затем соединять эту таблицу с другими таблицами или таблицами, сформированными подзапросами. Теперь создать отчет или сформировать набор данных для внешних систем можно с помощью единственного запроса. Раньше для этого требовалось несколько запросов или использование процедурного языка.

Формирование таблиц

С помощью подзапросов можно как резюмировать имеющиеся данные, так и формировать данные, которых в БД нет ни в какой форме. Например, требуется сгруппировать клиентов по денежным суммам, размещенным на депозитных счетах, но использовать для этого описания групп, не хранящиеся в БД. Скажем, надо разбить клиентов на следующие группы:

Группа	Нижний предел (долларов)	Верхний предел (долларов)
`Small Fry` (мелкота)	0	4 999,99
`Average Joes` (середняки)	5 000	9 999,99
`Heavy Hitters` (тяжеловесы)	10 000	9 999 999,99

Чтобы сформировать эти группы в рамках одного запроса, потребуется способ определения этих трех групп. Первый шаг – создать запрос, формирующий описания групп:

```
mysql> SELECT 'Small Fry' name, 0 low_limit, 4999.99 high_limit
    -> UNION ALL
    -> SELECT 'Average Joes' name, 5000 low_limit, 9999.99 high_limit
    -> UNION ALL
    -> SELECT 'Heavy Hitters' name, 10000 low_limit, 9999999.99 high_limit;
+---------------+-----------+------------+
| name          | low_limit | high_limit |
+---------------+-----------+------------+
| Small Fry     |         0 |    4999.99 |
| Average Joes  |      5000 |    9999.99 |
| Heavy Hitters |     10000 | 9999999.99 |
+---------------+-----------+------------+
3 rows in set (0.00 sec)
```

Здесь с помощью оператора для работы с наборами `union all` (объединить все) результаты трех отдельных запросов сводятся в один результирующий набор. Каждый запрос получает три литерала. Результаты этих трех запросов объединяются для формирования таблицы, состоящей из трех строк и трех столбцов. Теперь у нас есть запрос для формирования необходимых групп. Его можно поместить в блок `from` другого запроса для формирования групп клиентов:

```
mysql> SELECT groups.name, COUNT(*) num_customers
    -> FROM
    ->  (SELECT SUM(a.avail_balance) cust_balance
```

```
    ->   FROM account a INNER JOIN product p
    ->     ON a.product_cd = p.product_cd
    ->   WHERE p.product_type_cd = 'ACCOUNT'
    ->   GROUP BY a.cust_id) cust_rollup INNER JOIN
    ->  (SELECT 'Small Fry' name, 0 low_limit, 4999.99 high_limit
    ->   UNION ALL
    ->   SELECT 'Average Joes' name, 5000 low_limit,
    ->     9999.99 high_limit
    ->   UNION ALL
    ->   SELECT 'Heavy Hitters' name, 10000 low_limit,
    ->     9999999.99 high_limit) groups
    ->   ON cust_rollup.cust_balance
    ->     BETWEEN groups.low_limit AND groups.high_limit
    -> GROUP BY groups.name;
+---------------+---------------+
| name          | num_customers |
+---------------+---------------+
| Average Joes  |             2 |
| Heavy Hitters |             4 |
| Small Fry     |             5 |
+---------------+---------------+
3 rows in set (0.01 sec)
```

В блоке `from` имеется два подзапроса: первый подзапрос, `cust_rollup`, возвращает общий остаток по депозитным счетам для каждого клиента, а второй подзапрос, `groups`, формирует таблицу, содержащую три группы клиентов. Вот таблица, сгенерированная подзапросом `cust_rollup`:

```
mysql> SELECT SUM(a.avail_balance) cust_balance
    -> FROM account a INNER JOIN product p
    ->   ON a.product_cd = p.product_cd
    -> WHERE p.product_type_cd = 'ACCOUNT'
    -> GROUP BY a.cust_id;
+--------------+
| cust_balance |
+--------------+
|      4557.75 |
|      2458.02 |
|      3270.25 |
|      6788.98 |
|      2237.97 |
|     10122.37 |
|      5000.00 |
|      3875.18 |
|     10971.22 |
|     23575.12 |
|     38552.05 |
+--------------+
11 rows in set (0.05 sec)
```

Затем таблица, сгенерированная подзапросом `cust_rollup`, соединяется с таблицей `groups` посредством условия вхождения в диапазон (`cust_rol-`

lup.cust_balance BETWEEN groups.low_limit AND groups.high_limit). Наконец, соединенные данные группируются и подсчитывается число клиентов в каждой группе для формирования окончательного результирующего набора.

Конечно, можно было бы не использовать подзапрос, а просто создать постоянную таблицу для хранения описаний групп. При таком подходе через некоторое время БД изобиловала бы небольшими специальными таблицами, причины появления которых мало кто помнил бы. Мне приходилось работать в средах, где пользователям БД позволялось создавать собственные таблицы для специальных целей. Результаты были просто губительными (таблицы, не включенные в резервные копии; таблицы, потерянные при обновлениях сервера; простои сервера из-за проблем распределения памяти и т. д.). Однако, вооружившись запросами, можно придерживаться политики, при которой таблицы добавляются в БД, только если есть очевидная необходимость хранения новых данных.

Подзапросы, ориентированные на задачи

В системах, используемых для создания отчетов или наборов данных, часто встречаются следующие запросы:

```
mysql> SELECT p.name product, b.name branch,
    ->   CONCAT(e.fname, ' ', e.lname) name,
    ->   SUM(a.avail_balance) tot_deposits
    -> FROM account a INNER JOIN employee e
    ->   ON a.open_emp_id = e.emp_id
    ->   INNER JOIN branch b
    ->   ON a.open_branch_id = b.branch_id
    ->   INNER JOIN product p
    ->   ON a.product_cd = p.product_cd
    -> WHERE p.product_type_cd = 'ACCOUNT'
    -> GROUP BY p.name, b.name, e.fname, e.lname;
+-----------------------+---------------+-----------------+--------------+
| product               | branch        | name            | tot_deposits |
+-----------------------+---------------+-----------------+--------------+
| certificate of deposit | Headquarters  | Michael Smith   |     11500.00 |
| certificate of deposit | Woburn Branch | Paula Roberts   |      8000.00 |
| checking account       | Headquarters  | Michael Smith   |       782.16 |
| checking account       | Quincy Branch | John Blake      |      1057.75 |
| checking account       | So. NH Branch | Theresa Markham |     67852.33 |
| checking account       | Woburn Branch | Paula Roberts   |      3315.77 |
| money market account   | Headquarters  | Michael Smith   |     14832.64 |
| money market account   | Quincy Branch | John Blake      |      2212.50 |
| savings account        | Headquarters  | Michael Smith   |       767.77 |
| savings account        | So. NH Branch | Theresa Markham |       387.99 |
| savings account        | Woburn Branch | Paula Roberts   |       700.00 |
+-----------------------+---------------+-----------------+--------------+
11 rows in set (0.02 sec)
```

Этот запрос суммирует все остатки депозитных счетов по типу счета, сотруднику, открывшему счета, и отделениям, в которых были открыты счета. Если внимательнее посмотреть на запрос, увидим, что таблицы product, branch и employee нужны только в целях отображения и что все необходимое для группировки (product_cd, open_branch_id, open_emp_id и avail_balance) есть в таблице account. Поэтому задачу по формированию групп можно было бы выделить в подзапрос, а затем для получения нужного результата соединить остальные три таблицы с таблицей, сгенерированной подзапросом. Вот подзапрос группировки:

```
mysql> SELECT product_cd, open_branch_id branch_id, open_emp_id emp_id,
    ->   SUM(avail_balance) tot_deposits
    -> FROM account
    -> GROUP BY product_cd, open_branch_id, open_emp_id;
+------------+-----------+--------+--------------+
| product_cd | branch_id | emp_id | tot_deposits |
+------------+-----------+--------+--------------+
| BUS        |         2 |     10 |      9345.55 |
| BUS        |         4 |     16 |         0.00 |
| CD         |         1 |      1 |     11500.00 |
| CD         |         2 |     10 |      8000.00 |
| CHK        |         1 |      1 |       782.16 |
| CHK        |         2 |     10 |      3315.77 |
| CHK        |         3 |     13 |      1057.75 |
| CHK        |         4 |     16 |     67852.33 |
| MM         |         1 |      1 |     14832.64 |
| MM         |         3 |     13 |      2212.50 |
| SAV        |         1 |      1 |       767.77 |
| SAV        |         2 |     10 |       700.00 |
| SAV        |         4 |     16 |       387.99 |
| SBL        |         3 |     13 |     50000.00 |
+------------+-----------+--------+--------------+
14 rows in set (0.01 sec)
```

Это – сердце запроса; все остальные таблицы нужны только для того, чтобы обеспечить осмысленные строки для шапки таблицы вместо имен столбцов внешних ключей product_cd, open_branch_id и open_emp_id. Следующий запрос включает запрос к таблице account в качестве подзапроса и соединяет результирующую таблицу с тремя остальными таблицами:

```
mysql> SELECT p.name product, b.name branch,
    ->   CONCAT(e.fname, ' ', e.lname) name,
    ->   account_groups.tot_deposits
    -> FROM
    ->  (SELECT product_cd, open_branch_id branch_id,
    ->     open_emp_id emp_id,
    ->     SUM(avail_balance) tot_deposits
    ->   FROM account
    ->   GROUP BY product_cd, open_branch_id, open_emp_id) account_groups
    ->   INNER JOIN employee e ON e.emp_id = account_groups.emp_id
    ->   INNER JOIN branch b ON b.branch_id = account_groups.branch_id
```

```
    ->   INNER JOIN product p ON p.product_cd = account_groups.product_cd
    -> WHERE p.product_type_cd = 'ACCOUNT';
+------------------------+---------------+-----------------+--------------+
| product                | branch        | name            | tot_deposits |
+------------------------+---------------+-----------------+--------------+
| certificate of deposit | Headquarters  | Michael Smith   |     11500.00 |
| checking account       | Headquarters  | Michael Smith   |       782.16 |
| money market account   | Headquarters  | Michael Smith   |     14832.64 |
| savings account        | Headquarters  | Michael Smith   |       767.77 |
| certificate of deposit | Woburn Branch | Paula Roberts   |      8000.00 |
| checking account       | Woburn Branch | Paula Roberts   |      3315.77 |
| savings account        | Woburn Branch | Paula Roberts   |       700.00 |
| checking account       | Quincy Branch | John Blake      |      1057.75 |
| money market account   | Quincy Branch | John Blake      |      2212.50 |
| checking account       | So. NH Branch | Theresa Markham |     67852.33 |
| savings account        | So. NH Branch | Theresa Markham |       387.99 |
+------------------------+---------------+-----------------+--------------+
11 rows in set (0.00 sec)
```

Я понимаю, что «на вкус и цвет товарищей нет», но этот вариант запроса мне нравится намного больше, чем большая плоская версия. И он может быстрее выполняться, потому что группировка проведена по небольшим столбцам внешних ключей (`product_cd`, `open_branch_id`, `open_emp_id`), а не по столбцам, предположительно содержащим длинные строки (`branch.name`, `product.name`, `employee.fname`, `employee.lname`).

Подзапросы в условиях фильтрации

Во многих примерах данной главы подзапросы используются как выражения в условиях фильтрации, поэтому для вас не будет сюрпризом, что это одно из основных применений подзапросов. Но условия фильтрации, использующие подзапросы, встречаются не только в блоке `where`. Например, следующий запрос использует блок `having` для поиска сотрудника, открывшего наибольшее количество счетов:

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id
    -> HAVING COUNT(*) = (SELECT MAX(emp_cnt.how_many)
    ->   FROM (SELECT COUNT(*) how_many
    ->     FROM account
    ->     GROUP BY open_emp_id) emp_cnt);
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|           1 |        8 |
+-------------+----------+
1 row in set (0.01 sec)
```

Подзапрос блока `having` находит максимальное число счетов, открытых одним сотрудником, а основной запрос находит сотрудника, открывшего это количество счетов. Если бы с наибольшим числом от-

крытых счетов были связаны несколько сотрудников, запрос возвратил бы несколько строк.

Подзапросы как генераторы выражений

В этом последнем разделе главы я завершу тему, с которой начал, – скалярные подзапросы, возвращающие один столбец и одну строку. Кроме условий фильтрации скалярные подзапросы применимы везде, где может появляться выражение, включая блоки `select` и `order by` запроса и блок `values` (значения) выражения `insert`.

Ранее в этой главе, в разделе «Подзапросы, ориентированные на задачи», было показано, как с помощью подзапроса отделить механизм группировки от остального запроса. Вот вариант того же запроса, использующий подзапросы с той же целью, но по-другому:

```
mysql> SELECT
    ->  (SELECT p.name FROM product p
    ->   WHERE p.product_cd = a.product_cd
    ->     AND p.product_type_cd = 'ACCOUNT') product,
    ->  (SELECT b.name FROM branch b
    ->   WHERE b.branch_id = a.open_branch_id) branch,
    ->  (SELECT CONCAT(e.fname, ' ', e.lname) FROM employee e
    ->   WHERE e.emp_id = a.open_emp_id) name,
    ->   SUM(a.avail_balance) tot_deposits
    -> FROM account a
    -> GROUP BY a.product_cd, a.open_branch_id, a.open_emp_id;
+-----------------------+---------------+-----------------+--------------+
| product               | branch        | name            | tot_deposits |
+-----------------------+---------------+-----------------+--------------+
| NULL                  | Woburn Branch | Paula Roberts   |      9345.55 |
| NULL                  | So. NH Branch | Theresa Markham |         0.00 |
| certificate of deposit | Headquarters  | Michael Smith   |     11500.00 |
| certificate of deposit | Woburn Branch | Paula Roberts   |      8000.00 |
| checking account      | Headquarters  | Michael Smith   |       782.16 |
| checking account      | Woburn Branch | Paula Roberts   |      3315.77 |
| checking account      | Quincy Branch | John Blake      |      1057.75 |
| checking account      | So. NH Branch | Theresa Markham |     67852.33 |
| money market account  | Headquarters  | Michael Smith   |     14832.64 |
| money market account  | Quincy Branch | John Blake      |      2212.50 |
| savings account       | Headquarters  | Michael Smith   |       767.77 |
| savings account       | Woburn Branch | Paula Roberts   |       700.00 |
| savings account       | So. NH Branch | Theresa Markham |       387.99 |
| NULL                  | Quincy Branch | John Blake      |     50000.00 |
+-----------------------+---------------+-----------------+--------------+
14 rows in set (0.01 sec)
```

Между этим запросом и приведенной ранее версией, использующей подзапрос в блоке `from`, есть два основных различия:

- Вместо соединения таблиц `product`, `branch` и `employee` с данными счета в блоке `select` используются связанные скалярные подзапросы для поиска типа счета, отделения и сотрудника.

- Результирующий набор содержит 14 строк, а не 11, и три типа счетов – null.

Три дополнительные строки появляются в результирующем наборе потому, что предыдущая версия запроса включала условие фильтрации p.product_type_cd = 'ACCOUNT'. Этот фильтр исключал строки для счетов типов INSURANCE (страховка) и LOAN (ссуда), например небольшие ссуды коммерческим предприятиям. Поскольку в этой версии запроса нет соединения с таблицей product, нет возможности включить условие фильтрации в основной запрос. Связанный подзапрос к таблице product включает этот фильтр, но единственный производимый им эффект – указание null вместо типа счета. Если хотите избавиться от дополнительных трех строк, можно соединить таблицу product с таблицей account и включить условие фильтрации или просто сделать следующее:

```
mysql> SELECT all_prods.product, all_prods.branch,
    ->   all_prods.name, all_prods.tot_deposits
    -> FROM
    ->  (SELECT
    ->    (SELECT p.name FROM product p
    ->     WHERE p.product_cd = a.product_cd
    ->       AND p.product_type_cd = 'ACCOUNT') product,
    ->    (SELECT b.name FROM branch b
    ->     WHERE b.branch_id = a.open_branch_id) branch,
    ->    (SELECT CONCAT(e.fname, ' ', e.lname) FROM employee e
    ->     WHERE e.emp_id = a.open_emp_id) name,
    ->     SUM(a.avail_balance) tot_deposits
    ->   FROM account a
    ->   GROUP BY a.product_cd, a.open_branch_id, a.open_emp_id) all_prods
    -> WHERE all_prods.product IS NOT NULL;
+-----------------------+--------------+-----------------+--------------+
| product               | branch       | name            | tot_deposits |
+-----------------------+--------------+-----------------+--------------+
| certificate of deposit | Headquarters | Michael Smith   |     11500.00 |
| certificate of deposit | Woburn Branch | Paula Roberts  |      8000.00 |
| checking account      | Headquarters | Michael Smith   |       782.16 |
| checking account      | Woburn Branch | Paula Roberts  |      3315.77 |
| checking account      | Quincy Branch | John Blake     |      1057.75 |
| checking account      | So. NH Branch | Theresa Markham |     67852.33 |
| money market account  | Headquarters | Michael Smith   |     14832.64 |
| money market account  | Quincy Branch | John Blake     |      2212.50 |
| savings account       | Headquarters | Michael Smith   |       767.77 |
| savings account       | Woburn Branch | Paula Roberts  |       700.00 |
| savings account       | So. NH Branch | Theresa Markham |       387.99 |
+-----------------------+--------------+-----------------+--------------+
11 rows in set (0.01 sec)
```

Теперь, после помещения предыдущего запроса в подзапрос (названный all_prods) и добавления условия фильтрации для исключения значений null столбца product, запрос возвращает желаемые 11 строк. В итоге по-

лучаем запрос, где выполняется группировка только необработанных данных таблицы account, а затем результат приукрашивается с помощью данных из трех других таблиц. И все это *без всяких соединений*.

Как отмечалось ранее, скалярные подзапросы тоже могут появляться в блоке order by. Следующий запрос извлекает данные сотрудников, отсортированные по фамилиям начальников сотрудников и затем по фамилиям самих сотрудников:

```
mysql> SELECT emp.emp_id, CONCAT(emp.fname, ' ', emp.lname) emp_name,
    ->  (SELECT CONCAT(boss.fname, ' ', boss.lname)
    ->   FROM employee boss
    ->   WHERE boss.emp_id = emp.superior_emp_id) boss_name
    -> FROM employee emp
    -> WHERE emp.superior_emp_id IS NOT NULL
    -> ORDER BY (SELECT boss.lname FROM employee boss
    ->   WHERE boss.emp_id = emp.superior_emp_id), emp.lname;
+--------+------------------+-----------------+
| emp_id | emp_name         | boss_name       |
+--------+------------------+-----------------+
|     14 | Cindy Mason      | John Blake      |
|     15 | Frank Portman    | John Blake      |
|      9 | Jane Grossman    | Helen Fleming   |
|      8 | Sarah Parker     | Helen Fleming   |
|      7 | Chris Tucker     | Helen Fleming   |
|     13 | John Blake       | Susan Hawthorne |
|      6 | Helen Fleming    | Susan Hawthorne |
|      5 | John Gooding     | Susan Hawthorne |
|     16 | Theresa Markham  | Susan Hawthorne |
|     10 | Paula Roberts    | Susan Hawthorne |
|     17 | Beth Fowler      | Theresa Markham |
|     18 | Rick Tulman      | Theresa Markham |
|     12 | Samantha Jameson | Paula Roberts   |
|     11 | Thomas Ziegler   | Paula Roberts   |
|      2 | Susan Barker     | Michael Smith   |
|      3 | Robert Tyler     | Michael Smith   |
|      4 | Susan Hawthorne  | Robert Tyler    |
+--------+------------------+-----------------+
17 rows in set (0.01 sec)
```

Этот запрос использует два связанных скалярных подзапроса: один в блоке select, извлекающий полное имя руководителя каждого сотрудника, а другой в блоке order by, возвращающий только фамилию руководителя сотрудника для целей сортировки.

Наряду с применением скалярных подзапросов в выражении select можно использовать несвязанные скалярные подзапросы, формирующие значения для выражения insert. Например, предполагается создать новую строку счета. Предоставлены следующие данные:

- Тип счета («savings account»)
- Федеральный ID клиента («555-55-5555»)

- Название отделения, в котором был открыт счет («Quincy Branch»)
- Имя и фамилия операциониста, открывшего счет («Frank Portman»)

Прежде чем можно будет создать строку в таблице account, понадобится найти значения ключей всех этих элементов данных, чтобы заполнить столбцы внешних ключей таблицы account. Сделать это можно двумя способами: выполнить четыре запроса для извлечения значений первичных ключей и поместить эти значения в выражение insert или получить значения четырех ключей с помощью подзапросов внутри выражения insert. Вот пример второго подхода:

```
INSERT INTO account
 (account_id, product_cd, cust_id, open_date, last_activity_date,
  status, open_branch_id, open_emp_id, avail_balance, pending_balance)
VALUES (NULL,
 (SELECT product_cd FROM product WHERE name = 'savings account'),
 (SELECT cust_id FROM customer WHERE fed_id = '555-55-5555'),
  '2005-01-25', '2005-01-25', 'ACTIVE',
 (SELECT branch_id FROM branch WHERE name = 'Quincy Branch'),
 (SELECT emp_id FROM employee WHERE lname = 'Portman' AND fname = 'Frank'),
  0, 0);
```

Единственное SQL-выражение позволяет вам одновременно создать строку в таблице account и найти четыре значения столбцов внешнего ключа. Однако у этого подхода есть один недостаток. Если с помощью подзапросов заполнять столбцы, допускающие значения null, выражение выполнится успешно, даже если один из подзапросов не возвратит значение. Например, если в четвертом подзапросе в имени Frank Portman сделана опечатка, строка в таблице account будет все равно создана, но столбцу open_emp_id будет присвоено значение null.

Краткий обзор подзапросов

В этой главе рассмотрено множество тем, поэтому, пожалуй, не лишне вкратце повторить их. Примеры данной главы продемонстрировали подзапросы, которые:

- Возвращают один столбец и одну строку, один столбец и несколько строк, а также несколько столбцов и строк.
- Не зависят от содержащего выражения (несвязанные подзапросы).
- Используют один или более столбцов из содержащего выражения (связанные подзапросы).
- Применяются в условиях, используемых операторами сравнения и специальными операторами in, not in, exists и not exists.
- Могут находиться в выражениях select, update, delete и insert.
- Формируют таблицы, которые можно соединить в запросе с другими таблицами.

- Позволяют формировать значения для заполнения таблицы или столбцов результирующего набора запроса.
- Используются в блоках `select`, `from`, `where`, `having` и `order by` запросов.

Очевидно, подзапросы – очень многогранный инструмент, поэтому не отчаивайтесь, если при первом прочтении главы вы поймете не все представленные здесь принципы. Продолжайте экспериментировать с подзапросами и вскоре обнаружите, что при написании каждого необычного SQL-выражения очень полезно рассматривать возможность использования подзапроса.

Упражнения

Эти упражнения ориентированы на проверку понимания подзапросов. Решения приведены в приложении C.

9.1

Создайте запрос к таблице `account`, использующий условие фильтрации с несвязанным подзапросом к таблице `product` для поиска всех кредитных счетов (`product.product_type_cd = 'LOAN'`). Должны быть выбраны ID счета, код счета, ID клиента и доступный остаток.

9.2

Переработайте запрос из упражнения 9.1, используя *связанный* подзапрос к таблице `product` для получения того же результата.

9.3

Соедините следующий запрос с таблицей `employee`, чтобы показать уровень квалификации каждого сотрудника:

```
SELECT 'trainee' name, '2004-01-01' start_dt, '2005-12-31' end_dt
UNION ALL
SELECT 'worker' name, '2002-01-01' start_dt, '2003-12-31' end_dt
UNION ALL
SELECT 'mentor' name, '2000-01-01' start_dt, '2001-12-31' end_dt
```

Дайте подзапросу псевдоним `levels` (уровни) и включите ID сотрудника, имя, фамилию и квалификацию (`levels.name`). (Совет: в условии соединения определяйте диапазон, в который попадает столбец `employee.start_date`, с помощью условия неравенства.)

9.4

Создайте запрос к таблице `employee` для получения ID, имени и фамилии сотрудника вместе с названиями отдела и отделения, к которым он приписан. Не используйте соединение таблиц.

10

И снова соединения

На данный момент читатель должен владеть концепцией внутреннего соединения, представленной в главе 5. Основное внимание в этой главе уделено другим способам соединения таблиц, включая внешнее и перекрестное соединения.

Внешние соединения

До сих пор ни в одном из приведенных примеров, включающих запросы к нескольким таблицам, не поднимался вопрос о том, что не все строки таблицы могут соответствовать условиям соединения. Например, при соединении таблицы `account` с таблицей `customer` ничего не было сказано о возможности отсутствия для значения столбца `cust_id` таблицы `account` соответствующего значения в столбце `cust_id` таблицы `customer`. Если бы такое случилось, некоторые строки одной из таблиц не вошли бы в результирующий набор.

На всякий случай давайте проверим данные таблицы. Вот столбцы `account_id` и `cust_id` таблицы `account`:

```
mysql> SELECT account_id, cust_id
    -> FROM account;
+------------+---------+
| account_id | cust_id |
+------------+---------+
|          1 |       1 |
|          2 |       1 |
|          3 |       1 |
|          4 |       2 |
|          5 |       2 |
|          6 |       3 |
|          7 |       3 |
|          8 |       4 |
|          9 |       4 |
```

```
|         10 |        4 |
|         11 |        5 |
|         12 |        6 |
|         13 |        6 |
|         14 |        7 |
|         15 |        8 |
|         16 |        8 |
|         17 |        9 |
|         18 |        9 |
|         19 |        9 |
|         20 |       10 |
|         21 |       10 |
|         22 |       11 |
|         23 |       12 |
|         24 |       13 |
+------------+----------+
24 rows in set (0.04 sec)
```

Имеется 24 счета 13 разных клиентов с ID клиента от 1 до 13, по крайней мере по одному счету на каждого. Вот множество клиентских ID таблицы `customer`:

```
mysql> SELECT cust_id
    -> FROM customer;

+---------+
| cust_id |
+---------+
|       1 |
|       2 |
|       3 |
|       4 |
|       5 |
|       6 |
|       7 |
|       8 |
|       9 |
|      10 |
|      11 |
|      12 |
|      13 |
+---------+
13 rows in set (0.02 sec)
```

В таблице `customer` 13 строк с ID от 1 до 13. Таким образом, каждый ID клиента включен в таблицу `account`, по крайней мере, один раз. Следовательно, при соединении двух таблиц по столбцу `cust_id` можно ожидать, что в результирующий набор будут включены все 24 строки (если нет других условий фильтрации):

```
mysql> SELECT a.account_id, c.cust_id
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id;
```

```
+------------+---------+
| account_id | cust_id |
+------------+---------+
|          1 |       1 |
|          2 |       1 |
|          3 |       1 |
|          4 |       2 |
|          5 |       2 |
|          6 |       3 |
|          7 |       3 |
|          8 |       4 |
|          9 |       4 |
|         10 |       4 |
|         11 |       5 |
|         12 |       6 |
|         13 |       6 |
|         14 |       7 |
|         15 |       8 |
|         16 |       8 |
|         17 |       9 |
|         18 |       9 |
|         19 |       9 |
|         20 |      10 |
|         21 |      10 |
|         22 |      11 |
|         23 |      12 |
|         24 |      13 |
+------------+---------+
24 rows in set (0.00 sec)
```

Как и ожидалось, в результирующем наборе представлены все 24 счета. Но что произойдет, если соединить таблицу `account` с одной из специализированных таблиц клиентов, например таблицей `business`?

```
mysql> SELECT a.account_id, b.cust_id, b.name
    -> FROM account a INNER JOIN business b
    ->   ON a.cust_id = b.cust_id;
+------------+---------+------------------------+
| account_id | cust_id | name                   |
+------------+---------+------------------------+
|         20 |      10 | Chilton Engineering    |
|         21 |      10 | Chilton Engineering    |
|         22 |      11 | Northeast Cooling Inc. |
|         23 |      12 | Superior Auto Body     |
|         24 |      13 | AAA Insurance Inc.     |
+------------+---------+------------------------+
5 rows in set (0.00 sec)
```

Теперь в результирующем наборе только пять строк вместо 24. Заглянем в таблицу `business`, чтобы понять, почему так произошло:

```
mysql> SELECT cust_id, name
    -> FROM business;
```

```
+---------+-----------------------+
| cust_id | name                  |
+---------+-----------------------+
|      10 | Chilton Engineering   |
|      11 | Northeast Cooling Inc. |
|      12 | Superior Auto Body    |
|      13 | AAA Insurance Inc.    |
+---------+-----------------------+
4 rows in set (0.01 sec)
```

Из 13 строк таблицы клиентов только четыре относятся к юридическим лицам. И поскольку у одного из юридических лиц два счета, в общей сложности с юридическими лицами связаны пять строк таблицы `account`.

Но что делать, если требуется, чтобы запрос возвращал *все* счета, но при этом включал название фирмы, только если счет связан с юридическим лицом? Это пример, когда необходимо *внешнее соединение* (*outer join*) таблиц `account` и `business`:

```
mysql> SELECT a.account_id, a.cust_id, b.name
    -> FROM account a LEFT OUTER JOIN business b
    ->   ON a.cust_id = b.cust_id;
+------------+---------+-----------------------+
| account_id | cust_id | name                  |
+------------+---------+-----------------------+
|          1 |       1 | NULL                  |
|          2 |       1 | NULL                  |
|          3 |       1 | NULL                  |
|          4 |       2 | NULL                  |
|          5 |       2 | NULL                  |
|          6 |       3 | NULL                  |
|          7 |       3 | NULL                  |
|          8 |       4 | NULL                  |
|          9 |       4 | NULL                  |
|         10 |       4 | NULL                  |
|         11 |       5 | NULL                  |
|         12 |       6 | NULL                  |
|         13 |       6 | NULL                  |
|         14 |       7 | NULL                  |
|         15 |       8 | NULL                  |
|         16 |       8 | NULL                  |
|         17 |       9 | NULL                  |
|         18 |       9 | NULL                  |
|         19 |       9 | NULL                  |
|         20 |      10 | Chilton Engineering   |
|         21 |      10 | Chilton Engineering   |
|         22 |      11 | Northeast Cooling Inc. |
|         23 |      12 | Superior Auto Body    |
|         24 |      13 | AAA Insurance Inc.    |
+------------+---------+-----------------------+
24 rows in set (0.00 sec)
```

Внешнее соединение включает все строки одной таблицы и вводит данные второй таблицы только в случае обнаружения соответствующих строк. В данном случае в результат вошли все строки таблицы account, поскольку задано left outer join (левостороннее внешнее соединение) и таблица account находится в левой части описания соединения. Столбец name имеет значение null для всех строк, кроме четырех строк клиентов-юридических лиц (cust_id 10, 11, 12 и 13). Вот аналогичный запрос с внешним соединением, но с таблицей individual вместо business:

```
mysql> SELECT a.account_id, a.cust_id, i.fname, i.lname
    -> FROM account a LEFT OUTER JOIN individual i
    ->   ON a.cust_id = i.cust_id;
+------------+---------+----------+---------+
| account_id | cust_id | fname    | lname   |
+------------+---------+----------+---------+
|          1 |       1 | James    | Hadley  |
|          2 |       1 | James    | Hadley  |
|          3 |       1 | James    | Hadley  |
|          4 |       2 | Susan    | Tingley |
|          5 |       2 | Susan    | Tingley |
|          6 |       3 | Frank    | Tucker  |
|          7 |       3 | Frank    | Tucker  |
|          8 |       4 | John     | Hayward |
|          9 |       4 | John     | Hayward |
|         10 |       4 | John     | Hayward |
|         11 |       5 | Charles  | Frasier |
|         12 |       6 | John     | Spencer |
|         13 |       6 | John     | Spencer |
|         14 |       7 | Margaret | Young   |
|         15 |       8 | Louis    | Blake   |
|         16 |       8 | Louis    | Blake   |
|         17 |       9 | Richard  | Farley  |
|         18 |       9 | Richard  | Farley  |
|         19 |       9 | Richard  | Farley  |
|         20 |      10 | NULL     | NULL    |
|         21 |      10 | NULL     | NULL    |
|         22 |      11 | NULL     | NULL    |
|         23 |      12 | NULL     | NULL    |
|         24 |      13 | NULL     | NULL    |
+------------+---------+----------+---------+
24 rows in set (0.00 sec)
```

Этот запрос, по сути, противоположен предыдущему: выводятся имена и фамилии физических лиц, тогда как для юридических лиц эти столбцы имеют значение null.

Сравнение левосторонних и правосторонних внешних соединений

В предыдущем разделе в примерах внешних соединений было задано left outer join. Ключевое слово left свидетельствует о том, что табли-

ца, находящаяся в левой части блока `from`, отвечает за определение числа строк в результирующем наборе, а таблица в правой части предоставляет значения столбцов в случае обнаружения соответствия. Рассмотрим следующий пример:

```
mysql> SELECT c.cust_id, b.name
    -> FROM customer c LEFT OUTER JOIN business b
    ->   ON c.cust_id = b.cust_id;
+---------+------------------------+
| cust_id | name                   |
+---------+------------------------+
|       1 | NULL                   |
|       2 | NULL                   |
|       3 | NULL                   |
|       4 | NULL                   |
|       5 | NULL                   |
|       6 | NULL                   |
|       7 | NULL                   |
|       8 | NULL                   |
|       9 | NULL                   |
|      10 | Chilton Engineering    |
|      11 | Northeast Cooling Inc. |
|      12 | Superior Auto Body     |
|      13 | AAA Insurance Inc.     |
+---------+------------------------+
13 rows in set (0.00 sec)
```

Блок `from` определяет левостороннее внешнее соединение. Таким образом, в результирующий набор входят все 13 строк таблицы `customer`, а таблица `business` поставляет значения во второй столбец результирующего набора для четырех клиентов-юридических лиц. Если выполнить такой же запрос, но указать правостороннее внешнее соединение, будет получен следующий результат:

```
mysql> SELECT c.cust_id, b.name
    -> FROM customer c RIGHT OUTER JOIN business b
    ->   ON c.cust_id = b.cust_id;
+---------+------------------------+
| cust_id | name                   |
+---------+------------------------+
|      10 | Chilton Engineering    |
|      11 | Northeast Cooling Inc. |
|      12 | Superior Auto Body     |
|      13 | AAA Insurance Inc.     |
+---------+------------------------+
4 rows in set (0.00 sec)
```

Теперь число строк результирующего набора определяется количеством строк таблицы `business`. Вот почему в этом множестве всего четыре строки.

Помните, что оба запроса осуществляют внешние соединения. Ключевые слова `left` и `right` просто сообщают оптимизатору БД, какая таблица может иметь пробелы в данных. Если нужно провести внешнее соединение таблиц A и B таким образом, чтобы в результирующий набор входили все строки из A и те строки из B, для которых есть соответствующие данные, можно задать или `A left outer join B`, или `B right outer join A`.

Трехсторонние внешние соединения

В некоторых случаях может потребоваться провести внешнее соединение одной таблицы с двумя другими таблицами. Например, нужен список всех счетов с указанием или имени и фамилии физического лица, или названия фирмы для юридического лица:

```
mysql> SELECT a.account_id, a.product_cd,
    ->   CONCAT(i.fname, ' ', i.lname) person_name,
    ->   b.name business_name
    -> FROM account a LEFT OUTER JOIN individual i
    ->   ON a.cust_id = i.cust_id
    ->   LEFT OUTER JOIN business b
    ->   ON a.cust_id = b.cust_id;
+------------+------------+-----------------+-----------------------+
| account_id | product_cd | person_name     | business_name         |
+------------+------------+-----------------+-----------------------+
|          1 | CHK        | James Hadley    | NULL                  |
|          2 | SAV        | James Hadley    | NULL                  |
|          3 | CD         | James Hadley    | NULL                  |
|          4 | CHK        | Susan Tingley   | NULL                  |
|          5 | SAV        | Susan Tingley   | NULL                  |
|          6 | CHK        | Frank Tucker    | NULL                  |
|          7 | MM         | Frank Tucker    | NULL                  |
|          8 | CHK        | John Hayward    | NULL                  |
|          9 | SAV        | John Hayward    | NULL                  |
|         10 | MM         | John Hayward    | NULL                  |
|         11 | CHK        | Charles Frasier | NULL                  |
|         12 | CHK        | John Spencer    | NULL                  |
|         13 | CD         | John Spencer    | NULL                  |
|         14 | CD         | Margaret Young  | NULL                  |
|         15 | CHK        | Louis Blake     | NULL                  |
|         16 | SAV        | Louis Blake     | NULL                  |
|         17 | CHK        | Richard Farley  | NULL                  |
|         18 | MM         | Richard Farley  | NULL                  |
|         19 | CD         | Richard Farley  | NULL                  |
|         20 | CHK        | NULL            | Chilton Engineering   |
|         21 | BUS        | NULL            | Chilton Engineering   |
|         22 | BUS        | NULL            | Northeast Cooling Inc. |
|         23 | CHK        | NULL            | Superior Auto Body    |
|         24 | SBL        | NULL            | AAA Insurance Inc.    |
+------------+------------+-----------------+-----------------------+
24 rows in set (0.00 sec)
```

Результаты включают все 24 строки таблицы account, а также имена клиентов или названия фирм, поступающие из двух других таблиц в результате внешнего соединения.

Мне не известны ограничения в MySQL, касающиеся количества таблиц, с которыми можно осуществлять внешнее соединение. Но чтобы сократить число соединений в запросе, всегда можно воспользоваться подзапросами. Например, предыдущий пример можно переписать так:

```
mysql> SELECT account_ind.account_id, account_ind.product_cd,
    ->   account_ind.person_name,
    ->   b.name business_name
    -> FROM
    ->  (SELECT a.account_id, a.product_cd, a.cust_id,
    ->     CONCAT(i.fname, ' ', i.lname) person_name
    ->   FROM account a LEFT OUTER JOIN individual i
    ->     ON a.cust_id = i.cust_id) account_ind
    ->   LEFT OUTER JOIN business b
    ->   ON account_ind.cust_id = b.cust_id;
+------------+------------+-----------------+-----------------------+
| account_id | product_cd | person_name     | business_name         |
+------------+------------+-----------------+-----------------------+
|          1 | CHK        | James Hadley    | NULL                  |
|          2 | SAV        | James Hadley    | NULL                  |
|          3 | CD         | James Hadley    | NULL                  |
|          4 | CHK        | Susan Tingley   | NULL                  |
|          5 | SAV        | Susan Tingley   | NULL                  |
|          6 | CHK        | Frank Tucker    | NULL                  |
|          7 | MM         | Frank Tucker    | NULL                  |
|          8 | CHK        | John Hayward    | NULL                  |
|          9 | SAV        | John Hayward    | NULL                  |
|         10 | MM         | John Hayward    | NULL                  |
|         11 | CHK        | Charles Frasier | NULL                  |
|         12 | CHK        | John Spencer    | NULL                  |
|         13 | CD         | John Spencer    | NULL                  |
|         14 | CD         | Margaret Young  | NULL                  |
|         15 | CHK        | Louis Blake     | NULL                  |
|         16 | SAV        | Louis Blake     | NULL                  |
|         17 | CHK        | Richard Farley  | NULL                  |
|         18 | MM         | Richard Farley  | NULL                  |
|         19 | CD         | Richard Farley  | NULL                  |
|         20 | CHK        | NULL            | Chilton Engineering   |
|         21 | BUS        | NULL            | Chilton Engineering   |
|         22 | BUS        | NULL            | Northeast Cooling Inc. |
|         23 | CHK        | NULL            | Superior Auto Body    |
|         24 | SBL        | NULL            | AAA Insurance Inc.    |
+------------+------------+-----------------+-----------------------+
24 rows in set (0.00 sec)
```

В этом варианте запроса внешнее соединение таблицы individual с таблицей account осуществляется в подзапросе account_ind, результаты которого затем путем внешнего соединения соединяются с таблицей bu-

siness. Таким образом, каждый запрос (подзапрос и основной запрос) использует только одно внешнее соединение. При работе с другой БД (не с MySQL) эта стратегия может пригодиться для осуществления внешнего соединения с более чем одной таблицей.

Рекурсивные внешние соединения

В главе 5 была представлена концепция рекурсивного соединения, при котором таблица соединяется сама с собой. Вот пример рекурсивного соединения из главы 5, в котором таблица employee соединяется сама с собой для формирования списка сотрудников и их руководителей:

```
mysql> SELECT e.fname, e.lname,
    ->   e_mgr.fname mgr_fname, e_mgr.lname mgr_lname
    -> FROM employee e INNER JOIN employee e_mgr
    ->   ON e.superior_emp_id = e_mgr.emp_id;
+----------+-----------+-----------+-----------+
| fname    | lname     | mgr_fname | mgr_lname |
+----------+-----------+-----------+-----------+
| Susan    | Barker    | Michael   | Smith     |
| Robert   | Tyler     | Michael   | Smith     |
| Susan    | Hawthorne | Robert    | Tyler     |
| John     | Gooding   | Susan     | Hawthorne |
| Helen    | Fleming   | Susan     | Hawthorne |
| Chris    | Tucker    | Helen     | Fleming   |
| Sarah    | Parker    | Helen     | Fleming   |
| Jane     | Grossman  | Helen     | Fleming   |
| Paula    | Roberts   | Susan     | Hawthorne |
| Thomas   | Ziegler   | Paula     | Roberts   |
| Samantha | Jameson   | Paula     | Roberts   |
| John     | Blake     | Susan     | Hawthorne |
| Cindy    | Mason     | John      | Blake     |
| Frank    | Portman   | John      | Blake     |
| Theresa  | Markham   | Susan     | Hawthorne |
| Beth     | Fowler    | Theresa   | Markham   |
| Rick     | Tulman    | Theresa   | Markham   |
+----------+-----------+-----------+-----------+
17 rows in set (0.02 sec)
```

Этот запрос функционирует нормально за исключением одной маленькой неувязки: в результирующий набор не включаются сотрудники, у которых нет начальника. Однако после замены внутреннего соединения на внешнее в результирующий набор попадают все сотрудники, даже те, у которых нет руководителя:

```
mysql> SELECT e.fname, e.lname,
    ->   e_mgr.fname mgr_fname, e_mgr.lname mgr_lname
    -> FROM employee e LEFT OUTER JOIN employee e_mgr
    ->   ON e.superior_emp_id = e_mgr.emp_id;
+----------+-----------+-----------+-----------+
| fname    | lname     | mgr_fname | mgr_lname |
```

```
+----------+-----------+-----------+-----------+
| Michael  | Smith     | NULL      | NULL      |
| Susan    | Barker    | Michael   | Smith     |
| Robert   | Tyler     | Michael   | Smith     |
| Susan    | Hawthorne | Robert    | Tyler     |
| John     | Gooding   | Susan     | Hawthorne |
| Helen    | Fleming   | Susan     | Hawthorne |
| Chris    | Tucker    | Helen     | Fleming   |
| Sarah    | Parker    | Helen     | Fleming   |
| Jane     | Grossman  | Helen     | Fleming   |
| Paula    | Roberts   | Susan     | Hawthorne |
| Thomas   | Ziegler   | Paula     | Roberts   |
| Samantha | Jameson   | Paula     | Roberts   |
| John     | Blake     | Susan     | Hawthorne |
| Cindy    | Mason     | John      | Blake     |
| Frank    | Portman   | John      | Blake     |
| Theresa  | Markham   | Susan     | Hawthorne |
| Beth     | Fowler    | Theresa   | Markham   |
| Rick     | Tulman    | Theresa   | Markham   |
+----------+-----------+-----------+-----------+
18 rows in set (0.00 sec)
```

Результирующий набор теперь включает Майкла Смита (Michael Smith), который является президентом банка, следовательно, начальника у него нет. Для формирования списка всех сотрудников и их начальников, если таковые имеются, запрос использует левостороннее внешнее соединение. Если сделать внешнее соединение правосторонним, будут получены следующие результаты:

```
mysql> SELECT e.fname, e.lname,
    ->   e_mgr.fname mgr_fname, e_mgr.lname mgr_lname
    -> FROM employee e RIGHT OUTER JOIN employee e_mgr
    ->  ON e.superior_emp_id = e_mgr.emp_id;
+----------+-----------+-----------+-----------+
| fname    | lname     | mgr_fname | mgr_lname |
+----------+-----------+-----------+-----------+
| Susan    | Barker    | Michael   | Smith     |
| Robert   | Tyler     | Michael   | Smith     |
| NULL     | NULL      | Susan     | Barker    |
| Susan    | Hawthorne | Robert    | Tyler     |
| John     | Gooding   | Susan     | Hawthorne |
| Helen    | Fleming   | Susan     | Hawthorne |
| Paula    | Roberts   | Susan     | Hawthorne |
| John     | Blake     | Susan     | Hawthorne |
| Theresa  | Markham   | Susan     | Hawthorne |
| NULL     | NULL      | John      | Gooding   |
| Chris    | Tucker    | Helen     | Fleming   |
| Sarah    | Parker    | Helen     | Fleming   |
| Jane     | Grossman  | Helen     | Fleming   |
| NULL     | NULL      | Chris     | Tucker    |
| NULL     | NULL      | Sarah     | Parker    |
```

```
| NULL     | NULL      | Jane      | Grossman  |
| Thomas   | Ziegler   | Paula     | Roberts   |
| Samantha | Jameson   | Paula     | Roberts   |
| NULL     | NULL      | Thomas    | Ziegler   |
| NULL     | NULL      | Samantha  | Jameson   |
| Cindy    | Mason     | John      | Blake     |
| Frank    | Portman   | John      | Blake     |
| NULL     | NULL      | Cindy     | Mason     |
| NULL     | NULL      | Frank     | Portman   |
| Beth     | Fowler    | Theresa   | Markham   |
| Rick     | Tulman    | Theresa   | Markham   |
| NULL     | NULL      | Beth      | Fowler    |
| NULL     | NULL      | Rick      | Tulman    |
+----------+-----------+-----------+-----------+
28 rows in set (0.00 sec)
```

По этому запросу выбираются все руководители (по-прежнему третий и четвертый столбцы) вместе со всеми их подчиненными. Поэтому Майкл Смит появляется дважды – как начальник Сьюзен Баркер (Susan Barker) и Роберта Тайлера (Robert Tyler). Сьюзен Баркер появляется один раз, она никем не руководит (значения `null` в первом и втором столбцах). Все 18 сотрудников появляются в третьем и четвертом столбцах, по крайней мере, один раз. Некоторые появляются несколько раз, если у них в подчинении несколько сотрудников. Таким образом, в результирующем наборе 28 строк. Этот результат очень отличается от результата предыдущего запроса, а обеспечен он изменением всего одного ключевого слова (`left` на `right`). Следовательно, при использовании внешнего соединения необходимо тщательно продумывать, каким оно должно быть – левосторонним или правосторонним.

Перекрестные соединения

В главе 5 была представлена концепция декартова произведения, которое, в сущности, является результатом соединения нескольких таблиц без указания каких-либо условий соединения. Декартово произведение довольно часто используется в результате случайности (т. е. когда разработчики просто забывают добавить в блок `from` условие соединения), но на самом деле не так уж широко распространено. Однако если *действительно* требуется получить декартово произведение двух таблиц, должно быть задано *перекрестное соединение*:

```
mysql> SELECT pt.name, p.product_cd, p.name
    -> FROM product p CROSS JOIN product_type pt;
+------------------------------+------------+-------------------------+
| name                         | product_cd | name                    |
+------------------------------+------------+-------------------------+
| Customer Accounts            | AUT        | auto loan               |
| Customer Accounts            | BUS        | business line of credit |
| Customer Accounts            | CD         | certificate of deposit  |
```

```
| Customer Accounts             | CHK        | checking account        |
| Customer Accounts             | MM         | money market account    |
| Customer Accounts             | MRT        | home mortgage           |
| Customer Accounts             | SAV        | savings account         |
| Customer Accounts             | SBL        | small business loan     |
| Insurance Offerings           | AUT        | auto loan               |
| Insurance Offerings           | BUS        | business line of credit |
| Insurance Offerings           | CD         | certificate of deposit  |
| Insurance Offerings           | CHK        | checking account        |
| Insurance Offerings           | MM         | money market account    |
| Insurance Offerings           | MRT        | home mortgage           |
| Insurance Offerings           | SAV        | savings account         |
| Insurance Offerings           | SBL        | small business loan     |
| Individual and Business Loans | AUT        | auto loan               |
| Individual and Business Loans | BUS        | business line of credit |
| Individual and Business Loans | CD         | certificate of deposit  |
| Individual and Business Loans | CHK        | checking account        |
| Individual and Business Loans | MM         | money market account    |
| Individual and Business Loans | MRT        | home mortgage           |
| Individual and Business Loans | SAV        | savings account         |
| Individual and Business Loans | SBL        | small business loan     |
+-------------------------------+------------+-------------------------+
24 rows in set (0.00 sec)
```

Этот запрос формирует декартово произведение таблиц `product` и `product_type`. В результате получаем 24 строки (8 строк `product` умножаются на 3 строки `product_type`). Но теперь, когда известно, что такое перекрестное соединение и как оно задается, надо определиться с тем, зачем оно используется. Большинство книг по SQL описывают, что такое перекрестное соединение, и затем говорят, что используется оно редко. Но мне бы хотелось поделиться с читателем ситуациями, в которых я нахожу перекрестное соединение довольно полезным.

В главе 9 обсуждалось, как использовать подзапросы для создания таблиц. Используемый пример показывал, как построить таблицу, включающую три строки, которая могла быть соединена с другими таблицами. Вот таблица из того примера:

```
mysql> SELECT 'Small Fry' name, 0 low_limit, 4999.99 high_limit
    -> UNION ALL
    -> SELECT 'Average Joes' name, 5000 low_limit, 9999.99 high_limit
    -> UNION ALL
    -> SELECT 'Heavy Hitters' name, 10000 low_limit, 9999999.99 high_limit;
+--------------+-----------+------------+
| name         | low_limit | high_limit |
+--------------+-----------+------------+
| Small Fry    |         0 |    4999.99 |
| Average Joes |      5000 |    9999.99 |
| Heavy Hitters |     10000 | 9999999.99 |
+--------------+-----------+------------+
3 rows in set (0.00 sec)
```

Хотя эта таблица является именно тем, что требовалось для разделения клиентов на три группы на основании их совокупного остатка на счете, эта стратегия слияния однострочных таблиц с помощью оператора `union all` не очень подходит, если требуется соорудить большую таблицу.

Например, требуется создать запрос, формирующий строку для каждого дня 2004 года, но в БД нет таблицы, содержащей строки для всех дней. Используя стратегию из примера главы 9, можно было бы сделать что-то вроде этого:

```
SELECT '2004-01-01' dt
UNION ALL
SELECT '2004-01-02' dt
UNION ALL
SELECT '2004-01-03' dt
UNION ALL
...
...
...
SELECT '2004-12-29' dt
UNION ALL
SELECT '2004-12-30' dt
UNION ALL
SELECT '2004-12-31' dt
```

Создавать запрос, соединяющий результаты 366 запросов, немного утомительно, поэтому, наверное, нужна другая стратегия. Что если сгенерировать таблицу с 366 строками (2004 год был високосным) и одним столбцом, содержащим число от 0 до 366, и затем добавлять это число дней к 1 января 2004? Вот одна из возможных методик формирования подобной таблицы:

```
mysql> SELECT ones.num + tens.num + hundreds.num
    -> FROM
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 1 num UNION ALL
    ->   SELECT 2 num UNION ALL
    ->   SELECT 3 num UNION ALL
    ->   SELECT 4 num UNION ALL
    ->   SELECT 5 num UNION ALL
    ->   SELECT 6 num UNION ALL
    ->   SELECT 7 num UNION ALL
    ->   SELECT 8 num UNION ALL
    ->   SELECT 9 num) ones
    ->   CROSS JOIN
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 10 num UNION ALL
    ->   SELECT 20 num UNION ALL
    ->   SELECT 30 num UNION ALL
    ->   SELECT 40 num UNION ALL
    ->   SELECT 50 num UNION ALL
    ->   SELECT 60 num UNION ALL
```

```
    ->   SELECT 70 num UNION ALL
    ->   SELECT 80 num UNION ALL
    ->   SELECT 90 num) tens
    ->   CROSS JOIN
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 100 num UNION ALL
    ->   SELECT 200 num UNION ALL
    ->   SELECT 300 num) hundreds;
+-----------------------------------+
| ones.num + tens.num + hundreds.num |
+-----------------------------------+
|                                 0 |
|                                 1 |
|                                 2 |
|                                 3 |
|                                 4 |
|                                 5 |
|                                 6 |
|                                 7 |
|                                 8 |
|                                 9 |
|                                10 |
|                                11 |
|                                12 |
...
...
...
|                               391 |
|                               392 |
|                               393 |
|                               394 |
|                               395 |
|                               396 |
|                               397 |
|                               398 |
|                               399 |
+-----------------------------------+
400 rows in set (0.00 sec)
```

Если найти декартово произведение трех наборов – {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, {0, 10, 20, 30, 40, 50, 60, 70, 80, 90} и {0, 100, 200, 300} – и сложить значения во всех трех столбцах, получится результат, состоящий из 400 строк, содержащих все числа от 0 до 399. Хотя это больше 366 строк, необходимых для формирования набора дней 2004 года, избавиться от лишних строк достаточно просто. Вскоре я покажу, как это сделать.

Следующий шаг – преобразовать числа в набор дат. Для этого воспользуемся функцией `date_add()` и добавим каждое число в результирующий набор к 1 января 2004 года. Затем введем условие фильтрации, чтобы отбросить все даты 2005 года.

```
mysql> SELECT DATE_ADD('2004-01-01',
    ->   INTERVAL (ones.num + tens.num + hundreds.num) DAY) dt
    -> FROM
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 1 num UNION ALL
    ->   SELECT 2 num UNION ALL
    ->   SELECT 3 num UNION ALL
    ->   SELECT 4 num UNION ALL
    ->   SELECT 5 num UNION ALL
    ->   SELECT 6 num UNION ALL
    ->   SELECT 7 num UNION ALL
    ->   SELECT 8 num UNION ALL
    ->   SELECT 9 num) ones
    ->   CROSS JOIN
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 10 num UNION ALL
    ->   SELECT 20 num UNION ALL
    ->   SELECT 30 num UNION ALL
    ->   SELECT 40 num UNION ALL
    ->   SELECT 50 num UNION ALL
    ->   SELECT 60 num UNION ALL
    ->   SELECT 70 num UNION ALL
    ->   SELECT 80 num UNION ALL
    ->   SELECT 90 num) tens
    ->   CROSS JOIN
    ->  (SELECT 0 num UNION ALL
    ->   SELECT 100 num UNION ALL
    ->   SELECT 200 num UNION ALL
    ->   SELECT 300 num) hundreds
    -> WHERE DATE_ADD('2004-01-01',
    ->   INTERVAL (ones.num + tens.num + hundreds.num) DAY) < '2005-01-01';
+------------+
| dt         |
+------------+
| 2004-01-01 |
| 2004-01-02 |
| 2004-01-03 |
| 2004-01-04 |
| 2004-01-05 |
| 2004-01-06 |
| 2004-01-07 |
| 2004-01-08 |
| 2004-01-09 |
| 2004-01-10 |
...
...
...
| 2004-02-20 |
| 2004-02-21 |
| 2004-02-22 |
| 2004-02-23 |
```

```
| 2004-02-24 |
| 2004-02-25 |
| 2004-02-26 |
| 2004-02-27 |
| 2004-02-28 |
| 2004-02-29 |
| 2004-03-01 |
...
...
...
| 2004-12-20 |
| 2004-12-21 |
| 2004-12-22 |
| 2004-12-23 |
| 2004-12-24 |
| 2004-12-25 |
| 2004-12-26 |
| 2004-12-27 |
| 2004-12-28 |
| 2004-12-29 |
| 2004-12-30 |
| 2004-12-31 |
+------------+
366 rows in set (0.01 sec)
```

В этом подходе замечательно то, что результирующий набор автоматически включает 29 февраля без всяких дополнительных вмешательств, поскольку сервер БД вычисляет его, когда добавляет 59 дней к 1 января 2004 года.

Теперь, когда у нас есть механизм получения всех дней 2004 года, что мы должны с ними сделать? Ну, может потребоваться сгенерировать запрос, по которому будут выводиться все дни 2004 года с номерами счетов, открытых в каждый из дней, с числом осуществленных операций и т. д. Вот пример, дающий ответ на первый вопрос:

```
mysql> SELECT days.dt, COUNT(a.account_id)
    -> FROM account a RIGHT OUTER JOIN
    ->  (SELECT DATE_ADD('2004-01-01',
    ->     INTERVAL (ones.num + tens.num + hundreds.num) DAY) dt
    ->   FROM
    ->    (SELECT 0 num UNION ALL
    ->     SELECT 1 num UNION ALL
    ->     SELECT 2 num UNION ALL
    ->     SELECT 3 num UNION ALL
    ->     SELECT 4 num UNION ALL
    ->     SELECT 5 num UNION ALL
    ->     SELECT 6 num UNION ALL
    ->     SELECT 7 num UNION ALL
    ->     SELECT 8 num UNION ALL
    ->     SELECT 9 num) ones
    ->     CROSS JOIN
```

```
    ->    (SELECT 0 num UNION ALL
    ->     SELECT 10 num UNION ALL
    ->     SELECT 20 num UNION ALL
    ->     SELECT 30 num UNION ALL
    ->     SELECT 40 num UNION ALL
    ->     SELECT 50 num UNION ALL
    ->     SELECT 60 num UNION ALL
    ->     SELECT 70 num UNION ALL
    ->     SELECT 80 num UNION ALL
    ->     SELECT 90 num) tens
    ->     CROSS JOIN
    ->    (SELECT 0 num UNION ALL
    ->     SELECT 100 num UNION ALL
    ->     SELECT 200 num UNION ALL
    ->     SELECT 300 num) hundreds
    ->   WHERE DATE_ADD('2004-01-01',
    ->     INTERVAL (ones.num + tens.num + hundreds.num) DAY) <
    ->       '2005-01-01') days
    ->   ON days.dt = a.open_date
    -> GROUP BY days.dt;
+------------+---------------------+
| dt         | COUNT(a.account_id) |
+------------+---------------------+
| 2004-01-01 |                   0 |
| 2004-01-02 |                   0 |
| 2004-01-03 |                   0 |
| 2004-01-04 |                   0 |
| 2004-01-05 |                   0 |
| 2004-01-06 |                   0 |
| 2004-01-07 |                   0 |
| 2004-01-08 |                   0 |
| 2004-01-09 |                   0 |
| 2004-01-10 |                   0 |
| 2004-01-11 |                   0 |
| 2004-01-12 |                   1 |
| 2004-01-13 |                   0 |
| 2004-01-14 |                   0 |
| 2004-01-15 |                   0 |
...
...
...
| 2004-12-15 |                   0 |
| 2004-12-16 |                   0 |
| 2004-12-17 |                   0 |
| 2004-12-18 |                   0 |
| 2004-12-19 |                   0 |
| 2004-12-20 |                   0 |
| 2004-12-21 |                   0 |
| 2004-12-22 |                   0 |
| 2004-12-23 |                   0 |
| 2004-12-24 |                   0 |
```

```
| 2004-12-25 |                   0 |
| 2004-12-26 |                   0 |
| 2004-12-27 |                   0 |
| 2004-12-28 |                   1 |
| 2004-12-29 |                   0 |
| 2004-12-30 |                   0 |
| 2004-12-31 |                   0 |
+------------+---------------------+
366 rows in set (0.03 sec)
```

Это один из самых интересных запросов, встречавшихся до сих пор в данной книге. Его ценность в том, что он включает перекрестные соединения, внешние соединения, функцию работы с датами, группировку, операции с множествами (`union all`) и агрегатную функцию (`count()`). Это не самое элегантное решение заданной проблемы, но оно послужит примером того, как с небольшой долей творчества и твердым знанием языка программирования даже такой редко используемый механизм, как перекрестные соединения, можно сделать могущественным инструментом в наборе инструментов SQL.

Естественные соединения

Если вы ленивы (а кто не ленив?), можно выбрать тип соединения, при котором сервер БД сам определяет необходимые условия соединения указанных вами таблиц. Известный как *естественное соединение* (*natural join*), этот тип соединения делает предположение о необходимых условиях соединения, полагаясь на идентичные имена столбцов в таблицах. Например, таблица `account` включает столбец `cust_id`, являющийся внешним ключом к таблице `customer`, первичный ключ которой также имеет имя `cust_id`. Таким образом, можно написать запрос, использующий для соединения этих двух таблиц `natural join`:

```
mysql> SELECT a.account_id, a.cust_id, c.cust_type_cd, c.fed_id
    -> FROM account a NATURAL JOIN customer c;
+------------+---------+--------------+-------------+
| account_id | cust_id | cust_type_cd | fed_id      |
+------------+---------+--------------+-------------+
|          1 |       1 | I            | 111-11-1111 |
|          2 |       1 | I            | 111-11-1111 |
|          3 |       1 | I            | 111-11-1111 |
|          4 |       2 | I            | 222-22-2222 |
|          5 |       2 | I            | 222-22-2222 |
|          6 |       3 | I            | 333-33-3333 |
|          7 |       3 | I            | 333-33-3333 |
|          8 |       4 | I            | 444-44-4444 |
|          9 |       4 | I            | 444-44-4444 |
|         10 |       4 | I            | 444-44-4444 |
|         11 |       5 | I            | 555-55-5555 |
|         12 |       6 | I            | 666-66-6666 |
|         13 |       6 | I            | 666-66-6666 |
```

```
|         14 |       7 | I            | 777-77-7777 |
|         15 |       8 | I            | 888-88-8888 |
|         16 |       8 | I            | 888-88-8888 |
|         17 |       9 | I            | 999-99-9999 |
|         18 |       9 | I            | 999-99-9999 |
|         19 |       9 | I            | 999-99-9999 |
|         20 |      10 | B            | 04-1111111  |
|         21 |      10 | B            | 04-1111111  |
|         22 |      11 | B            | 04-2222222  |
|         23 |      12 | B            | 04-3333333  |
|         24 |      13 | B            | 04-4444444  |
+------------+---------+--------------+-------------+
24 rows in set (0.02 sec)
```

Поскольку задано естественное соединение, сервер проверил описания таблиц и добавил для этих двух таблиц условие соединения `a.cust_id = c.cust_id`.

Все это хорошо и замечательно, но что будет, если имена столбцов в таблицах не совпадают? Например, в таблице `account` также есть внешний ключ к таблице `branch`, но этот столбец в таблице `account` назван `open_branch_id`, а не `branch_id`. Посмотрим, что произойдет, если попытаться провести `natural join` между таблицами `account` и `branch`:

```
mysql> SELECT a.account_id, a.cust_id, a.open_branch_id,
    -> FROM account a NATURAL JOIN branch b;
+------------+---------+----------------+---------------+
| account_id | cust_id | open_branch_id | name          |
+------------+---------+----------------+---------------+
|          1 |       1 |              2 | Headquarters  |
|          2 |       1 |              2 | Headquarters  |
|          3 |       1 |              2 | Headquarters  |
|          4 |       2 |              2 | Headquarters  |
|          5 |       2 |              2 | Headquarters  |
|          6 |       3 |              3 | Headquarters  |
|          7 |       3 |              3 | Headquarters  |
|          8 |       4 |              1 | Headquarters  |
|          9 |       4 |              1 | Headquarters  |
|         10 |       4 |              1 | Headquarters  |
|         11 |       5 |              4 | Headquarters  |
|         12 |       6 |              1 | Headquarters  |
|         13 |       6 |              1 | Headquarters  |
|         14 |       7 |              2 | Headquarters  |
|         15 |       8 |              4 | Headquarters  |
|         16 |       8 |              4 | Headquarters  |
|         17 |       9 |              1 | Headquarters  |
|         18 |       9 |              1 | Headquarters  |
|         19 |       9 |              1 | Headquarters  |
|         20 |      10 |              4 | Headquarters  |
|         21 |      10 |              4 | Headquarters  |
|         22 |      11 |              2 | Headquarters  |
|         23 |      12 |              4 | Headquarters  |
```

```
|          24 |       13 |              3 | Headquarters  |
...
...
...
|           1 |        1 |              2 | So. NH Branch |
|           2 |        1 |              2 | So. NH Branch |
|           3 |        1 |              2 | So. NH Branch |
|           4 |        2 |              2 | So. NH Branch |
|           5 |        2 |              2 | So. NH Branch |
|           6 |        3 |              3 | So. NH Branch |
|           7 |        3 |              3 | So. NH Branch |
|           8 |        4 |              1 | So. NH Branch |
|           9 |        4 |              1 | So. NH Branch |
|          10 |        4 |              1 | So. NH Branch |
|          11 |        5 |              4 | So. NH Branch |
|          12 |        6 |              1 | So. NH Branch |
|          13 |        6 |              1 | So. NH Branch |
|          14 |        7 |              2 | So. NH Branch |
|          15 |        8 |              4 | So. NH Branch |
|          16 |        8 |              4 | So. NH Branch |
|          17 |        9 |              1 | So. NH Branch |
|          18 |        9 |              1 | So. NH Branch |
|          19 |        9 |              1 | So. NH Branch |
|          20 |       10 |              4 | So. NH Branch |
|          21 |       10 |              4 | So. NH Branch |
|          22 |       11 |              2 | So. NH Branch |
|          23 |       12 |              4 | So. NH Branch |
|          24 |       13 |              3 | So. NH Branch |
+-------------+----------+----------------+---------------+
96 rows in set (0.03 sec)
```

Кажется, здесь что-то не так; запрос должен возвращать не более 24 строк, поскольку в таблице `account` 24 строки. Произошло следующее: поскольку сервер не смог найти два столбца с одинаковыми именами в этих двух таблицах, условие соединения сформировано не было, и для таблиц было выполнено перекрестное соединение, что в результате дало 96 строк (24 счета умножить на 4 отделения).

Так что стоит ли снижение нагрузки на наши дряхлые пальцы (в виду отсутствия необходимости набирать условие соединения) возникающих при этом неприятностей? Конечно, нет. Следует избегать применения этого типа соединения и использовать внутренние соединения с явными условиями соединения.

Упражнения

Следующие упражнения протестируют понимание внешних и перекрестных соединений. Ответы приведены в приложении C.

10.1

Напишите запрос, возвращающий все типы счетов и открытые счета этих типов (для соединения с таблицей `product` используйте столбец `product_cd` таблицы `account`). Должны быть включены все типы счетов, даже если не был открыт ни один счет определенного типа.

10.2

Переформулируйте запрос из упражнения 10.1 и примените другой тип внешнего соединения (т. е. если в упражнении 10.1 использовалось левостороннее внешнее соединение, используйте правостороннее), так чтобы результаты были, как в упражнении 10.1.

10.3

Проведите внешнее соединение таблицы `account` с таблицами `individual` и `business` (посредством столбца `account.cust_id`) таким образом, чтобы результирующий набор содержал по одной строке для каждого счета. Должны быть включены столбцы `count.account_id`, `account.product_cd`, `individual.fname`, `individual.lname` и `business.name`.

10.4 (дополнительно)

Разработайте запрос, который сформирует набор {1, 2, 3,..., 99, 100}. (Совет: используйте перекрестное соединение как минимум с двумя подзапросами в блоке `from`.)

11

Условная логика

В определенных ситуациях может потребоваться, чтобы SQL-выражения вели себя так или иначе в зависимости от значений определенных столбцов или выражений. Эта глава посвящена написанию выражений, которые могут вести себя по-разному в зависимости от данных, полученных во время выполнения.

Что такое условная логика?

Условная логика – это просто способность выбирать одно из направлений выполнения программы. Например, при запросе информации о клиенте может потребоваться в зависимости от типа клиента извлечь столбцы `fname/lname` таблицы `individual` или столбец `name` таблицы `business`. С помощью внешних соединений можно было бы выбрать обе строки и дать возможность вызывающему определить, какую из них использовать:

```
mysql> SELECT c.cust_id, c.fed_id, c.cust_type_cd,
    ->   CONCAT(i.fname, ' ', i.lname) indiv_name,
    ->   b.name business_name
    -> FROM customer c LEFT OUTER JOIN individual i
    ->   ON c.cust_id = i.cust_id
    ->   LEFT OUTER JOIN business b
    ->   ON c.cust_id = b.cust_id;
+---------+------------+-------------+----------------+----------------------+
| cust_id | fed_id     | cust_type_cd | indiv_name     | business_name        |
+---------+------------+-------------+----------------+----------------------+
|       1 | 111-11-1111 | I           | James Hadley   | NULL                 |
|       2 | 222-22-2222 | I           | Susan Tingley  | NULL                 |
|       3 | 333-33-3333 | I           | Frank Tucker   | NULL                 |
|       4 | 444-44-4444 | I           | John Hayward   | NULL                 |
|       5 | 555-55-5555 | I           | Charles Frasier | NULL                 |
```

```
|       6 | 666-66-6666 | I           | John Spencer    | NULL                  |
|       7 | 777-77-7777 | I           | Margaret Young  | NULL                  |
|       8 | 888-88-8888 | I           | Louis Blake     | NULL                  |
|       9 | 999-99-9999 | I           | Richard Farley  | NULL                  |
|      10 | 04-1111111  | B           | NULL            | Chilton Engineering   |
|      11 | 04-2222222  | B           | NULL            | Northeast Cooling Inc. |
|      12 | 04-3333333  | B           | NULL            | Superior Auto Body    |
|      13 | 04-4444444  | B           | NULL            | AAA Insurance Inc.    |
+--------+------------+-------------+----------------+-----------------------+
13 rows in set (0.13 sec)
```

Вызывающий может взглянуть на значение столбца `cust_type_cd` и выбрать, какой столбец использовать – `indiv_name` или `business_name`. Однако вместо этого можно было бы применить условную логику, воспользовавшись *выражением case*, чтобы определить тип клиента и возвратить соответствующую строку.

```
mysql> SELECT c.cust_id, c.fed_id,
    ->   CASE
    ->     WHEN c.cust_type_cd = 'I'
    ->       THEN CONCAT(i.fname, ' ', i.lname)
    ->     WHEN c.cust_type_cd = 'B'
    ->       THEN b.name
    ->     ELSE 'Unknown'
    ->   END name
    -> FROM customer c LEFT OUTER JOIN individual i
    ->   ON c.cust_id = i.cust_id
    ->   LEFT OUTER JOIN business b
    ->   ON c.cust_id = b.cust_id;
+---------+-------------+-----------------------+
| cust_id | fed_id      | name                  |
+---------+-------------+-----------------------+
|       1 | 111-11-1111 | James Hadley          |
|       2 | 222-22-2222 | Susan Tingley         |
|       3 | 333-33-3333 | Frank Tucker          |
|       4 | 444-44-4444 | John Hayward          |
|       5 | 555-55-5555 | Charles Frasier       |
|       6 | 666-66-6666 | John Spencer          |
|       7 | 777-77-7777 | Margaret Young        |
|       8 | 888-88-8888 | Louis Blake           |
|       9 | 999-99-9999 | Richard Farley        |
|      10 | 04-1111111  | Chilton Engineering   |
|      11 | 04-2222222  | Northeast Cooling Inc. |
|      12 | 04-3333333  | Superior Auto Body    |
|      13 | 04-4444444  | AAA Insurance Inc.    |
+---------+-------------+-----------------------+
13 rows in set (0.00 sec)
```

Эта версия запроса возвращает один столбец `name`. Он формируется *выражением case, начинающимся во второй строке запроса*, которое в дан-

ном случае проверяет значение столбца cust_type_cd и возвращает имя/фамилию физического лица *или* название фирмы.

Выражение case

Все основные серверы БД включают встроенные функции, имитирующие выражение if-then-else, которое есть в большинстве языков программирования (например, функция decode() Oracle, функция if() MySQL и функция coalesce() SQL Server). Выражения case тоже разработаны для поддержки логики if-then-else, но в сравнении со встроенными функциями обладают двумя преимуществами:

- Выражение case является частью стандарта SQL (версия SQL92) и реализовано в Oracle Database, SQL Server и MySQL.
- Выражения case встроены в грамматику SQL и могут быть включены в выражения select, insert, update и delete.

В следующих двух разделах представлены выражения case двух разных типов, а затем я привожу несколько примеров выражений case в действии.

Выражения case с перебором вариантов

Приведенное ранее в этой главе выражение case – пример *выражения case с перебором вариантов* (*searched case expression*), имеющего следующий синтаксис:

```
CASE
  WHEN C1 THEN E1
  WHEN C2 THEN E2
  ...
  WHEN CN THEN EN
  [ELSE ED]
END
```

В этом описании символами C1, C2, ..., CN обозначены условия, а символами E1, E2, ..., EN – выражения, которые должны быть возвращены выражением case. Если условие в блоке when выполняется, выражение case возвращает соответствующее выражение. Кроме того, символ ED представляет применяемое по умолчанию выражение, возвращаемое выражением case, если не выполнено *ни одно* из условий C1, C2, ..., CN (блок else является необязательным, поэтому он заключен в квадратные скобки). Все выражения, возвращаемые различными блоками when, должны обеспечивать результаты одного типа (например, date, number, varchar).

Вот пример выражения case с перебором вариантов:

```
CASE
  WHEN employee.title = 'Head Teller'
    THEN 'Head Teller'
  WHEN employee.title = 'Teller'
```

```
    AND YEAR(employee.start_date) > 2004
    THEN 'Teller Trainee'
  WHEN employee.title = 'Teller'
    AND YEAR(employee.start_date) < 2003
    THEN 'Experienced Teller'
  WHEN employee.title = 'Teller'
    THEN 'Teller'
  ELSE 'Non-Teller'
END
```

Это выражение `case` возвращает строку, с помощью которой можно определять расценки почасовой оплаты, печатать бейджи с именами и т. д. При вычислении выражения `case` блоки `when` обрабатываются сверху вниз. Как только одно из условий блока `when` принимает значение `true`, возвращается соответствующее выражение, а все остальные блоки `when` игнорируются. Если ни одно из условий блока `when` не выполняется, возвращается выражение блока `else`.

Хотя предыдущий пример возвращает строковые выражения, помните, что выражения case могут возвращать выражения любого типа, включая подзапросы. Вот еще одна версия приведенного ранее в этой главе запроса имени физического лица/названия фирмы, в которой для извлечения данных из таблиц `individual` и `business` вместо внешних соединений используются подзапросы:

```
mysql> SELECT c.cust_id, c.fed_id,
    ->   CASE
    ->     WHEN c.cust_type_cd = 'I' THEN
    ->      (SELECT CONCAT(i.fname, ' ', i.lname)
    ->       FROM individual i
    ->       WHERE i.cust_id = c.cust_id)
    ->     WHEN c.cust_type_cd = 'B' THEN
    ->      (SELECT b.name
    ->       FROM business b
    ->       WHERE b.cust_id = c.cust_id)
    ->     ELSE 'Unknown'
    ->   END name
    -> FROM customer c;
+---------+-------------+------------------------+
| cust_id | fed_id      | name                   |
+---------+-------------+------------------------+
|       1 | 111-11-1111 | James Hadley           |
|       2 | 222-22-2222 | Susan Tingley          |
|       3 | 333-33-3333 | Frank Tucker           |
|       4 | 444-44-4444 | John Hayward           |
|       5 | 555-55-5555 | Charles Frasier        |
|       6 | 666-66-6666 | John Spencer           |
|       7 | 777-77-7777 | Margaret Young         |
|       8 | 888-88-8888 | Louis Blake            |
|       9 | 999-99-9999 | Richard Farley         |
|      10 | 04-1111111  | Chilton Engineering    |
```

```
|      11 | 04-2222222  | Northeast Cooling Inc. |
|      12 | 04-3333333  | Superior Auto Body     |
|      13 | 04-4444444  | AAA Insurance Inc.     |
+---------+-------------+------------------------+
13 rows in set (0.01 sec)
```

В этом варианте запроса в блок `from` включена только таблица `customer` и соответствующее имя для каждого клиента получается с помощью связанного подзапроса. Эта версия мне нравится больше той, где применяются внешние соединения, поскольку здесь сервер считывает данные таблицы `individual` и `business` только в случае необходимости, а не соединяет все три таблицы.

Простые выражения case

Простое выражение `case` (*simple case expression*) очень похоже на выражение `case` с перебором вариантов, но несколько менее функционально. Вот его синтаксис:

```
CASE V0
  WHEN V1 THEN E1
  WHEN V2 THEN E2
  ...
  WHEN VN THEN EN
  [ELSE ED]
END
```

В этом описании `V0` представляет значение, а символы `V1`, `V2`, ..., `VN` – значения, сравниваемые с `V0`. Символы `E1`, `E2`, ..., `EN` представляют выражения, возвращаемые выражением case, а `ED` – выражение, которое должно быть возвращено, если ни одно из значений набора `V1`, `V2`, ..., `VN` не соответствует значению `V0`.

Вот пример простого выражения `case`:

```
CASE customer.cust_type_cd
  WHEN 'I' THEN
   (SELECT CONCAT(i.fname, ' ', i.lname)
    FROM individual I
    WHERE i.cust_id = customer.cust_id)
  WHEN 'B' THEN
   (SELECT b.name
    FROM business b
    WHERE b.cust_id = customer.cust_id)
  ELSE 'Unknown Customer Type'
END
```

Простые выражения `case` менее функциональны, чем выражения `case` с перебором вариантов, потому что в них нельзя задать собственные условия; в них просто используются условия равенства. Чтобы понять, что имеется в виду, рассмотрим выражение `case` с перебором вариантов, логика которого аналогична предыдущему простому выражению `case`.

```
CASE
  WHEN customer.cust_type_cd = 'I' THEN
   (SELECT CONCAT(i.fname, ' ', i.lname)
    FROM individual I
    WHERE i.cust_id = customer.cust_id)
  WHEN customer.cust_type_cd = 'B' THEN
   (SELECT b.name
    FROM business b
    WHERE b.cust_id = customer.cust_id)
  ELSE 'Unknown Customer Type'
END
```

Выражения `case` с перебором вариантов позволяют создавать условия вхождения в диапазон, условия неравенства и составные условия, использующие `and`/`or`/`not`, поэтому я бы рекомендовал применять выражения `case` с перебором вариантов во всех случаях, кроме самых простых.

Примеры выражений case

Следующие разделы представляют различные примеры, иллюстрирующие применение условной логики в SQL-выражениях.

Трансформации результирующих наборов

Иногда агрегирование проводится по конечному набору значений, например по дням недели, но требуется, чтобы результирующий набор включал всего одну строку с количеством столбцов, соответствующим количеству значений, а не по одной строке на каждое значение. В качестве примера возьмем запрос, возвращающий число счетов, открытых в каждом году, начиная с 2000 года:

```
mysql> SELECT YEAR(open_date) year, COUNT(*) how_many
    -> FROM account
    -> WHERE open_date > '1999-12-31'
    -> GROUP BY YEAR(open_date);
+------+----------+
| year | how_many |
+------+----------+
| 2000 |        3 |
| 2001 |        4 |
| 2002 |        5 |
| 2003 |        3 |
| 2004 |        9 |
+------+----------+
5 rows in set (0.00 sec)
```

Чтобы трансформировать этот результирующий набор в одну строку с шестью столбцами (по одному для каждого года, с 2000 по 2005), понадобится создать шесть столбцов и в каждом столбце просуммировать *только* строки, относящиеся к данному году:

```
mysql> SELECT
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2000 THEN 1
    ->         ELSE 0
    ->       END) year_2000,
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2001 THEN 1
    ->         ELSE 0
    ->       END) year_2001,
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2002 THEN 1
    ->         ELSE 0
    ->       END) year_2002,
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2003 THEN 1
    ->         ELSE 0
    ->       END) year_2003,
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2004 THEN 1
    ->         ELSE 0
    ->       END) year_2004,
    ->   SUM(CASE
    ->         WHEN EXTRACT(YEAR FROM open_date) = 2005 THEN 1
    ->         ELSE 0
    ->       END) year_2005
    -> FROM account
    -> WHERE open_date > '1999-12-31';
+-----------+-----------+-----------+-----------+-----------+-----------+
| year_2000 | year_2001 | year_2002 | year_2003 | year_2004 | year_2005 |
+-----------+-----------+-----------+-----------+-----------+-----------+
|         3 |         4 |         5 |         3 |         9 |         0 |
+-----------+-----------+-----------+-----------+-----------+-----------+
1 row in set (0.01 sec)
```

Все шесть выражений для столбцов предыдущего запроса идентичны, за исключением значения года. Когда функция `extract()` возвращает нужный год, выражение `case` возвращает значение `1`. В противном случае возвращается `0`. Суммируя все счета, открытые с 2000 года, каждый столбец возвращает число счетов, открытых в соответствующий год. Очевидно, что такие трансформации практически применимы только для небольшого числа значений. Решение задачи по формированию столбцов для каждого года, начиная с 1905-го, быстро стало бы слишком громоздким.

Селективная агрегация

Ранее в главе 9 было показано частичное решение задачи поиска счетов, остаток на которых не соответствует данным таблицы `transaction`. Причиной предоставления частичного решения была необходимость применения условной логики, но теперь у нас есть все для того, чтобы закончить работу. Вот на чем я остановился в главе 9:

```
SELECT CONCAT('ALERT! : Account #', a.account_id,
  ' Has Incorrect Balance!')
FROM account a
WHERE (a.avail_balance, a.pending_balance) <>
 (SELECT SUM(<expression to generate available balance>),
    SUM(<expression to generate pending balance>)
  FROM transaction t
  WHERE t.account_id = a.account_id);
```

Для суммирования отдельных транзакций по данному счету этот запрос использует связанный подзапрос к таблице `transaction`. При суммировании транзакций следует учитывать два факта:

- Суммы транзакций всегда положительны, поэтому чтобы понять, является ли транзакция дебетовой или кредитовой, необходимо посмотреть на ее тип и изменить знак (умножить на -1) для дебетовых транзакций.
- Если дата в столбце `funds_avail_date` больше текущей, транзакция должна быть добавлена в суммарный отложенный остаток, а не в суммарный доступный остаток.

Из доступного остатка некоторые транзакции должны быть исключены, а в отложенный остаток включаются все транзакции, что делает его более простым для вычисления. Вот выражение `case` для вычисления отложенного остатка:

```
CASE
  WHEN transaction.txn_type_cd = 'DBT'
    THEN transaction.amount * -1
  ELSE transaction.amount
END
```

Таким образом, все суммы транзакций для дебетовых транзакций умножаются на -1, а для кредитовых транзакций остаются неизменными. Точно такая же логика применяется и к вычислению доступного остатка, но здесь должны быть включены только те транзакции, которые стали доступными. Поэтому выражение `case` для вычисления доступного остатка включает один дополнительный блок `when`:

```
CASE
  WHEN transaction.funds_avail_date > CURRENT_TIMESTAMP()
    THEN 0
  WHEN transaction.txn_type_cd = 'DBT'
    THEN transaction.amount * -1
  ELSE transaction.amount
END
```

В первом блоке `when` недоступные фонды, такие как неоплаченные чеки, будут добавлять к сумме 0 долларов. Вот окончательный вариант запроса с двумя выражениями `case`:

```
SELECT CONCAT('ALERT! : Account #', a.account_id,
  ' Has Incorrect Balance!')
```

```
FROM account a
WHERE (a.avail_balance, a.pending_balance) <>
 (SELECT
    SUM(CASE
          WHEN t.funds_avail_date > CURRENT_TIMESTAMP()
            THEN 0
          WHEN t.txn_type_cd = 'DBT'
            THEN t.amount * -1
          ELSE t.amount
        END),
    SUM(CASE
          WHEN t.txn_type_cd = 'DBT'
            THEN t.amount * -1
          ELSE t.amount
        END)
  FROM transaction t
  WHERE t.account_id = a.account_id);
```

С помощью условной логики исходные данные поставляются в агрегатные функции `sum()` двумя выражениями `case`, позволяя суммировать соответствующие денежные объемы.

Проверка существования

В некоторых случаях требуется установить существование связи между двумя сущностями, не касаясь количественных показателей. Например, нужно узнать, есть ли у клиента текущие или сберегательные счета, не интересуясь тем, сколько у него счетов каждого типа. Вот запрос с несколькими выражениями `case`, формирующими два столбца выходных данных. Первый показывает, есть ли у клиента текущие счета, а второй – есть ли сберегательные счета:

```
mysql> SELECT c.cust_id, c.fed_id, c.cust_type_cd,
    ->   CASE
    ->     WHEN EXISTS (SELECT 1 FROM account a
    ->       WHERE a.cust_id = c.cust_id
    ->         AND a.product_cd = 'CHK') THEN 'Y'
    ->     ELSE 'N'
    ->   END has_checking,
    ->   CASE
    ->     WHEN EXISTS (SELECT 1 FROM account a
    ->       WHERE a.cust_id = c.cust_id
    ->         AND a.product_cd = 'SAV') THEN 'Y'
    ->     ELSE 'N'
    ->   END has_savings
    -> FROM customer c;
+---------+-------------+--------------+--------------+-------------+
| cust_id | fed_id      | cust_type_cd | has_checking | has_savings |
+---------+-------------+--------------+--------------+-------------+
|       1 | 111-11-1111 | I            | Y            | Y           |
|       2 | 222-22-2222 | I            | Y            | Y           |
```

```
|       3 | 333-33-3333 | I            | Y            | N           |
|       4 | 444-44-4444 | I            | Y            | Y           |
|       5 | 555-55-5555 | I            | Y            | N           |
|       6 | 666-66-6666 | I            | Y            | N           |
|       7 | 777-77-7777 | I            | N            | N           |
|       8 | 888-88-8888 | I            | Y            | Y           |
|       9 | 999-99-9999 | I            | Y            | N           |
|      10 | 04-1111111  | B            | Y            | N           |
|      11 | 04-2222222  | B            | N            | N           |
|      12 | 04-3333333  | B            | Y            | N           |
|      13 | 04-4444444  | B            | N            | N           |
+---------+-------------+--------------+--------------+-------------+
13 rows in set (0.00 sec)
```

Каждое выражение `case` включает связанный подзапрос к таблице `account`: один для поиска текущих счетов, другой – сберегательных счетов. Поскольку каждый блок `when` использует оператор `exists`, условия выполняются, если у клиента есть, по крайней мере, один счет искомого типа.

В других случаях нас может интересовать количество встретившихся строк, но лишь постольку поскольку. Например, следующий запрос с помощью простого выражения `case` подсчитывает количество счетов каждого клиента, а затем возвращает `'None'`, `'1'`, `'2'` или `'3+'`:

```
mysql> SELECT c.cust_id, c.fed_id, c.cust_type_cd,
    ->   CASE (SELECT COUNT(*) FROM account a
    ->       WHERE a.cust_id = c.cust_id)
    ->     WHEN 0 THEN 'None'
    ->     WHEN 1 THEN '1'
    ->     WHEN 2 THEN '2'
    ->     ELSE '3+'
    ->   END num_accounts
    -> FROM customer c;
+---------+-------------+--------------+--------------+
| cust_id | fed_id      | cust_type_cd | num_accounts |
+---------+-------------+--------------+--------------+
|       1 | 111-11-1111 | I            | 3+           |
|       2 | 222-22-2222 | I            | 2            |
|       3 | 333-33-3333 | I            | 2            |
|       4 | 444-44-4444 | I            | 3+           |
|       5 | 555-55-5555 | I            | 1            |
|       6 | 666-66-6666 | I            | 2            |
|       7 | 777-77-7777 | I            | 1            |
|       8 | 888-88-8888 | I            | 2            |
|       9 | 999-99-9999 | I            | 3+           |
|      10 | 04-1111111  | B            | 2            |
|      11 | 04-2222222  | B            | 1            |
|      12 | 04-3333333  | B            | 1            |
|      13 | 04-4444444  | B            | 1            |
+---------+-------------+--------------+--------------+
13 rows in set (0.01 sec)
```

В этом запросе я не хотел проводить различия между клиентами, имеющими более двух счетов, поэтому выражение case просто создает категорию '3+'. Подобный запрос может быть полезным при поиске клиентов, с которыми можно связаться и предложить открыть новый счет в банке.

Ошибки деления на нуль

Проводя вычисления, включающие деление, нужно все время заботиться о том, чтобы знаменатель никогда не был равен нулю. Некоторые серверы БД, такие как Oracle Database, встретив нулевой знаменатель, формируют ошибку, а MySQL просто присваивает результату вычисления значение null, как показывает следующий пример:

```
mysql> SELECT 100 / 0;
+---------+
| 100 / 0 |
+---------+
|    NULL |
+---------+
1 row in set (0.00 sec)
```

Чтобы защитить вычисления от ошибок или, еще хуже, от загадочного получения null, следует ко всем знаменателям применять условную логику, как показано далее:

```
mysql> SELECT a.cust_id, a.product_cd, a.avail_balance /
    ->   CASE
    ->     WHEN prod_tots.tot_balance = 0 THEN 1
    ->     ELSE prod_tots.tot_balance
    ->   END percent_of_total
    -> FROM account a INNER JOIN
    ->  (SELECT a.product_cd, SUM(a.avail_balance) tot_balance
    ->   FROM account a
    ->   GROUP BY a.product_cd) prod_tots
    ->   ON a.product_cd = prod_tots.product_cd;
+---------+------------+------------------+
| cust_id | product_cd | percent_of_total |
+---------+------------+------------------+
|      10 | BUS        |           0.0000 |
|      11 | BUS        |           1.0000 |
|       1 | CD         |           0.1538 |
|       6 | CD         |           0.5128 |
|       7 | CD         |           0.2564 |
|       9 | CD         |           0.0769 |
|       1 | CHK        |           0.0145 |
|       2 | CHK        |           0.0309 |
|       3 | CHK        |           0.0145 |
|       4 | CHK        |           0.0073 |
|       5 | CHK        |           0.0307 |
|       6 | CHK        |           0.0017 |
```

```
|       8 | CHK         |            0.0478 |
|       9 | CHK         |            0.0017 |
|      10 | CHK         |            0.3229 |
|      12 | CHK         |            0.5281 |
|       3 | MM          |            0.1298 |
|       4 | MM          |            0.3219 |
|       9 | MM          |            0.5483 |
|       1 | SAV         |            0.2694 |
|       2 | SAV         |            0.1078 |
|       4 | SAV         |            0.4137 |
|       8 | SAV         |            0.2091 |
|      13 | SBL         |            1.0000 |
+---------+-------------+-------------------+
24 rows in set (0.00 sec)
```

Этот запрос вычисляет отношение остатка на счете к общему остатку для всех счетов одного типа. Поскольку для некоторых типов счетов, таких как ссуды коммерческим предприятиям, общий остаток может равняться нулю, если на текущий момент все ссуды полностью выплачены, лучше всего включить выражение `case`, гарантирующее, что знаменатель никогда не будет равен нулю.

Условные обновления

При обновлении строк таблицы вам иногда придется принимать решения относительно того, какие значения должны быть заданы в определенных столбцах. Например, после вставки новой транзакции должны измениться столбцы `avail_balance`, `pending_balance` и `last_activity_date` таблицы `account`. С обновлением двух последних столбцов проблем нет, но чтобы правильно изменить столбец `avail_balance`, надо проверить столбец `funds_avail_date` таблицы `transaction` и выяснить, сразу ли доступны фонды транзакции. Допустим, только что вставлена транзакция с ID `999`, тогда изменить три столбца таблицы `account` можно с помощью следующего выражения `update`:

```
UPDATE account
  SET last_activity_date = CURRENT_TIMESTAMP(),
  pending_balance = pending_balance +
   (SELECT t.amount *
      CASE t.txn_type_cd WHEN 'DBT' THEN -1 ELSE 1 END
    FROM transaction t
    WHERE t.txn_id = 999),
  avail_balance = avail_balance +
   (SELECT
      CASE
        WHEN t.funds_avail_date > CURRENT_TIMESTAMP() THEN 0
        ELSE t.amount *
          CASE t.txn_type_cd WHEN 'DBT' THEN -1 ELSE 1 END
      END
    FROM transaction t
    WHERE t.txn_id = 999)
```

```
 17  WHERE account.account_id =
 18   (SELECT t.account_id
 19    FROM transaction t
 20    WHERE t.txn_id = 999);
```

Здесь всего три выражения case: два из них (строки 5 и 13) служат для изменения знака суммы транзакции для дебетовых транзакций, а третье выражение case (строка 10) предназначено для проверки даты доступности фондов. Если эта дата еще не наступила, к доступному остатку добавляется нуль; в противном случае добавляется сумма транзакции.

Обработка значений Null

Хотя значения null удобны для хранения в таблицах неизвестных значений столбцов, они не всегда подходят для отображения или использования в выражениях. Например, в окне ввода данных вы, скорее всего, предпочтете отображать слово «unknown», а не оставлять пустое поле. При извлечении данных выражение case позволяет вместо значения null подставлять строку:

```
SELECT emp_id, fname, lname,
  CASE
    WHEN title IS NULL THEN 'Unknown'
    ELSE title
  END
FROM employee;
```

Значения null в вычислениях часто являются причиной результата null, как показывает следующий пример:

```
mysql> SELECT (7 * 5) / ((3 + 14) * null);
+-----------------------------+
| (7 * 5) / ((3 + 14) * null) |
+-----------------------------+
|                        NULL |
+-----------------------------+
1 row in set (0.08 sec)
```

Проводя вычисления, полезно преобразовать значения null в число (обычно 0 или 1) с помощью выражения case, чтобы обеспечить результат вычисления, отличный от null. Например, при вычислении с участием столбца account.avail_balance можно было бы подставить 0 (при сложении или вычитании) или 1 (при умножении или делении) для тех счетов, которые уже открыты, но средства на них еще не помещены:

```
SELECT <some calcuation> +
  CASE
    WHEN avail_balance IS NULL THEN 0
    ELSE avail_balance
  END
  + <rest of calculation>
...
```

Если числовой столбец может содержать значения null, хорошо бы использовать условную логику при любых вычислениях с участием этого столбца, чтобы гарантировать значимые результаты.

Упражнения

Проверьте свою способность применять условную логику с помощью следующих примеров. Выполнив задание, сравните свои решения с ответами, приведенными в приложении C.

11.1

Перепишите следующий запрос, использующий простое выражение case, таким образом, чтобы получить аналогичные результаты с помощью выражения case с перебором вариантов. Попытайтесь свести к минимуму количество блоков when.

```
SELECT emp_id,
  CASE title
    WHEN 'President' THEN 'Management'
    WHEN 'Vice President' THEN 'Management'
    WHEN 'Treasurer' THEN 'Management'
    WHEN 'Loan Manager' THEN 'Management'
    WHEN 'Operations Manager' THEN 'Operations'
    WHEN 'Head Teller' THEN 'Operations'
    WHEN 'Teller' THEN 'Operations'
    ELSE 'Unknown'
  END
FROM employee;
```

11.2

Перепишите следующий запрос так, чтобы результирующий набор содержал всего одну строку и четыре столбца (по одному для каждого отделения). Назовите столбцы branch_1, branch_2 и т. д.

```
mysql> SELECT open_branch_id, COUNT(*)
    -> FROM account
    -> GROUP BY open_branch_id;
+----------------+----------+
| open_branch_id | COUNT(*) |
+----------------+----------+
|              1 |        8 |
|              2 |        7 |
|              3 |        3 |
|              4 |        6 |
+----------------+----------+
4 rows in set (0.00 sec)
```

12

Транзакции

До сих пор все примеры в данной книге были примерами одиночных SQL-выражений. В этой главе рассматриваются требования и среда, необходимые для совместного выполнения нескольких SQL-выражений.

Многопользовательские базы данных

Системы управления базами данных разрешают обращаться к данным и изменять их не только одному пользователю, но и нескольким одновременно. Если каждый пользователь выполняет запросы, как это происходит с хранилищем данных в течение обычных рабочих часов, для сервера БД это не создает больших проблем. Однако если некоторые пользователи добавляют и/или изменяют данные, серверу приходится сохранять довольно много промежуточных результатов.

К примеру, создается отчет, представляющий доступный остаток всех текущих счетов, открытых в отделении. Однако одновременно с выполнением отчета происходит следующее:

- Служащий отделения обрабатывает вклад для одного из клиентов.
- Клиент заканчивает снимать деньги на банкомате в операционном зале.
- Банковское приложение, выполняющееся в конце каждого месяца, начисляет процент по счетам.

Следовательно, пока создается отчет, несколько пользователей изменяют данные. Так, какие цифры должны появиться в отчете? Ответ отчасти зависит от того, как сервер реализовывает *блокировку* (*locking*) – механизм управления одновременным использованием ресурсов данных. Большинство серверов БД применяют одну из двух стратегий блокировки:

- Пользователи, осуществляющие запись в БД, должны запрашивать и получать от сервера *блокировку записи* (*write lock*) для изменения данных. А пользователи, считывающие данные из БД, должны запрашивать и получать от сервера *блокировку чтения* (*read lock*) для осуществления запросов к данным. В то время как чтение может осуществляться одновременно несколькими пользователями, для каждой таблицы (или ее части) одновременно выдается только одна блокировка записи, и запросы на чтение блокируются до тех пор, пока не будет снята блокировка записи.
- Пользователи, осуществляющие запись в БД, для изменения данных должны запрашивать и получать от сервера блокировку записи, но пользователи, считывающие данные, для запроса данных не нуждаются ни в каком типе блокировки. Вместо этого сервер гарантирует, что читатель видит непротиворечивое представление данных (данные представляются неизменными, даже несмотря на то, что другие пользователи могут их модифицировать), начиная с момента начала запроса до его завершения. Этот подход известен как *контроль версий* (*versioning*).

У обеих стратегий есть свои достоинства и недостатки. При первом подходе время ожидания может оказаться длительным, если одновременно поступило много запросов на чтение и запись. Второй подход может создать проблемы в случае длительных запросов, поскольку происходит изменение данных. В данной книге обсуждаются три сервера: Microsoft SQL Server использует первый подход, Oracle Database – второй, а MySQL – оба подхода (в зависимости от выбранного пользователем *механизма хранения* (*storage engine*), который обсуждается немного позже).

Также есть ряд различных стратегий блокировки ресурса. Блокирование может выполняться на одном из трех разных уровней, или с одной из трех *детализаций* (*granularities*):

Блокирование таблицы

Предотвращает одновременное изменение несколькими пользователями данных одной таблицы.

Блокирование страницы

Предотвращает одновременное изменение несколькими пользователями данных одной страницы таблицы (страница – сегмент памяти, обычно от 2 до 16 Кбайт).

Блокирование строки

Предотвращает одновременное изменение несколькими пользователями одной строки таблицы.

У этих подходов тоже есть свои плюсы и минусы. При блокировке всей таблицы возникает очень мало промежуточных результатов, но по мере роста числа пользователей такая блокировка очень быстро приводит

к недопустимым временам ожидания. С другой стороны, в случае блокировки строки сохраняется намного больше промежуточных результатов, но такая блокировка позволяет многим пользователям вносить изменения в одну таблицу, если это касается разных строк. Из трех серверов, обсуждаемых в этой книге, Microsoft SQL Server использует блокировки страницы и строки, Oracle Database – блокировку строки, а MySQL может блокировать таблицу, страницу или строку (опять же в зависимости от выбранного механизма хранения).

Возвращаясь к отчету: данные, появляющиеся на его страницах, будут отражать состояние БД или на момент начала создания отчета (если сервер использует контроль версий), или на момент осуществления сервером блокировки чтения (если сервер использует блокировки и чтения, и записи).

Что такое транзакция?

Если бы серверы БД работали безостановочно, если бы пользователи всегда позволяли программам завершать выполнение и если бы приложения всегда завершались без неустранимых ошибок, прерывающих выполнение, то незачем было бы обсуждать параллельный доступ к базам данных. Однако ни на одну из перечисленных ситуаций рассчитывать нельзя. Следовательно, чтобы несколько пользователей могли осуществлять доступ к одним и тем же данным, необходим еще один элемент.

Этой дополнительной деталью пазла параллелизма является *транзакция* (*transaction*) – механизм группировки нескольких SQL-выражений, позволяющий успешно выполниться *всем* или *ни одному* из них. Если клиент пытается перевести 500 долларов со сберегательного счета на текущий, он немного расстроится, если деньги будут успешно сняты с первого счета, но не внесены на второй. Какой бы ни была причина сбоя (сервер был выключен для проведения работ по техническому обслуживанию, истекло время ожидания запроса на блокировку страницы таблицы `account` и др.), клиент захочет вернуть свои 500 долларов.

Чтобы защититься от ошибок такого рода, программа, обрабатывающая запрос на перевод, сначала начинает транзакцию, затем выполняет SQL-выражения, необходимые для перемещения денег со сберегательного счета на текущий, и, если все проходит успешно, завершает транзакцию, формируя команду `commit` (фиксировать). Однако если происходит что-то непредвиденное, программа выдает команду `rollback` (откат), которая указывает серверу отменить все изменения, внесенные с момента начала транзакции. Весь процесс может выглядеть так:

```
START TRANSACTION;

/* Снять деньги с первого счета, обеспечив достаточный остаток */
UPDATE account SET avail_balance = avail_balance - 500
WHERE account_id = 9988
  AND avail_balance > 500;
```

```
IF <Предыдущим выражением была изменена ровно одна строка> THEN
  /* Внести деньги на следующий счет */
  UPDATE account SET avail_balance = avail_balance + 500
    WHERE account_id = 9989;
  IF <Предыдущим выражением была изменена ровно одна строка> THEN
    /* Все получилось, сделать изменения постоянными */
    COMMIT;
  ELSE
    /* Что-то не так, отменить все изменения, сделанные в данной транзакции */
    ROLLBACK;
  END IF;
ELSE
  /* Недостаток средств на счете или при обновлении возникла ошибка */
  ROLLBACK;
END IF;
```

Хотя предыдущий фрагмент кода может показаться похожим на один из процедурных языков программирования, предоставляемых основными компаниями-производителями БД, такими как PL/SQL от Oracle или Transact SQL от Microsoft, он написан на псевдокоде и не пытается имитировать ни один конкретный язык.

Приведенный выше фрагмент кода начинается с запуска транзакции. После этого делается попытка удалить 500 долларов с текущего счета и затем добавить 500 долларов на сберегательный счет. Если все проходит хорошо, транзакция фиксируется; однако если что-то не так, происходит откат транзакции, т. е. все внесенные с начала транзакции изменения отменяются.

С помощью транзакции программа гарантирует, что пятьсот долларов или останутся на сберегательном счету, или перейдут на текущий счет без какой-либо возможности краха. Независимо от того, была ли транзакция зафиксирована или произошел откат, все ресурсы, занятые (например, блокировка записи) во время выполнения транзакции, по завершении транзакции высвобождаются.

Конечно, если программе удается завершить оба выражения `update`, но сервер выключается до того, как смогут выполниться `commit` или `rollback`, откат транзакции произойдет, когда сервер вернется в рабочий режим. (Одна из задач, которую должен выполнить сервер перед возвращением в нормальный режим работы, – найти все незавершенные транзакции, запущенные на момент выключения сервера, и выполнить их откат.)

Запуск транзакции

Серверы БД обрабатывают создание транзакций одним из двух возможных способов:

- Активная транзакция всегда присутствует для каждого сеанса работы с БД, поэтому нет ни необходимости, ни способа для явного

начала транзакции. По завершении транзакции сервер автоматически начинает новую транзакцию для сеанса пользователя.

- Если транзакция не начата явно, отдельные SQL-выражения фиксируются автоматически независимо друг от друга. Чтобы начать транзакцию, сначала нужно запустить на выполнение команду.

Из трех рассматриваемых серверов первый подход использует Oracle Database, а Microsoft SQL Server и MySQL – второй. Одно из преимуществ подхода Oracle к обработке транзакций в том, что даже в случае одиночной SQL-команды есть возможность сделать откат, если пользователя не удовлетворяет результат или он изменил свое мнение. Таким образом, если вы забудете вставить блок `where` в выражение `delete`, останется возможность отменить неверные действия (разумеется, только если вы осознаете, допив свой утренний кофе, что не хотели удалять все 125 000 строк своей таблицы)). Однако при работе с MySQL и SQL Server, как только нажата клавиша Enter, изменения, осуществленные SQL-выражением, становятся постоянными (и тогда только администратор БД сможет восстановить исходные данные из резервной копии или какими-либо иными средствами).

Стандарт SQL:2003 включает команду `start transaction` (запустить транзакцию), предназначенную для явного начала транзакции. MySQL соответствует этому стандарту, а пользователи SQL Server должны вызывать команду `begin transaction` (начать транзакцию). Для обоих серверов, пока транзакция не начата явно, все операции выполняются в *режиме автоматической фиксации* (*autocommit mode*), т. е. сервер автоматически фиксирует отдельные выражения. Следовательно, можно принять решение об использовании транзакций и выполнить команду запустить/начать транзакцию или просто позволить серверу фиксировать отдельные выражения.

Оба сервера, MySQL и SQL Server, позволяют отключать режим автоматической фиксации для отдельных сеансов. В этом случае серверы будут вести себя в отношении транзакций точно так же, как Oracle Database. В SQL Server для отключения режима автоматической фиксации служит следующая команда:

```
SET IMPLICIT_TRANSACTIONS ON
```

MySQL позволяет отключить режим автоматической фиксации так:

```
SET AUTOCOMMIT=0
```

Если режим автоматической фиксации выключен, все SQL-команды выполняются в рамках транзакции, и их фиксацию или откат следует выполнять явно.

Рекомендация: при каждом входе в систему следует отключать режим автоматической фиксации. Выполнение всех SQL-выражений в рамках транзакции должно войти в привычку. По край-

ней мере, это поможет вам избежать унизительной необходимости просить администратора БД восстановить уничтоженные по неосторожности данные.

Завершение транзакции

Если транзакция запущена – явно посредством команды `start transaction` или неявно сервером БД, – пользователь должен явно завершить ее, чтобы внесенные им изменения стали постоянными. Это делается с помощью команды `commit`, которая указывает серверу пометить изменения как постоянные и высвободить все ресурсы (т. е. снять блокировку страниц или строк), используемые во время транзакции.

Если решено отменить все изменения, сделанные с момента начала транзакции, необходимо выполнить команду `rollback`, которая указывает серверу вернуть данные в то состояние, в каком они находились до начала транзакции. После завершения выполнения `rollback` все ресурсы, используемые сеансом, высвобождаются.

Кроме выполнения команды `commit` или `rollback`, возможны еще несколько сценариев завершения транзакции – или как косвенный результат действий пользователя, или как результат чего-то, находящегося вне власти пользователя:

- Выключение сервера; в этом случае откат транзакции будет выполнен автоматически при возобновлении работы сервера.
- Выполнение SQL-выражения управления схемой, например `alter table`, что приведет к фиксации текущей транзакции и запуску новой.
- Выполнение еще одной команды `start transaction`, в результате чего происходит фиксация предыдущей транзакции.
- Преждевременное завершение транзакции сервером, который выявил *взаимоблокировку* (*deadlock*) и решил, что виновна в этом данная транзакция. В этом случае будет выполнен откат транзакции и пользователь получит сообщение об ошибке.

Из этих четырех сценариев первый и третий довольно просты, а вот два других заслуживают некоторого внимания. Если говорить о втором сценарии, изменения базы данных, независимо от того, было ли это добавление новой таблицы, индексация или удаление столбца из таблицы, не могут быть отменены. Таким образом, команды, изменяющие схему, должны выполняться вне транзакции. Поэтому если транзакция уже запущена, сервер зафиксирует ее, выполнит команду(-ы) SQL-выражений управления схемой и затем автоматически запустит новую транзакцию для данного сеанса. Сервер не будет информировать пользователя о происходящем, поэтому следует действовать аккуратно, чтобы выражения, составляющие единицу работы, невзначай не были разбросаны сервером по нескольким транзакциям.

Четвертый сценарий занимается выявлением взаимоблокировок. Взаимоблокировка происходит, когда две разные транзакции ожидают ресурсов, удерживаемых другой транзакцией. Например, транзакция A только что обновила таблицу `account` и ожидает блокировки записи для таблицы `transaction`. В это время транзакция B вставила строку в таблицу `transaction` и ожидает блокировки записи для таблицы `account`. Если случится, что обе транзакции изменяют одну и ту же страницу или строку (в зависимости от детализации блокировок, используемой сервером БД), каждая из них будет бесконечно ожидать, когда другая транзакция завершит выполнение и высвободит необходимый ресурс. Серверы БД всегда должны быть настороже и выявлять такие ситуации, чтобы не остановиться полностью; при выявлении взаимоблокировки выбирается одна из транзакций (произвольно или по какому-то критерию) и осуществляется ее откат, чтобы дать возможность другой транзакции продолжить выполнение.

В отличие от обсуждавшегося ранее второго сценария, сервер БД сформирует ошибку и сообщит пользователю о том, что был сделан откат его транзакции из-за выявления взаимоблокировки. Например, при работе с MySQL будет получена ошибка #1213, сопровождаемая следующим сообщением:

```
Message: Deadlock found when trying to get lock; try restarting transaction
```
(Сообщение: Обнаружена взаимоблокировка при попытке блокировки; попытайтесь перезапустить транзакцию)

Как предлагает сообщение об ошибке, разумным будет повторно запустить транзакцию, для которой был сделан откат из-за выявления взаимоблокировки. Однако если взаимоблокировки становятся довольно частым явлением, вероятно, необходимо скорректировать приложения, осуществляющие доступ к БД, чтобы снизить вероятность взаимоблокировок (одна общепринятая стратегия – обеспечить, чтобы доступ к ресурсам всегда осуществлялся в одном и том же порядке, например изменение данных счета выполнялось бы перед вставкой данных транзакции).

Точки сохранения транзакций

В некоторых случаях может возникнуть проблема, когда требуется откат транзакции, но не хочется отменять *все*, что было сделано в рамках этой транзакции. Для таких ситуаций в транзакции можно установить одну или более *точек сохранения* (*savepoints*) и использовать их для отката к определенному месту транзакции, а не откатывать полностью к началу.

Всем точкам сохранения должны быть присвоены имена, что позволит иметь несколько таких точек в одной транзакции. Создать точку сохранения `my_savepoint` можно так:

```
SAVEPOINT my_savepoint;
```

Чтобы сделать откат к определенной точке сохранения, просто выполняется команда `rollback`, за которой следуют ключевые слова `to savepoint` (к точке сохранения) и имя точки сохранения:

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

Выбор механизма хранения

Для низкоуровневых операций с БД, таких как извлечение из таблицы конкретной строки по значению первичного ключа, Oracle Database и Microsoft SQL Server используют всего один механизм хранения. А сервер MySQL спроектирован так, что для обеспечения низкоуровневой функциональности БД, включая блокировку ресурсов и управление транзакциями, могут использоваться несколько механизмов хранения. MySQL версии 4.1 поддерживает следующие механизмы хранения:

MyISAM

Нетранзакционный механизм, использующий блокировки таблицы.

MEMORY

Нетранзакционный механизм, применяемый для таблиц в оперативной памяти.

BDB

Транзакционный механизм, использующий блокировку на уровне страницы.

InnoDB

Транзакционный механизм, использующий блокировку на уровне строки.

Merge

Специальный механизм, предназначенный для создания нескольких идентичных таблиц MyISAM, создающих при этом впечатление одной таблицы (также называется сегментированием таблиц).

NDB

Специальный механизм, предназначенный для распределения одной БД по нескольким компьютерам (также называется кластеризацией).

Archive

Специальный механизм, предназначенный для хранения больших объемов неиндексированных данных, преимущественно для архивных целей.

Не надо думать, что MySQL вынуждает выбирать для БД единственный механизм хранения. Этот сервер достаточно гибок и обеспечивает возможность применять для каждой таблицы собственный механизм хранения. Однако для всех таблиц, которые могут принимать участие в транзакциях, следовало бы использовать механизм хранения InnoDB, применяющий блокировку на уровне строки и контроль версий, обеспечивая тем самым наиболее высокий уровень согласованности из всех механизмов хранения.

Механизм хранения можно задавать явно при создании таблицы или изменять его для уже существующей таблицы. Если механизм хранения таблицы не известен, можно воспользоваться командой `show table`, например:

```
mysql> SHOW TABLE STATUS LIKE 'transaction' \G
********************* 1. row *********************
           Name: transaction
         Engine: InnoDB
...
 Create_options:
        Comment: InnoDB free: 3072 kB; ...
```

Взглянув на второй элемент, можно понять, что таблица `transaction` уже использует механизм InnoDB. Если бы это было не так, можно было бы назначить механизм InnoDB для этой таблицы посредством следующей команды:

```
ALTER TABLE transaction ENGINE = INNODB;
```

Вот пример использования точек сохранения:

```
START TRANSACTION;

UPDATE product
SET date_retired = CURRENT_TIMESTAMP( )
WHERE product_cd = 'XYZ';

SAVEPOINT before_close_accounts;

UPDATE account
SET status = 'CLOSED', close_date = CURRENT_TIMESTAMP( ),
  last_activity_date = CURRENT_TIMESTAMP( )
WHERE product_cd = 'XYZ';

ROLLBACK TO SAVEPOINT before_close_accounts;

COMMIT;
```

Результирующий эффект этой транзакции состоит в том, что фиктивный тип счета `XYZ` выходит из обращения, но ни один из счетов не закрывается.

При использовании точек сохранения необходимо помнить следующее:

- Несмотря на название, при создании точки сохранения ничего не сохраняется. Если требуется сделать транзакцию постоянной, необходимо выполнить команду `commit`.
- Если использовать команду `rollback` без указания точки сохранения, все точки сохранения транзакции будут проигнорированы и отменена будет вся транзакция.

Работая с SQL Server, используйте его собственные команды: `save transaction` (сохранить транзакцию) – для создания точки сохранения и `rollback transaction` (откатить транзакцию) – для отката к точке сохранения. За каждой такой командой должно следовать имя точки сохранения.

13

Индексы и ограничения

Поскольку данная книга посвящена методикам программирования, предыдущие двенадцать глав рассматривали элементы языка SQL, позволяющие создавать мощные выражения `select`, `insert`, `update` и `delete`. Однако базы данных обладают и другими средствами, *косвенно* влияющими на создаваемый код. В этой главе рассмотрены два таких средства: индексы и ограничения.

Индексы

При вставке строки в таблицу сервер БД не пытается поместить данные в какое-то определенное место таблицы. Например, добавляя строку в таблицу `department`, сервер размещает ее не по порядку номеров столбца `dept_id` и не в алфавитном порядке по столбцу `name`. Вместо этого сервер просто помещает данные в следующую доступную ячейку памяти в файле (сервер хранит список свободной памяти для каждой таблицы). Поэтому, чтобы ответить на запрос к таблице `department`, серверу приходится проверять каждую ее строку. Например, выполняется следующий запрос:

```
mysql> SELECT dept_id, name
    -> FROM department
    -> WHERE name LIKE 'A%';
+---------+----------------+
| dept_id | name           |
+---------+----------------+
|       3 | Administration |
+---------+----------------+
1 row in set (0.03 sec)
```

Чтобы найти все отделы, названия которых начинаются на `'A'`, сервер должен просмотреть каждую строку таблицы `department` и проверить

содержимое столбца name. Если имя отдела начинается с 'A', строка добавляется в результирующий набор.

Для таблицы, состоящей только из трех строк, этот метод хорош. А представьте, сколько времени потребуется, чтобы ответить на этот запрос, если в таблице 3 000 000 строк? При некотором числе строк (больше трех и меньше 3 000 000) сервер достигает того предела, когда уже не успевает ответить на запрос в течение приемлемого промежутка времени без дополнительной помощи. Эта помощь приходит в форме одного или нескольких *индексов* таблицы department.

Даже если вы ни разу не слышали об индексе БД, вы, безусловно, знаете, что такое индекс (в этой книге он тоже есть). Индекс – это просто механизм поиска определенного элемента ресурса. Например, в конце каждого технического издания есть предметный указатель, позволяющий найти определенное слово или фразу. В указателе эти слова и фразы перечислены в алфавитном порядке, что позволяет читателю быстро перейти к определенной букве, выбрать нужную запись и затем найти страницу или страницы, где можно отыскать это слово или фразу.

Как человек использует предметный указатель, чтобы найти слова в печатном издании, так и сервер БД с помощью индексов выявляет местоположение строк в таблице. Индексы – это специальные таблицы, содержимое которых, в отличие от обычных таблиц данных, *хранится* в определенном порядке. Однако индекс включает не *все* данные сущности, а только столбец (или столбцы), используемый для определения местоположения строк в таблице данных, а также информацию, описывающую физическое размещение строк. Поэтому предназначение индексов – помочь в извлечении подмножества строк и столбцов таблицы *без* необходимости проверять все строки.

Создание индекса

Вернемся к таблице department. Допустим, принято решение добавить индекс для столбца name, чтобы ускорить выполнение запросов по полному или частичному имени отдела, а также операций update или delete, использующих это имя. Вот как можно добавить такой индекс в базу данных MySQL:

```
mysql> ALTER TABLE department
    -> ADD INDEX dept_name_idx (name);
Query OK, 3 rows affected (0.08 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

Это выражение создает индекс (точнее, индекс на основе B-дерева, но более подробно об этом позже) для столбца department.name. Более того, индексу присвоено имя dept_name_idx. При наличии индекса оптимизатор запросов (обсуждаемый в главе 3) воспользуется им, если сочтет это выгодным (например, если в таблице department только три строки, оптимизатор прекрасно справится и без использования индекса и про-

смотрит всю таблицу). Если для таблицы есть несколько индексов, оптимизатору придется выбрать индекс, лучше всего подходящий для конкретного SQL-выражения.

MySQL рассматривает индексы как необязательные компоненты таблицы, вот почему для добавления или удаления индекса используется команда `alter table` (видоизменить таблицу). Другие серверы БД, включая SQL Server и Oracle Database, считают индексы независимыми объектами схемы. Поэтому для SQL Server и Oracle индекс формировался бы с помощью команды `create index` (создать индекс):

```
CREATE INDEX dept_name_idx
ON department (name);
```

Все серверы БД позволяют просматривать доступные индексы. Увидеть все индексы определенной таблицы пользователи MySQL могут с помощью команды `show` (показать):

```
mysql> SHOW INDEX FROM department \G
*************************** 1. row ***************************
       Table: department
  Non_unique: 0
    Key_name: PRIMARY
Seq_in_index: 1
 Column_name: dept_id
   Collation: A
 Cardinality: 3
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
*************************** 2. row ***************************
       Table: department
  Non_unique: 0
    Key_name: dept_name_uidx
Seq_in_index: 1
 Column_name: name
   Collation: A
 Cardinality: 3
    Sub_part: NULL
      Packed: NULL
        Null:
  Index_type: BTREE
     Comment:
2 rows in set (0.02 sec)
```

Из результата видно, что для таблицы `department` есть два индекса: `PRIMARY` – для столбца `dept_id` и `dept_name_idx` – для столбца `name`. Поскольку мы создавали только один индекс (`dept_name_idx`), может возник-

нуть вопрос, откуда взялся второй. При создании таблицы department выражение create table включало ограничение, назначающее столбец dept_id первичным ключом таблицы. Вот выражение для создания таблицы:

```
CREATE TABLE department
 (dept_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  name VARCHAR(20) NOT NULL,
  CONSTRAINT pk_department PRIMARY KEY (dept_id)
 );
```

Когда таблица была создана, сервер MySQL автоматически сформировал индекс для столбца первичного ключа, которым в данном случае является dept_id, и назвал индекс PRIMARY. Ограничения будут рассмотрены в этой главе позже.

Если после создания индекса выясняется, что он не оправдывает себя, его можно удалить следующим образом:

```
mysql> ALTER TABLE department
    -> DROP INDEX dept_name_idx;
Query OK, 3 rows affected (0.02 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

Пользователи SQL Server и Oracle Database для уничтожения индекса должны использовать команду drop index:

```
DROP INDEX dept_name_idx;
```

Уникальные индексы

При проектировании БД важно определить, какие столбцы могут содержать дублирующие данные, а какие нет. Например, в таблице individual может быть два клиента по имени Джон Смит (John Smith), поскольку у каждой строки будут свои идентификатор (cust_id), дата рождения и идентификационный номер (customer.fed_id), помогающие различать их. Однако вряд ли допустимо, чтобы в таблице department было два отдела под одним и тем же названием. Можно установить правило, запрещающее дублирование имен отделов, создав для столбца department.name *уникальный индекс* (*unique index*).

Уникальный индекс выполняет несколько функций, поскольку помимо обеспечения всех преимуществ обычного индекса он также служит механизмом запрета дублирования значений в индексируемом столбце. При любой вставке строки или изменении индексированного столбца сервер БД проверяет уникальный индекс, чтобы увидеть, нет ли такого значения в другой строке таблицы. Вот как создавался бы уникальный индекс для столбца department.name:

```
mysql> ALTER TABLE department
    -> ADD UNIQUE dept_name_idx (name);
```

```
Query OK, 3 rows affected (0.04 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

В SQL Server и Oracle Database при создании индекса нужно только добавить ключевое слово `unique`:

```
CREATE UNIQUE INDEX dept_name_idx
ON department (name);
```

При наличии такого индекса в случае попытки добавить еще один отдел под названием `'Operations'` будет получена ошибка:

```
mysql> INSERT INTO department (dept_id, name)
    -> VALUES (999, 'Operations');
ERROR 1062 (23000): Duplicate entry 'Operations' for key 2
```

Нет необходимости создавать уникальные индексы для столбца(-ов) первичного ключа, поскольку сервер уже проверяет уникальность значений первичных ключей. Однако при необходимости для одной таблицы можно создать несколько уникальных индексов.

Составные индексы

Кроме уже представленных индексов по одному столбцу, можно создавать индексы, охватывающие несколько столбцов. Если, например, требуется проводить поиск сотрудников по имени и фамилии, можно сделать индекс сразу для *двух* столбцов:

```
mysql> ALTER TABLE employee
    -> ADD INDEX emp_names_idx (lname, fname);
Query OK, 18 rows affected (0.10 sec)
Records: 18  Duplicates: 0  Warnings: 0
```

Этот индекс будет полезен для запросов, использующих имя и фамилию или только фамилию, но не подходит для запросов, в которых задано только имя сотрудника. Чтобы понять почему, рассмотрим, как проводился бы поиск телефонного номера. Чтобы быстро найти чей-то номер телефона, если известны имя и фамилия, можно воспользоваться телефонной книгой, поскольку она организована по фамилии, а потом по имени. Если известно только имя человека, придется просматривать все записи телефонной книги и выбирать каждую запись с указанным именем.

Поэтому при создании составных индексов (multiple-column indexes) необходимо тщательно продумать, какой столбец указывать первым, а какой вторым и т. д., чтобы индекс был максимально полезным. Однако следует помнить, что если требуется обеспечить адекватное время ответа, ничто не мешает создать несколько индексов, используя тот же набор столбцов, но в другом порядке.

Типы индексов

Индексация – мощный инструмент, но из-за большого разнообразия типов данных единственная стратегия индексации не всегда является оптимальной. Следующие разделы иллюстрируют разные типы индексации, доступные в различных серверах.

Индексы на основе B-дерева

Все приведенные до сих пор индексы – это *индексы на основе сбалансированного дерева* (*balanced-tree indexes*), чаще называемые *индексами на основе B-дерева* (*B-tree indexes*). MySQL, Oracle Database и SQL Server используют такие индексы по умолчанию, поэтому если явно не запросить другой тип индекса, вы всегда получите этот индекс. Как и следовало ожидать, индексы на основе B-дерева организованы как деревья с одним или более уровнями *узлов* (*branch nodes*), приводящими к единственному уровню *листьев* (*leaf nodes*). Узлы используются для навигации по дереву, тогда как на листьях располагаются фактические значения и информация о местоположении. Например, индекс на основе B-дерева, созданный для столбца `employee.lname`, мог бы выглядеть примерно так, как показано на рис. 13.1.

Если бы был сделан запрос для выбора всех сотрудников, фамилии которых начинаются на `'G'`, сервер нашел бы верхний узел – *корневой узел* (*root node*) – и проследовал бы по связи к узлу, отвечающему за фамилии, начинающиеся с букв от `'A'` до `'M'`. Этот узел, в свою очередь, направил бы сервер к листу, содержащему фамилии, начинающиеся с букв от `'G'` до `'I'`. Затем сервер считывал бы значения листа до тех пор, пока не встретил бы значение, начинающееся не на `'G'` (которым в данном случае является `'Hawthorne'`).

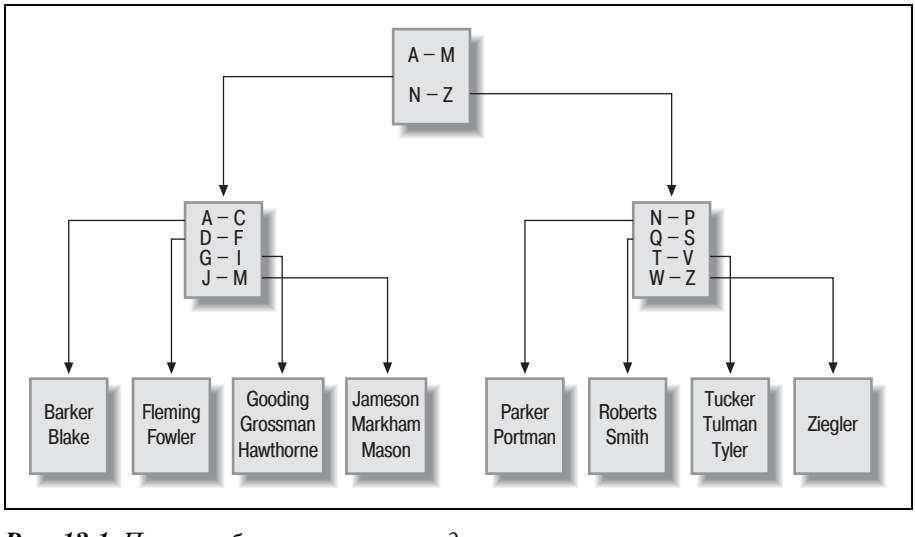


Рис. 13.1. *Пример сбалансированного дерева*

При вставке, обновлении и удалении данных таблицы `employee` сервер будет стараться сохранять сбалансированность дерева, чтобы количество узлов/листьев с одной стороны корневого узла не сильно превышало количество узлов с другой стороны. Сервер может добавлять или удалять узлы, чтобы более равномерно перераспределять значения. Он даже может добавить или удалить целый уровень узлов. Поддерживая дерево сбалансированным, сервер может быстро перемещаться к листьям и находить нужные значения без навигации по множеству уровней узлов.

Битовые индексы

Индексы на основе B-дерева замечательно подходят для обработки столбцов, содержащих много разных значений, таких как имена/фамилии клиентов, но они могут стать громоздкими для столбца с небольшим количеством значений. Например, принято решение сформировать индекс для столбца `account.product_cd`, чтобы обеспечить быстрый выбор всех счетов определенного типа (например, текущих, сберегательных). Однако есть всего восемь разных типов счетов, и некоторые из них встречаются гораздо чаще остальных. Поэтому по мере роста количества счетов могут возникнуть сложности с обеспечением сбалансированности индекса на основе B-дерева.

Для столбцов, содержащих небольшое количество значений при большом числе строк (это известно как данные *с малым кардинальным числом* (*low-cardinality*)), необходима другая стратегия индексации. Чтобы обработать эту ситуацию с большей эффективностью, Oracle Database включает *битовые индексы* (*bitmap indexes*), которые формируют битовый образ каждого значения, хранящегося в столбце. На рис. 13.2 показано, как может выглядеть битовый индекс для данных столбца `account.product_cd`.

Этот индекс содержит шесть битовых карт, по одной для каждого значения столбца `product_cd` (два из восьми доступных типов счетов не используются). Каждая битовая карта включает значение 0/1 для каж-

Value/row	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
BUS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0
CD	0	0	1	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0	0	0	0
CHK	1	0	0	1	0	1	0	1	0	0	1	1	0	0	1	0	1	0	0	1	0	0	1	0
MM	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	1	0	0	0	0	0	0
SAV	0	1	0	0	1	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
SBL	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

Рис. 13.2. *Пример битового индекса*

дой из 24 строк таблицы account. Таким образом, если серверу делается запрос на извлечение всех депозитных счетов денежного рынка (product_cd = 'MM'), он просто находит все значения 1 в битовой карте MM и возвращает строки 7, 10 и 18. Если требуется найти несколько значений, сервер также может комбинировать битовые карты. Например, если пользователь хочет получить все депозитные счета денежного рынка *и* сберегательные счета (product_cd = 'MM' или product_cd = 'SAV'), сервер может осуществить для битовых карт MM и SAV операцию OR (ИЛИ) и возвратит строки 2, 5, 7, 9, 10, 16 и 18.

Битовые индексы – милое компактное решение по индексации данных с малым кардинальным числом. Однако эта стратегия не годится, если число хранящихся в столбце значений слишком велико по сравнению с числом строк (в таком случае говорят о данных с *большим кардинальным числом* (*high-cardinality*)), потому что серверу пришлось бы обслуживать слишком много битовых карт. Например, не следует создавать битовый индекс для столбца первичного ключа, поскольку он является примером максимально возможного количества элементов (новое значение для каждой строки).

Пользователи Oracle могут формировать битовые индексы, просто добавляя ключевое слово bitmap в выражение create index:

```
CREATE BITMAP INDEX acc_prod_idx ON account (product_cd);
```

Битовые индексы широко используются в информационных хранилищах, где обычно индексируются большие объемы данных для столбцов, содержащих относительно небольшое количество значений (например, квартальные отчеты, географические регионы, продукты, продавцы).

Текстовые индексы

Если БД используется для хранения документов, может потребоваться обеспечить пользователям возможность выполнять поиск слов или фраз в документах. Конечно, не хочется, чтобы сервер открывал каждый документ и просматривал его в поисках нужного текста при каждом запросе. Традиционные стратегии индексации в данном случае не годятся. Чтобы справиться с этой ситуацией, MySQL и Oracle Database включают для документов специальные механизмы индексации и поиска. В SQL Server и MySQL есть так называемые индексы *по всему тексту* (*full-text*) (для MySQL индексы по всему тексту доступны только с механизмом хранения MyISAM). Oracle Database включает мощный инструментарий *Oracle Text*. Поиск по документам достаточно специализирован, поэтому не будем приводить здесь пример, но я хотел хотя бы сообщить о такой возможности.

Использование индексов

Обычно сервер использует индексы для быстрого обнаружения местоположения интересующих строк в конкретной таблице. После этого

сервер просматривает ассоциированную таблицу и извлекает дополнительную информацию, запрашиваемую пользователем. Рассмотрим следующий запрос:

```
mysql> SELECT emp_id, fname, lname
    -> FROM employee
    -> WHERE emp_id IN (1, 3, 9, 15, 22);
+--------+---------+----------+
| emp_id | fname   | lname    |
+--------+---------+----------+
|      1 | Michael | Smith    |
|      3 | Robert  | Tyler    |
|      9 | Jane    | Grossman |
|     15 | Frank   | Portman  |
+--------+---------+----------+
4 rows in set (0.00 sec)
```

Для этого запроса сервер может использовать в качестве индекса первичный ключ для столбца `emp_id`, чтобы найти местоположение сотрудников с ID 1, 3, 9, 15 и 22 в таблице `employee`, и затем просмотреть каждую из пяти строк, чтобы извлечь значения столбцов имени и фамилии.

Однако если индекс содержит все, что необходимо для удовлетворения запроса, серверу не надо просматривать ассоциированную таблицу. Для иллюстрации давайте посмотрим на то, как обращается оптимизатор запросов с одним и тем же запросом при разных типах индексов.

Запрос, агрегирующий остатки на счетах для определенных клиентов, выглядит так:

```
mysql> SELECT cust_id, SUM(avail_balance) tot_bal
    -> FROM account
    -> WHERE cust_id IN (1, 5, 9, 11)
    -> GROUP BY cust_id;
+---------+----------+
| cust_id | tot_bal  |
+---------+----------+
|       1 |  4557.75 |
|       5 |  2237.97 |
|       9 | 10971.22 |
|      11 |  9345.55 |
+---------+----------+
4 rows in set (0.00 sec)
```

С помощью выражения `explain` (объяснить) рассмотрим, как оптимизатор запросов MySQL принимает решение об обработке запроса. Сервер не будет выполнять запрос, а покажет план его выполнения:

```
mysql> EXPLAIN SELECT cust_id, SUM(avail_balance) tot_bal
    -> FROM account
    -> WHERE cust_id IN (1, 5, 9, 11)
    -> GROUP BY cust_id \G
*************************** 1. row ***************************
```

```
           id: 1
  select_type: SIMPLE
        table: account
         type: index
possible_keys: fk_a_cust_id
          key: fk_a_cust_id
      key_len: 4
          ref: NULL
         rows: 24
        Extra: Using where
1 row in set (0.00 sec)
```

Каждый сервер БД включает инструменты, позволяющие увидеть, как оптимизатор запросов обрабатывает SQL-выражение. В SQL Server увидеть план выполнения перед выполнением SQL-выражения позволяет выражение `set showplan_text on`. В Oracle Database есть выражение `explain plan`, записывающее план выполнения в специальную таблицу `plan_table`.

Если не вдаваться в детали, то план выполнения сообщает следующее:

- Индекс `fk_a_cust_id` используется для поиска строк таблицы `account`, которые удовлетворяют условию блока `where`.
- После прочтения индекса ожидается, что сервер просмотрит все 24 строки таблицы `account` для сбора данных о доступных остатках, поскольку он не знает, что могут существовать другие клиенты, кроме клиентов с ID 1, 5, 9 и 11.

Индекс `fk_a_cust_id` – еще один индекс, автоматически сгенерированный сервером, но на этот раз из-за ограничения внешнего ключа, а не ограничения первичного ключа (более подробно об этом позже в этой главе). Индекс `fk_a_cust_id` создан для столбца `account.cust_id`, таким образом, сервер использует его для определения местоположения клиентов с ID 1, 5, 9 и 11 в таблице `account`, а затем посещает эти строки для извлечения и агрегирования данных по доступным остаткам.

Далее добавим новый индекс с названием `acc_bal_idx` для обоих столбцов `cust_id` и `avail_balance`:

```
mysql> ALTER TABLE account
    -> ADD INDEX acc_bal_idx (cust_id, avail_balance);
Query OK, 24 rows affected (0.03 sec)
Records: 24  Duplicates: 0  Warnings: 0
```

Теперь, имея этот индекс, посмотрим, как оптимизатор будет обрабатывать тот же запрос:

```
mysql> EXPLAIN SELECT cust_id, SUM(avail_balance) tot_bal
    -> FROM account
    -> WHERE cust_id IN (1, 5, 9, 11)
    -> GROUP BY cust_id \G
*************************** 1. row ***************************
```

```
           id: 1
  select_type: SIMPLE
        table: account
         type: range
possible_keys: acc_bal_idx
          key: acc_bal_idx
      key_len: 4
          ref: NULL
         rows: 8
        Extra: Using where; Using index
1 row in set (0.01 sec)
```

Сравнение двух планов выполнения дает следующие различия:

- Вместо индекса `fk_a_cust_id` оптимизатор использует новый индекс `acc_bal_idx`.
- Оптимизатор ускоряет свою работу, поскольку теперь ему надо обработать только восемь строк вместо 24.
- Для обеспечения результатов запроса таблица `account` не нужна (обозначена через `Using index` в столбце `Extra`).

Следовательно, сервер может определить с помощью индексов местоположение строк в ассоциированной таблице или использовать индекс как таблицу, если он содержит все столбцы, необходимые для удовлетворения запроса.

Только что рассмотренный процесс – это пример оптимизации запроса. Оптимизация включает изучение SQL-выражения и установление доступных серверу ресурсов, необходимых для выполнения этого выражения. Чтобы обеспечить более эффективное выполнение выражения, можно изменить SQL-выражение или настроить ресурсы БД, или сделать и то, и другое. Оптимизация – тонкий вопрос, поэтому настоятельно рекомендую прочитать руководство по оптимизации для используемого сервера или найти хорошую книгу по этой теме, чтобы увидеть все возможные подходы к оптимизации для конкретного сервера.

Обратная сторона индексации

Если индексы – это так замечательно, почему бы не индексировать все подряд? Чтобы понять, почему большее количество индексов не всегда хорошо, надо помнить, что каждый индекс – это таблица (особого типа, но все равно таблица). Поэтому при каждом добавлении или удалении строки из таблицы все ее индексы должны быть модифицированы. При обновлении строки все задействованные индексы столбца или столбцов тоже должны изменяться. Следовательно, чем больше индексов, тем больше работы приходится выполнять серверу для обновления всех объектов схемы. А это очень замедляет процесс.

Индексам требуется некоторое количество памяти, а также некоторое внимание администраторов, поэтому наилучшая стратегия – добавлять

индекс только при возникновении вполне определенной необходимости. Если индекс нужен только для конкретной цели, например для выполнения программы по ежемесячному обслуживанию, всегда можно добавить индекс, выполнить программу и удалить индекс до того момента, пока он не понадобится вновь. Для хранилищ данных, где индексы очень нужны в рабочие часы (поскольку пользователи составляют отчеты и выполняют случайные запросы), но становятся проблемой при загрузке данных в хранилище в ночное время, общепринятой практикой является уничтожение индексов перед загрузкой данных и их повторное создание перед открытием хранилищ для работы.

Ограничения

Ограничение – это просто некоторое ограничивающее условие, налагаемое на один или более столбцов таблицы. Есть несколько разных типов ограничений, включая:

Ограничения первичного ключа (Primary-key constraints)

Идентифицируют столбец или столбцы, гарантирующие уникальность в рамках таблицы.

Ограничения внешнего ключа (Foreign-key constraints)

На один или более столбцов накладывается такое ограничение: они могут содержать только значения, содержащиеся в столбцах первичного ключа другой таблицы. Также могут ограничиваться допустимые значения других таблиц, если установлены правила `update cascade` (каскадное обновление) или `delete cascade` (каскадное удаление).

Ограничения уникальности (Unique constraints)

На один или более столбцов накладывается ограничение: они могут содержать только уникальные в рамках таблицы значения.

Проверочные ограничения целостности (Check constraints)

Ограничивают допустимые значения столбца.

Без ограничений под сомнением может оказаться непротиворечивость базы данных. Например, если сервер допускает изменение ID клиента в таблице `customer` без изменения этого же ID клиента в таблице `account`, все может закончиться тем, что счета больше не будут указывать на соответствующие записи клиентов (это явление известно как «осиротевшие» строки (orphaned rows)). Но если заданы ограничения первичного и внешнего ключей, сервер или сформирует ошибку в случае попытки изменения или удаления данных, на которые ссылаются другие таблицы, или распространит изменения на другие таблицы (более подробно об этом чуть позже).

Если при работе с сервером MySQL требуются ограничения внешнего ключа, в таблицах должен использоваться механизм хранения InnoDB.

Создание ограничений

Обычно ограничения создают одновременно с ассоциированной таблицей посредством выражения `create table`. Для иллюстрации приведем пример из сценария формирования схемы для БД, используемой в качестве примера к этой книге:

```
CREATE TABLE product
 (product_cd VARCHAR(10) NOT NULL,
  name VARCHAR(50) NOT NULL,
  product_type_cd VARCHAR (10) NOT NULL,
  date_offered DATE,
  date_retired DATE,
  CONSTRAINT fk_product_type_cd FOREIGN KEY (product_type_cd)
    REFERENCES product_type (product_type_cd),
  CONSTRAINT pk_product PRIMARY KEY (product_cd)
 );
```

Таблица `product` включает два ограничения: первое определяет столбец `product_cd` как первичный ключ таблицы, а второе – столбец `product_type_cd` как внешний ключ к таблице `product_type`. Альтернативный вариант: таблицу `product` можно было создать без ограничений, а ограничения первичного и внешнего ключей добавить позже посредством выражений `alter table`:

```
ALTER TABLE product
ADD CONSTRAINT pk_product PRIMARY KEY (product_cd);

ALTER TABLE product
ADD CONSTRAINT fk_product_type_cd FOREIGN KEY (product_type_cd)
  REFERENCES product_type (product_type_cd);
```

Если требуется убрать ограничения первичного или внешнего ключей, можно опять воспользоваться выражением `alter table`, только в этом случае задается `drop`, а не `add`:

```
ALTER TABLE product
DROP PRIMARY KEY;

ALTER TABLE product
DROP FOREIGN KEY fk_product_type_cd;
```

Ограничение первичного ключа обычно не удаляется, а ограничения внешнего ключа иногда отменяются во время определенных операций обслуживания, а потом устанавливаются вновь.

Ограничения и индексы

Как было показано в этой главе ранее, иногда создание ограничений включает автоматическое формирование индекса. Однако серверы БД ведут себя по-разному в зависимости от связи между ограничениями и индексами. В табл. 13.1 показаны связи между ограничениями и индексами в MySQL, SQL Server и Oracle Database.

Таблица 13.1. Формирование ограничений

Тип ограничения	MySQL	SQL Server	Oracle Database
Ограничение первичного ключа	Формирует уникальный индекс	Формирует уникальный индекс	Использует имеющийся индекс или создает новый
Ограничение внешнего ключа	Формирует индекс	Не формирует индекс	Не формирует индекс
Ограничение уникальности	Формирует уникальный индекс	Формирует уникальный индекс	Использует имеющийся индекс или создает новый

Следовательно, MySQL формирует новый индекс, чтобы реализовать ограничения первичного ключа, внешнего ключа и уникальности. SQL Server формирует новый индекс для ограничений первичного ключа и уникальности, а для ограничений внешнего ключа – *нет*. Oracle Database использует такой же подход, как и SQL Server, за тем исключением, что Oracle для введения в действие ограничений первичного ключа и уникальности будет использовать существующий индекс (если есть подходящий). Хотя ни SQL Server, ни Oracle Database не формируют индекс для ограничения внешнего ключа, документация обоих серверов рекомендует создавать индекс для каждого внешнего ключа.

Каскадные ограничения

Если при наличии ограничений внешнего ключа пользователь пытается вставить новую строку или изменить существующую и при этом получается, что столбец внешнего ключа не имеет соответствующего значения в родительской таблице, сервер формирует ошибку. Для иллюстрации рассмотрим данные таблиц `product` и `product_type`:

```
mysql> SELECT product_type_cd, name
    -> FROM product_type;
+-----------------+-------------------------------+
| product_type_cd | name                          |
+-----------------+-------------------------------+
| ACCOUNT         | Customer Accounts             |
| INSURANCE       | Insurance Offerings           |
| LOAN            | Individual and Business Loans |
+-----------------+-------------------------------+
3 rows in set (0.00 sec)

mysql> SELECT product_type_cd, product_cd, name
    -> FROM product
    -> ORDER BY product_type_cd;
+-----------------+------------+-------------------------+
| product_type_cd | product_cd | name                    |
+-----------------+------------+-------------------------+
| ACCOUNT         | CD         | certificate of deposit  |
| ACCOUNT         | CHK        | checking account        |
```

```
| ACCOUNT         | MM         | money market account    |
| ACCOUNT         | SAV        | savings account          |
| LOAN            | AUT        | auto loan                |
| LOAN            | BUS        | business line of credit |
| LOAN            | MRT        | home mortgage            |
| LOAN            | SBL        | small business loan      |
+-----------------+------------+--------------------------+
8 rows in set (0.01 sec)
```

В таблице product_type имеется три разных значения для столбца product_type_cd (ACCOUNT, INSURANCE и LOAN). Два из этих трех значений (ACCOUNT и LOAN) упоминаются в столбце product_type_cd таблицы product.

Следующее выражение делает попытку изменить значение столбца product_type_cd таблицы product на значение, которого нет в таблице product_type:

```
mysql> UPDATE product
    -> SET product_type_cd = 'XYZ'
    -> WHERE product_type_cd = 'LOAN';
ERROR 1216 (23000): Cannot add or update a child row: a foreign key constraint
fails
```

Из-за ограничения внешнего ключа на столбец product.product_type_cd сервер не разрешает провести такое обновление, поскольку в таблице product_type в столбце product_type_cd нет строки со значением 'XYZ'. Таким образом, ограничение внешнего ключа не позволяет изменить дочернюю строку, если в родительской нет соответствующего значения.

Однако что произошло бы, попробуй мы изменить значение *родительской* строки таблицы product_type на 'XYZ'? Вот выражение update, реализующее попытку изменить тип счета LOAN на XYZ:

```
mysql> UPDATE product_type
    -> SET product_type_cd = 'XYZ'
    -> WHERE product_type_cd = 'LOAN';
ERROR 1217 (23000): Cannot delete or update a parent row: a foreign
                                                          key constraint
fails
```

Опять формируется ошибка. На этот раз потому, что в таблице product есть дочерние строки, столбец product_type_cd которых содержит значение 'LOAN'. Это поведение ограничений внешнего ключа по умолчанию, но оно не единственное возможное. Можно указать серверу распространять это изменение на все дочерние строки, сохраняя, таким образом, целостность данных. Эта разновидность ограничения внешнего ключа, известная как *каскадное обновление* (*cascading update*), может быть установлена путем удаления существующего внешнего ключа и добавления нового, включающего блок on update cascade:

```
mysql> ALTER TABLE product
    -> DROP FOREIGN KEY fk_product_type_cd;
Query OK, 8 rows affected (0.02 sec)
```

```
Records: 8  Duplicates: 0  Warnings: 0

mysql> ALTER TABLE product
    -> ADD CONSTRAINT fk_product_type_cd FOREIGN KEY (product_type_cd)
    ->   REFERENCES product_type (product_type_cd)
    ->   ON UPDATE CASCADE;
Query OK, 8 rows affected (0.03 sec)
Records: 8  Duplicates: 0  Warnings: 0
```

Изменив ограничение таким образом, посмотрим, что произойдет, если попытаться выполнить выражение `update` снова:

```
mysql> UPDATE product_type
    -> SET product_type_cd = 'XYZ'
    -> WHERE product_type_cd = 'LOAN';
Query OK, 1 row affected (0.01 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

На этот раз выражение выполняется успешно. Для проверки того, что изменения были распространены на таблицу `product`, еще раз посмотрим на данные в обеих таблицах:

```
mysql> SELECT product_type_cd, name
    -> FROM product_type;
+-----------------+-------------------------------+
| product_type_cd | name                          |
+-----------------+-------------------------------+
| ACCOUNT         | Customer Accounts             |
| INSURANCE       | Insurance Offerings           |
| XYZ             | Individual and Business Loans |
+-----------------+-------------------------------+
3 rows in set (0.02 sec)

mysql> SELECT product_type_cd, product_cd, name
    -> FROM product
    -> ORDER BY product_type_cd;
+-----------------+------------+-------------------------+
| product_type_cd | product_cd | name                    |
+-----------------+------------+-------------------------+
| ACCOUNT         | CD         | certificate of deposit  |
| ACCOUNT         | CHK        | checking account        |
| ACCOUNT         | MM         | money market account    |
| ACCOUNT         | SAV        | savings account         |
| XYZ             | AUT        | auto loan               |
| XYZ             | BUS        | business line of credit |
| XYZ             | MRT        | home mortgage           |
| XYZ             | SBL        | small business loan     |
+-----------------+------------+-------------------------+
8 rows in set (0.01 sec)
```

Как видите, изменения таблицы `product_type` распространились и на таблицу `product`. Кроме каскадных обновлений можно задавать *каскадные удаления* (*cascading deletes*). При каскадном удалении, если стро-

ка удаляется в родительской таблице, соответствующие ей строки удаляются и в дочерней таблице. Для задания каскадного удаления используется блок on delete cascade:

```
ALTER TABLE product
ADD CONSTRAINT fk_product_type_cd FOREIGN KEY (product_type_cd)
  REFERENCES product_type (product_type_cd)
  ON UPDATE CASCADE
  ON DELETE CASCADE;
```

Теперь при таком варианте ограничения сервер будет обновлять дочерние строки таблицы product при обновлении строки в таблице product_type, а также удалять дочерние строки таблицы product при удалении строки таблицы product_type.

Каскадные ограничения – один из случаев, когда ограничения *непосредственно* влияют на код, который вы пишете. Чтобы полностью представлять эффект применения выражений update и delete, необходимо знать, для каких ограничений базы данных заданы каскадные обновления и/или удаления.

A

ER-диаграмма примера базы данных

На рис. A.1 представлена диаграмма сущностей и связей (entity-relationship, ER) базы данных, используемой в этой книге в качестве примера. Как следует из названия, диаграмма отображает сущности, или таблицы, базы данных и связи внешнего ключа между таблицами. Вот несколько подсказок, которые помогут понять условные обозначения:

- Каждый прямоугольник представляет таблицу. Имя таблицы указано в верхнем левом углу прямоугольника. Столбец (или столбцы) первичного ключа указан первым и отделен от обычных столбцов линией. Обычные столбцы перечислены под линией, столбцы внешнего ключа отмечены как «(FK)».
- Линиями между таблицами представлены связи внешнего ключа. Отметки на концах линий показывают допустимую кратность связи, которая может иметь значения нуль (0), один (1) или много (≤). Например, взглянув на связь между таблицами `account` и `product`, можно сказать, что счет должен относиться только к одной услуге, но для одной услуги может быть нуль, один или много счетов.

Более подробно моделирование баз данных и соответствующие инструменты рассмотрены в приложении D.

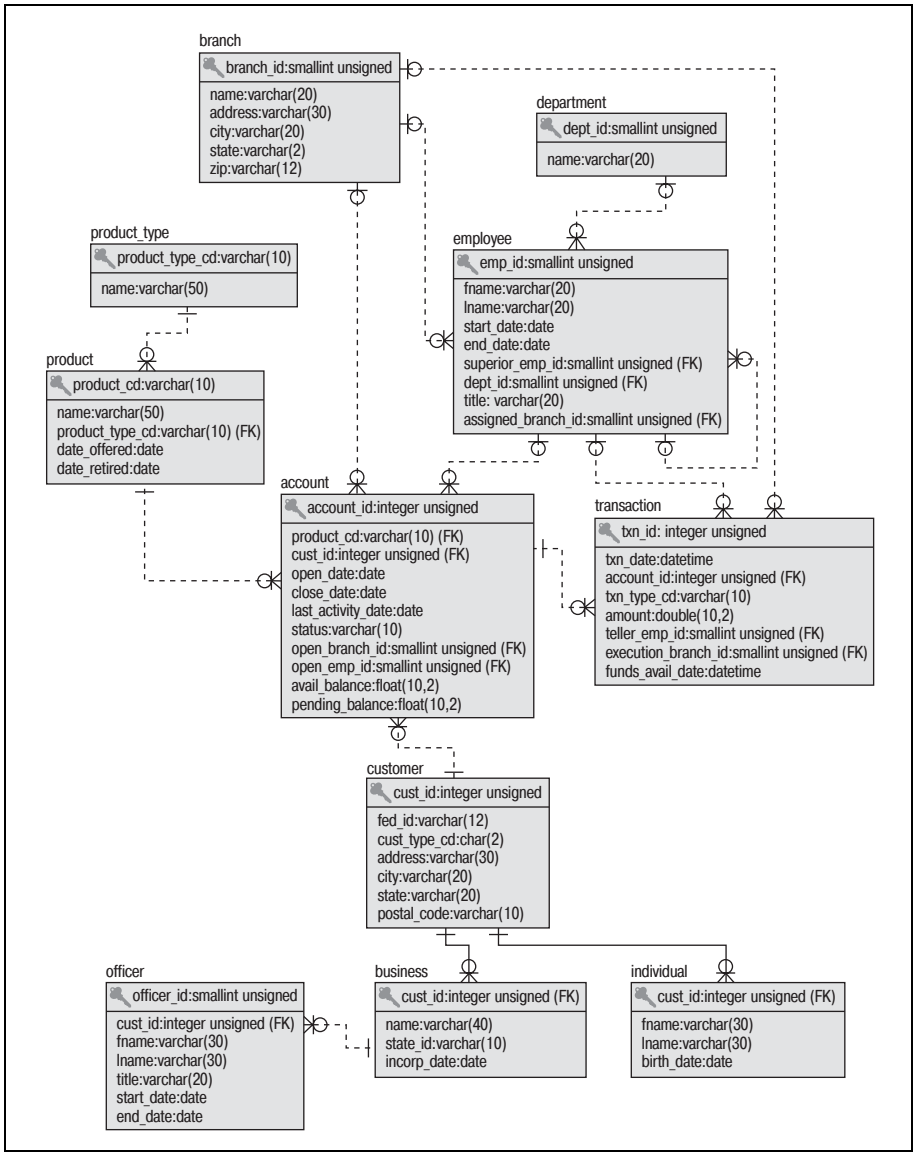


Рис. А.1. *ER-диаграмма*

B

MySQL-расширения языка SQL

Поскольку для всех примеров в данной книге используется сервер MySQL, я подумал, что для читателей, планирующих продолжать работать с MySQL, будет полезным включить приложение, посвященное MySQL-расширениям языка SQL. Это приложение рассматривает некоторые MySQL-расширения выражений `select`, `insert`, `update` и `delete`, очень полезные в определенных ситуациях.

Расширения выражения Select

Реализация выражения `select` в MySQL включает два дополнительных блока, обсуждаемых в следующих разделах.

Блок limit

В некоторых ситуациях нас не интересуют *все* строки, возвращаемые запросом. Например, можно создать запрос, выбирающий всех операционистов банка и все номера счетов, открытых каждым из них. Если цель запроса – выявить трех лучших операционистов для вручения награды от банка, необязательно знать, кто будет четвертым, пятым и т. д. Для разрешения подобных ситуаций выражение `select` MySQL включает блок `limit`, позволяющий ограничить число возвращаемых запросом строк.

Чтобы продемонстрировать использование блока `limit`, начнем с построения запроса, показывающего количество счетов, открытых каждым операционистом банка:

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id;
+-------------+----------+
```

```
| open_emp_id | how_many |
+-------------+----------+
|           1 |        8 |
|          10 |        7 |
|          13 |        3 |
|          16 |        6 |
+-------------+----------+
4 rows in set (0.31 sec)
```

Результат показал, что счета открывали четверо разных сотрудников. Если требуется ограничить результирующий набор только тремя записями, можно добавить блок `limit`. Он определяет, что должны быть возвращены только три записи:

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id
    -> LIMIT 3;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|           1 |        8 |
|          10 |        7 |
|          13 |        3 |
+-------------+----------+
3 rows in set (0.06 sec)
```

Теперь благодаря блоку `limit` (четвертая строка запроса) результирующий набор включает только три строки. Четвертый операционист (служащий с ID 16) удален из результирующего набора.

Сочетание блока limit с блоком order by

Предыдущий запрос возвращает три записи, но есть небольшая проблема: запрос не описывает, *какие* три записи из четырех нас интересуют. Если требуется выбрать *конкретные* три записи, например трех операционистов, открывших больше всего счетов, придется использовать блок `limit` вместе с блоком `order by`:

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id
    -> ORDER BY how_many DESC
    -> LIMIT 3;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|           1 |        8 |
|          10 |        7 |
|          16 |        6 |
+-------------+----------+
3 rows in set (0.03 sec)
```

Разница между этим и предыдущим запросами в том, что теперь блок `limit` применяется к упорядоченному набору. В итоге в конечном результирующем наборе имеем трех сотрудников, открывших наибольшее число счетов. Если требуется не произвольная выборка записей, обычно блок `limit` используется в паре с блоком `order by`.

Блок `limit` применяется после всех фильтров, группировок и расстановок, поэтому он никогда не изменит результат выражения `select`, только ограничит число возвращаемых им записей.

Необязательный второй параметр блока limit

Допустим, теперь вместо поиска трех лучших операционистов поставлена задача выбрать всех, кроме двух лучших (вместо награждения лучших исполнителей банк пошлет несколько самых непроизводительных операционистов на тренинг по повышению самооценки). Для подобных ситуаций блок `limit` предоставляет необязательный второй параметр. Если используются оба параметра, первый указывает, с какой строки добавлять записи в конечный результирующий набор, а второй – сколько строк включить. Обозначая записи порядковыми номерами, помните, что в MySQL первой записью является запись под номером 0. Следовательно, если стоит задача найти третьего лучшего работника, можно сделать следующее:

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id
    -> ORDER BY how_many DESC
    -> LIMIT 2, 1;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|          16 |        6 |
+-------------+----------+
1 row in set (0.00 sec)
```

В этом примере нулевая и первая записи отбрасываются; включаются записи, начиная со второй. Поскольку второй параметр блока `limit` равен 1, получаем только одну запись.

Если требуется начать со второй позиции и включить *все* оставшиеся записи, можно сделать второй аргумент блока `limit` достаточно большим, чтобы все оставшиеся записи гарантированно вошли в результирующий набор. Если неизвестно, сколько операционистов открывали новые счета, для выбора всех работников, кроме двух лучших, можно было бы сделать примерно следующее:

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id
    -> ORDER BY how_many DESC
```

```
    -> LIMIT 2, 999999999;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|          16 |        6 |
|          13 |        3 |
+-------------+----------+
2 rows in set (0.00 sec)
```

В этом варианте запроса отбрасываются нулевая и первая записи, а в результат включаются все записи вплоть до 999 999 999, начиная со второй (в данном случае таких записей всего две, но лучше немного переусердствовать, чем потерять нужные записи, недооценив их количество).

Ранжирующие запросы

Запросы, включающие блок `limit` в сочетании с блоком `order by`, можно назвать *ранжирующими запросами* (*ranking queries*), потому что они позволяют ранжировать данные. Я уже продемонстрировал пример ранжирования банковских сотрудников по числу открытых счетов. Но ранжирующие запросы используются для решения многих прикладных задач, таких как поиск:

- Пяти лучших продавцов 2005 года
- Третьего по числу круговых пробежек игрока в истории бейсбола
- 98 бестселлеров всех времен и народов, кроме Библии и цитатника Мао
- Двух самых непопулярных видов мороженого

Уже было рассмотрено, как найти трех лучших операционистов, третьего лучшего и всех, кроме двух лучших. Если я хочу сделать что-то аналогичное для четвертого примера (например, найти *худших* сотрудников), требуется просто изменить порядок сортировки на обратный, так чтобы результаты располагались, начиная с наименьшего числа открытых счетов и до наибольшего:

```
mysql> SELECT open_emp_id, COUNT(*) how_many
    -> FROM account
    -> GROUP BY open_emp_id
    -> ORDER BY how_many ASC
    -> LIMIT 2;
+-------------+----------+
| open_emp_id | how_many |
+-------------+----------+
|          13 |        3 |
|          16 |        6 |
+-------------+----------+
2 rows in set (0.24 sec)
```

За счет простого изменения порядка сортировки (с `ORDER BY how_many DESC` на `ORDER BY how_many ASC`) теперь запрос возвращает двух наихудших операционистов. Таким образом, с помощью блока с возрастающим

или убывающим порядком сортировки можно создавать ранжирующие запросы для решения большинства типовых прикладных задач.

Блок into outfile

Если требуется записать результат запроса в файл, можно выделить его, скопировать в буфер обмена и вставить в свой любимый редактор. Однако если результирующий набор запроса достаточно велик или если запрос выполняется из сценария, необходим способ записывать результаты в файл без участия пользователя. Для помощи в таких ситуациях MySQL включает блок `into outfile` (в выходной файл), в котором можно задать имя файла для записи результатов. Вот пример записи результатов запроса в каталог *c:\temp*:

```
mysql> SELECT emp_id, fname, lname, start_date
    -> INTO OUTFILE 'C:\\TEMP\\emp_list.txt'
    -> FROM employee;
Query OK, 18 rows affected (0.20 sec)
```

Как говорилось в главе 7, обратный слэш используется для экранирования символов в строке. Поэтому в Windows для задания пути потребуется ставить по два обратных слэша подряд.

Результаты запроса не выводятся на экран, а записываются в файл *emp_list.txt* и выглядят так:

```
1    Michael    Smith    2001-06-22
2    Susan    Barker    2002-09-12
3    Robert    Tyler    2000-02-09
4    Susan    Hawthorne    2002-04-24
...
16    Theresa    Markham    2001-03-15
17    Beth    Fowler    2002-06-29
18    Rick    Tulman    2002-12-12
```

Формат по умолчанию использует символ табуляции (`'\t'`) между столбцами и символ новой строки (`'\n'`) после каждой записи. Если требуется дополнительное форматирование данных, можно включить в блок `into outfile` несколько подблоков. Например, если надо представить данные в формате, называемом *форматом с разделителем «|»* (*pipe-delimited format*), то в подблоке `fields` (поля) можно в качестве символа-разделителя столбцов задать символ `'|'`:

```
mysql> SELECT emp_id, fname, lname, start_date
    -> INTO OUTFILE 'C:\\TEMP\\emp_list_delim.txt'
    ->   FIELDS TERMINATED BY '|'
    -> FROM employee;
Query OK, 18 rows affected (0.02 sec)
```

MySQL не позволяет перезаписывать существующий файл с помощью `into outfile`, поэтому если один и тот же запрос выпол-

няется больше одного раза, перед каждым выполнением следует удалить имеющийся файл.

Содержимое файла *emp_list_delim.txt* выглядит так:

```
1|Michael|Smith|2001-06-22
2|Susan|Barker|2002-09-12
3|Robert|Tyler|2000-02-09
4|Susan|Hawthorne|2002-04-24
...
16|Theresa|Markham|2001-03-15
17|Beth|Fowler|2002-06-29
18|Rick|Tulman|2002-12-12
```

Кроме формата с разделителями «|» можно форматировать данные запятыми (*формат с разделителями-запятыми*, *comma-delimited format*). В этом случае следует задать `fields terminated by ','`. Однако если записываемые в файл данные включают строки, применение запятых в качестве разделителей полей может вызвать проблемы. Запятые встречаются в строках намного чаще, чем символ «|». Рассмотрим следующий запрос, записывающий число и две строки, разделенные запятыми, в файл *comma1.txt*:

```
mysql> SELECT data.num, data.str1, data.str2
    -> INTO OUTFILE 'C:\\TEMP\\comma1.txt'
    ->   FIELDS TERMINATED BY ','
    -> FROM
    ->  (SELECT 1 num, 'This string has no commas' str1,
    ->     'This string, however, has two commas' str2) data;
Query OK, 1 row affected (0.04 sec)
```

Поскольку третий столбец выходного файла (`str2`) – строка, содержащая запятые, можно предположить, что у приложения, считывающего файл *comma1.txt*, возникнут проблемы при синтаксическом разборе строк и распределении их по столбцам. Но сервер MySQL предпринял специальные меры предосторожности для подобных ситуаций. Вот содержимое файла *comma1.txt*:

```
1,This string has no commas,This string\, however\, has two commas
```

Как видите, запятые в третьем столбце экранированы обратным слэшем, размещенным перед ними. Если выполнить этот же запрос, но использовать формат с разделителями «|», запятые *не* будут экранированы. Если хочется использовать в выходном файле другой символ экранирования, например еще одну запятую, его можно задать с помощью подблока `fields escaped by`.

Кроме разделителей столбцов можно задавать символ, используемый для разделения разных записей в файле данных. Если требуется, чтобы каждая запись в выходном файле была отделена не символом новой строки, а каким-то другим, можно воспользоваться подблоком `lines`:

```
mysql> SELECT emp_id, fname, lname, start_date
```

```
    -> INTO OUTFILE 'C:\\TEMP\\emp_list_atsign.txt'
    ->   FIELDS TERMINATED BY '|'
    ->   LINES TERMINATED BY '@'
    -> FROM employee;
Query OK, 18 rows affected (0.03 sec)
```

Поскольку в файле *emp_list_atsign.txt* для разделения записей не используется символ новой строки, весь файл выглядит как одна длинная текстовая строка, в которой каждая запись отделена символом '@':

```
1|Michael|Smith|2001-06-22@2|Susan|Barker|2002-09-12@3|Robert|Tyler|
2000-02-09@4|Susan|Hawthorne|2002-04-24@5|John|Gooding|2003-11-14@6|Helen|
Fleming|2004-03-17@7|Chris|Tucker|2004-09-15@8|Sarah|Parker|2002-12-02@9|
Jane|Grossman|2002-05-03@10|Paula|Roberts|2002-07-27@11|Thomas|Ziegler|
2000-10-23@12|Samantha|Jameson|2003-01-08@13|John|Blake|2000-05-11@14|
Cindy|Mason|2002-08-09@15|Frank|Portman|2003-04-01@16|Theresa|Markham|
2001-03-15@17|Beth|Fowler|2002-06-29@18|Rick|Tulman|2002-12-12@
```

Если понадобится сгенерировать файл данных для загрузки в электронную таблицу или рассылки в/за пределы организации, блок `into outfile` обеспечит достаточную гибкость для создания файла любого необходимого формата.

Сочетание выражений insert/update

Допустим, требуется создать таблицу для сбора информации о посещении клиентами отделений банка. Таблица должна содержать ID клиента, ID отделения и столбец `datetime` с датой и временем последнего посещения отделения клиентом. Строки в таблицу добавляются всякий раз, когда клиент посещает определенное отделение. Но если клиент уже был в этом отделении, следует просто обновить столбец `datetime` существующей строки. Вот описание таблицы:

```
CREATE TABLE branch_usage
 (branch_id SMALLINT UNSIGNED NOT NULL,
  cust_id INTEGER UNSIGNED NOT NULL,
  last_visited_on DATETIME,
  CONSTRAINT pk_branch_usage PRIMARY KEY (branch_id, cust_id)
 );
```

Кроме трех столбцов таблица `branch_usage` определяет ограничение первичного ключа для столбцов `branch_id` и `cust_id`. Следовательно, сервер отклонит любую добавляемую в таблицу строку, пара значений отделение/клиент которой уже есть в таблице.

Скажем, таблица создана, и клиент с ID 5 посещает главное отделение (отделение с ID 1) за первую неделю три раза. После первого визита в таблицу `branch_usage` можно вставить запись, поскольку для клиента с ID 5 и отделения с ID 1 записи еще нет:

```
mysql> INSERT INTO branch_usage (branch_id, cust_id, last_visited_on)
    -> VALUES (1, 5, CURRENT_TIMESTAMP(  ));
Query OK, 1 row affected (0.02 sec)
```

Однако при следующем посещении клиентом того же отделения потребуется *обновить* существующую запись, а не вставлять новую. В противном случае будет получена следующая ошибка:

```
ERROR 1062 (23000): Duplicate entry '1-5' for key 1
```

Чтобы избежать этой ошибки, можно запросить таблицу branch_usage и посмотреть, имеется ли данная пара значений клиент/отделение, а затем уже вставить запись, если таковой не найдено, или обновить имеющуюся строку, если она уже существует. Однако чтобы избавить пользователей от хлопот, разработчики MySQL расширили выражение insert и обеспечили возможность определять необходимость изменения одного или нескольких столбцов, если выражение insert дает сбой из-за дублирования ключей. Следующее выражение предписывает серверу изменять столбец last_visited_on, если данные клиент и отделение уже есть в таблице branch_usage:

```
mysql> INSERT INTO branch_usage (branch_id, cust_id, last_visited_on)
    -> VALUES (1, 5, CURRENT_TIMESTAMP(  ))

    -> ON DUPLICATE KEY UPDATE last_visited_on = CURRENT_TIMESTAMP(  );
Query OK, 2 rows affected (0.02 sec)
```

Блок on duplicate key (при дублировании ключа) позволяет выполнять одно и то же выражение при каждом появлении клиента с ID 5 в отделении с ID 1. Если выражение выполняется 100 раз, в результате первого прогона в таблицу добавляется одна строка. Следующие 99 выполнений обеспечивают изменение столбца last_visited_on соответственно

Замещение команды replace

До версии 4.1 сервера MySQL операции с возможностью обновления и вставки осуществлялись с помощью команды replace (заместить). Это собственное выражение, которое сначала удаляет существующую строку, если такое значение первичного ключа уже существует, а потом уже вставляет новую строку в таблицу. Выполняя операции с возможностью обновления и вставки при работе с версией 4.1 и более поздними, можно выбирать между командами replace и insert...on duplicate key.

Однако команда replace выполняет операцию удаления при встрече дублирующихся значений ключей, что может обусловить цепную реакцию, если используется механизм хранения InnoDB и наложены ограничения внешнего ключа. Если ограничения созданы посредством опции on delete cascade, при удалении строки целевой таблицы команда replace может автоматически удалить и строки других таблиц. Поэтому обычно более безопасным считается использование блока on duplicate key выражения insert, а не более старой команды replace.

текущему времени. Такой тип операций часто называют *операциями с возможностью обновления и вставки* (*upsert*), т. е. сочетанием выражений `update` и `insert`.

Упорядоченные обновления и удаления

Ранее здесь было показано, как с помощью сочетания блоков `limit` и `order by` можно писать запросы, формирующие ранжированную выборку (например, три лучших сотрудника по количеству открытых счетов). MySQL тоже позволяет использовать блоки `limit` и `order by` в выражениях `update` и `delete`, обеспечивая таким образом возможность изменять или удалять определенные строки таблицы на основании их ранга. Предположим, например, что требуется удалить строки таблицы, используемой для отслеживания регистраций пользователей в онлайновой банковской системе. Вот таблица, отслеживающая ID клиента и дату/время регистрации:

```
CREATE TABLE login_history
 (cust_id INTEGER UNSIGNED NOT NULL,
  login_date DATETIME,
  CONSTRAINT pk_login_history PRIMARY KEY (cust_id, login_date)
 );
```

Следующее выражение заполняет таблицу `login_history` некоторыми данными путем формирования перекрестного соединения между таблицами `account` и `customer` и формирования дат регистрации на основании значений столбца `open_date` таблицы `account`:

```
mysql> INSERT INTO login_history (cust_id, login_date)
    -> SELECT c.cust_id,
    ->   ADDDATE(a.open_date, INTERVAL a.account_id * c.cust_id HOUR)
    -> FROM customer c CROSS JOIN account a;
Query OK, 312 rows affected (0.03 sec)
Records: 312  Duplicates: 0  Warnings: 0
```

Теперь таблица заполнена 312 строками относительно случайных данных. Ваша задача – раз в месяц просмотреть данные таблицы `login_history`, составить для руководства отчет о тех, кто использует онлайновую банковскую систему, и затем удалить все записи кроме 50 последних. Один из возможных подходов – написать запрос с использованием блоков `order by` и `limit` для поиска 50 самых свежих регистраций:

```
mysql> SELECT login_date
    -> FROM login_history
    -> ORDER BY login_date DESC
    -> LIMIT 49,1;
+---------------------+
| login_date          |
+---------------------+
| 2004-07-02 09:00:00 |
+---------------------+
1 row in set (0.00 sec)
```

Вооружившись этой информацией, уже можно создать выражение delete, удаляющее все строки, значение столбца login_date которых меньше даты, возвращенной запросом:

```
mysql> DELETE FROM login_history
    -> WHERE login_date < '2004-07-02 09:00:00';
Query OK, 262 rows affected (0.02 sec)
```

Теперь таблица содержит 50 последних регистраций. Однако MySQL-расширения позволяют достичь тех же результатов с помощью единственного выражения delete с блоками limit и order by. Возвратив исходные 312 строк в таблицу login_history, можно выполнить следующее выражение:

```
mysql> DELETE FROM login_history
    -> ORDER BY login_date ASC
    -> LIMIT 262;
Query OK, 262 rows affected (0.05 sec)
```

Это выражение сортирует строки по возрастанию значений столбца login_date, затем первые 262 строки удаляются, а 50 самых свежих строк остаются.

В этом примере для построения блока limit необходимо было знать число строк в таблице (312 исходных строк – 50 оставшихся строк = 262 удалений). Лучше было бы отсортировать строки в убывающем порядке, указать серверу пропустить первые 50 строк и затем удалить оставшиеся строки:

```
DELETE FROM login_history
ORDER BY login_date DESC
LIMIT 49, 9999999;
```

Однако MySQL не обеспечивает возможности применения второго необязательного параметра при использовании блока limit в выражениях delete или update.

С помощью блоков limit и order by можно не только удалять, но и обновлять данные. Например, если банк решает для удержания лояльных клиентов добавить по 100 долларов на каждый из десяти самых старых счетов, можно сделать следующее:

```
mysql> UPDATE account
    -> SET avail_balance = avail_balance + 100
    -> WHERE product_cd IN ('CHK', 'SAV', 'MM')
    -> ORDER BY open_date ASC
    -> LIMIT 10;
Query OK, 10 rows affected (0.06 sec)
Rows matched: 10  Changed: 10  Warnings: 0
```

Это выражение сортирует счета в возрастающем порядке по дате открытия и затем изменяет первые десять записей, которыми в данном случае являются десять самых старых счетов.

Многотабличные обновления и удаления

В определенных ситуациях для выполнения поставленной задачи может понадобиться изменить или удалить данные из нескольких разных таблиц. Например, если обнаруживается, что в БД банка есть фиктивный клиент, выявленный в процессе аудита системы, вероятно, понадобится удалить данные из таблиц `account`, `customer` и `individual`.

Для этого раздела я создам набор клонов таблиц `account`, `customer` и `individual`, назвав их `account2`, `customer2` и `individual2`. Это позволит как защитить используемые в примере данные от изменений, так и избежать проблем с ограничениями внешнего ключа между таблицами (более подробно об этом в данном разделе позже). Вот выражения `create table` для формирования трех таблиц-клонов:

```
CREATE TABLE individual2 AS
SELECT * FROM individual;

CREATE TABLE customer2 AS
SELECT * FROM customer;

CREATE TABLE account2 AS
SELECT * FROM account;
```

Если бы ID фиктивного клиента был равен 1, можно было бы сгенерировать три разных выражения `delete` для каждой из трех таблиц:

```
DELETE FROM account2
WHERE cust_id = 1;

DELETE FROM customer2
WHERE cust_id = 1;

DELETE FROM individual2
WHERE cust_id = 1;
```

Но в MySQL можно не писать отдельные выражения `delete`, а создать одно *многотабличное* выражение `delete`, которое в данном случае выглядит так:

```
mysql> DELETE account2, customer2, individual2
    -> FROM account2 INNER JOIN customer2
    ->   ON account2.cust_id = customer2.cust_id
    ->   INNER JOIN individual2
    ->   ON customer2.cust_id = individual2.cust_id
    -> WHERE individual2.cust_id = 1;
Query OK, 5 rows affected (0.02 sec)
```

Это выражение удаляет все пять строк, по одной из таблиц `individual2` и `customer2` и три из таблицы `account2` (у клиента с ID = 1 три счета). В этом выражении три отдельных блока:

`delete`

Указывает таблицы, строки которых предназначенны для удаления.

`from`

Указывает таблицы, позволяющие идентифицировать строки, которые должны быть удалены. Этот блок по форме и выполняемым функциям аналогичен блоку `from` в выражении `select`; в блок `delete` необязательно включать все перечисленные здесь таблицы.

`where`

Содержит условия фильтрации, используемые для идентификации строк, которые должны быть удалены.

Многотабличное выражение `delete` очень похоже на выражение `select`, но с блоком `delete` вместо блока `select`. При удалении строк из одной таблицы с помощью многотабличного `delete` разница еще менее заметна. Например, вот выражение `select`, выбирающее ID всех счетов, принадлежащих Джону Хейварду (John Hayward):

```
mysql> SELECT account2.account_id
    -> FROM account2 INNER JOIN customer2
    ->   ON account2.cust_id = customer2.cust_id
    ->   INNER JOIN individual2
    ->   ON individual2.cust_id = customer2.cust_id
    -> WHERE individual2.fname = 'John'
    ->   AND individual2.lname = 'Hayward';
+------------+
| account_id |
+------------+
|          8 |
|          9 |
|         10 |
+------------+
3 rows in set (0.01 sec)
```

Если просмотрев результаты, вы решите удалить из таблицы `account2` все три счета Джона, потребуется только заменить в предыдущем запросе блок `select` блоком `delete` с указанием таблицы `account2`:

```
mysql> DELETE account2
    -> FROM account2 INNER JOIN customer2
    ->   ON account2.cust_id = customer2.cust_id
    ->   INNER JOIN individual2
    ->   ON customer2.cust_id = individual2.cust_id
    -> WHERE individual2.fname = 'John'
    ->   AND individual2.lname = 'Hayward';
Query OK, 3 rows affected (0.01 sec)
```

Надеюсь, это помогло лучше понять назначение блоков `delete` и `from` в многотабличном выражении `delete`. Оно функционально идентично следующему однотабличному выражению `delete`, определяющему ID клиента Джона Хейварда с помощью подзапроса:

```
DELETE FROM account2
WHERE cust_id =
```

```
 (SELECT cust_id
  FROM individual2
  WHERE fname = 'John' AND lname = 'Hayward';
```

Применяя многотабличное выражение `delete` для удаления строк из одной таблицы, вы просто выбираете подобный запросу формат с соединением таблиц, а не традиционное выражение `delete` с подзапросами. Реальная мощь многотабличных выражений `delete` заключается в возможности удаления данных из нескольких таблиц одним выражением, как показано в первом выражении этого раздела.

Кроме удаления строк из нескольких таблиц, MySQL также предоставляет возможность *изменять* строки в нескольких таблицах с помощью *многотабличного обновления* (*multitable update*). Скажем, происходит слияние двух банков. В базах данных обоих банков есть перекрывающиеся ID клиентов. Руководство одного из банков решает уладить проблему путем добавления 10 000 к каждому ID клиента своего банка, чтобы можно было безопасно импортировать данные второго банка. Следующий пример показывает, как с помощью одного выражения изменить ID клиента с ID 3 в таблицах `individual2`, `customer2` и `account2`:

```
mysql> UPDATE individual2 INNER JOIN customer2
    ->   ON individual2.cust_id = customer2.cust_id
    ->   INNER JOIN account2
    ->   ON customer2.cust_id = account2.cust_id
    -> SET individual2.cust_id = individual2.cust_id + 10000,
    ->   customer2.cust_id = customer2.cust_id + 10000,
    ->   account2.cust_id = account2.cust_id + 10000
    -> WHERE individual2.cust_id = 3;
Query OK, 4 rows affected (0.01 sec)
Rows matched: 5  Changed: 4  Warnings: 0
```

Это выражение изменяет четыре строки – по одной в таблицах `individual2` и `customer2` и две в таблице `account2`. Синтаксис многотабличного выражения `update` очень похож на синтаксис однотабличного выражения `update`, за исключением того, что в блоке `update` указываются несколько таблиц и соответствующие им условия соединения, а не просто одна таблица. Как и однотабличное выражение `update`, многотабличная версия включает блок `set`. Разница в том, что все упомянутые в блоке `update` таблицы можно изменить посредством блока `set`.

При использовании механизма хранения InnoDB, если задействованные таблицы имеют ограничения внешнего ключа, применять многотабличные выражения `delete` и `update` скорее всего не получится. Причина в том, что этот механизм не гарантирует проведение изменений в порядке, не нарушающем ограничения. Поэтому в такой ситуации следует использовать несколько однотабличных выражений в соответствующем порядке, так чтобы не нарушались ограничения внешнего ключа.

C

Решения к упражнениям

Глава 3

3.1

Извлеките ID, имя и фамилию всех банковских сотрудников. Выполните сортировку по фамилии и затем по имени.

```
mysql> SELECT emp_id, fname, lname
    -> FROM employee
    -> ORDER BY lname, fname;
+--------+----------+-----------+
| emp_id | fname    | lname     |
+--------+----------+-----------+
|      2 | Susan    | Barker    |
|     13 | John     | Blake     |
|      6 | Helen    | Fleming   |
|     17 | Beth     | Fowler    |
|      5 | John     | Gooding   |
|      9 | Jane     | Grossman  |
|      4 | Susan    | Hawthorne |
|     12 | Samantha | Jameson   |
|     16 | Theresa  | Markham   |
|     14 | Cindy    | Mason     |
|      8 | Sarah    | Parker    |
|     15 | Frank    | Portman   |
|     10 | Paula    | Roberts   |
|      1 | Michael  | Smith     |
|      7 | Chris    | Tucker    |
|     18 | Rick     | Tulman    |
|      3 | Robert   | Tyler     |
|     11 | Thomas   | Ziegler   |
+--------+----------+-----------+
18 rows in set (0.01 sec)
```

3.2

Извлеките ID счета, ID клиента и доступный остаток всех счетов, имеющих статус 'ACTIVE' (активный) и доступный остаток больше 2500 долларов.

```
mysql> SELECT account_id, cust_id, avail_balance
    -> FROM account
    -> WHERE status = 'ACTIVE'
    ->   AND avail_balance > 2500;
+------------+---------+---------------+
| account_id | cust_id | avail_balance |
+------------+---------+---------------+
|          3 |       1 |       3000.00 |
|         10 |       4 |       5487.09 |
|         13 |       6 |      10000.00 |
|         14 |       7 |       5000.00 |
|         15 |       8 |       3487.19 |
|         18 |       9 |       9345.55 |
|         20 |      10 |      23575.12 |
|         22 |      11 |       9345.55 |
|         23 |      12 |      38552.05 |
|         24 |      13 |      50000.00 |
+------------+---------+---------------+
10 rows in set (0.00 sec)
```

3.3

Напишите запрос к таблице account, возвращающий ID сотрудников, открывших счета (используйте столбец account.open_emp_id). Результирующий набор должен включать по одной строке для каждого сотрудника.

```
mysql> SELECT DISTINCT open_emp_id
    -> FROM account;
+-------------+
| open_emp_id |
+-------------+
|           1 |
|          10 |
|          13 |
|          16 |
+-------------+
4 rows in set (0.00 sec)
```

3.4

В этом запросе к нескольким наборам данных заполните пробелы (обозначенные как *<число>*) так, чтобы получить результат, приведенный ниже:

```
mysql> SELECT p.product_cd, a.cust_id, a.avail_balance
    -> FROM product p INNER JOIN account <1>
```

```
    ->   ON p.product_cd = <2>
    -> WHERE p.<3> = 'ACCOUNT';
+------------+---------+---------------+
| product_cd | cust_id | avail_balance |
+------------+---------+---------------+
| CD         |       1 |       3000.00 |
| CD         |       6 |      10000.00 |
| CD         |       7 |       5000.00 |
| CD         |       9 |       1500.00 |
| CHK        |       1 |       1057.75 |
| CHK        |       2 |       2258.02 |
| CHK        |       3 |       1057.75 |
| CHK        |       4 |        534.12 |
| CHK        |       5 |       2237.97 |
| CHK        |       6 |        122.37 |
| CHK        |       8 |       3487.19 |
| CHK        |       9 |        125.67 |
| CHK        |      10 |      23575.12 |
| CHK        |      12 |      38552.05 |
| MM         |       3 |       2212.50 |
| MM         |       4 |       5487.09 |
| MM         |       9 |       9345.55 |
| SAV        |       1 |        500.00 |
| SAV        |       2 |        200.00 |
| SAV        |       4 |        767.77 |
| SAV        |       8 |        387.99 |
+------------+---------+---------------+
21 rows in set (0.02 sec)
```

Верные значения для <1>, <2> и <3>:

1. a
2. a.product_cd
3. product_type_cd

Глава 4

4.1

Какие ID транзакций возвращают следующие условия фильтрации?

```
txn_date < '2005-02-26' AND (txn_type_cd = 'DBT' OR amount > 100)
```

ID транзакций 1, 2, 3, 5, 6 и 7.

4.2

Какие ID транзакций возвращают следующие условия фильтрации?

```
account_id IN (101,103) AND NOT (txn_type_cd = 'DBT' OR amount > 100)
```

ID транзакций 4 и 9.

4.3

Создайте запрос, выбирающий все счета, открытые в 2002 году.

```
mysql> SELECT account_id, open_date
    -> FROM account
    -> WHERE open_date BETWEEN '2002-01-01' AND '2002-12-31';
+------------+------------+
| account_id | open_date  |
+------------+------------+
|          6 | 2002-11-23 |
|          7 | 2002-12-15 |
|         12 | 2002-08-24 |
|         20 | 2002-09-30 |
|         21 | 2002-10-01 |
+------------+------------+
5 rows in set (0.01 sec)
```

4.4

Создайте запрос, выбирающий всех клиентов-физических лиц, второй буквой фамилии которых является буква `'a'` и есть `'e'` в любой позиции после `'a'`.

```
mysql> SELECT cust_id, lname, fname
    -> FROM individual
    -> WHERE lname LIKE '_a%e%';
+---------+--------+---------+
| cust_id | lname  | fname   |
+---------+--------+---------+
|       1 | Hadley | James   |
|       9 | Farley | Richard |
+---------+--------+---------+
2 rows in set (0.02 sec)
```

Глава 5

5.1

Заполните в следующем запросе пробелы (обозначенные как *<число>*), чтобы получить такие результаты:

```
mysql> SELECT e.emp_id, e.fname, e.lname, b.name
    -> FROM employee e INNER JOIN <1> b
    ->   ON e.assigned_branch_id = b.<2>;
+--------+----------+-----------+---------------+
| emp_id | fname    | lname     | name          |
+--------+----------+-----------+---------------+
|      1 | Michael  | Smith     | Headquarters  |
|      2 | Susan    | Barker    | Headquarters  |
|      3 | Robert   | Tyler     | Headquarters  |
```

```
|      4 | Susan    | Hawthorne | Headquarters  |
|      5 | John     | Gooding   | Headquarters  |
|      6 | Helen    | Fleming   | Headquarters  |
|      7 | Chris    | Tucker    | Headquarters  |
|      8 | Sarah    | Parker    | Headquarters  |
|      9 | Jane     | Grossman  | Headquarters  |
|     10 | Paula    | Roberts   | Woburn Branch |
|     11 | Thomas   | Ziegler   | Woburn Branch |
|     12 | Samantha | Jameson   | Woburn Branch |
|     13 | John     | Blake     | Quincy Branch |
|     14 | Cindy    | Mason     | Quincy Branch |
|     15 | Frank    | Portman   | Quincy Branch |
|     16 | Theresa  | Markham   | So. NH Branch |
|     17 | Beth     | Fowler    | So. NH Branch |
|     18 | Rick     | Tulman    | So. NH Branch |
+--------+----------+-----------+---------------+
18 rows in set (0.03 sec)
```

Верные значения для <1> **и** <2>**:**

1. `branch`
2. `branch_id`

5.2

Напишите запрос, по которому для каждого клиента-физического лица (`customer.cust_type_cd = 'I'`**) возвращаются ID счета, федеральный ID (**`customer.fed_id`**) и тип созданного счета (**`product.name`**).**

```
mysql> SELECT a.account_id, c.fed_id, p.name
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id
    ->   INNER JOIN product p
    ->   ON a.product_cd = p.product_cd
    -> WHERE c.cust_type_cd = 'I';
+------------+-------------+------------------------+
| account_id | fed_id      | name                   |
+------------+-------------+------------------------+
|          1 | 111-11-1111 | checking account       |
|          2 | 111-11-1111 | savings account        |
|          3 | 111-11-1111 | certificate of deposit |
|          4 | 222-22-2222 | checking account       |
|          5 | 222-22-2222 | savings account        |
|          6 | 333-33-3333 | checking account       |
|          7 | 333-33-3333 | money market account   |
|          8 | 444-44-4444 | checking account       |
|          9 | 444-44-4444 | savings account        |
|         10 | 444-44-4444 | money market account   |
|         11 | 555-55-5555 | checking account       |
|         12 | 666-66-6666 | checking account       |
|         13 | 666-66-6666 | certificate of deposit |
|         14 | 777-77-7777 | certificate of deposit |
```

```
|         15 | 888-88-8888 | checking account       |
|         16 | 888-88-8888 | savings account        |
|         17 | 999-99-9999 | checking account       |
|         18 | 999-99-9999 | money market account   |
|         19 | 999-99-9999 | certificate of deposit |
+------------+-------------+------------------------+
19 rows in set (0.00 sec)
```

5.3

Создайте запрос для выбора всех сотрудников, начальник которых приписан к другому отделу. Извлечь ID, имя и фамилию сотрудника.

```
mysql> SELECT e.emp_id, e.fname, e.lname
    -> FROM employee e INNER JOIN employee mgr
    ->   ON e.superior_emp_id = mgr.emp_id
    -> WHERE e.dept_id != mgr.dept_id;
+--------+-------+-----------+
| emp_id | fname | lname     |
+--------+-------+-----------+
|      4 | Susan | Hawthorne |
|      5 | John  | Gooding   |
+--------+-------+-----------+
2 rows in set (0.00 sec)
```

Глава 6

6.1

Имеются множество A = {L M N O P} и множество B = {P Q R S T}. Какие множества будут получены в результате следующих операций:

- A union B = {L M N O P Q R S T}
- A union all B = {L M N O P P Q R S T}
- A intersect B = {P}
- A except B = {L M N O}

6.2

Напишите составной запрос для выбора имен и фамилий всех клиентов-физических лиц, а также имен и фамилий всех сотрудников.

```
mysql> SELECT fname, lname
    -> FROM individual
    -> UNION
    -> SELECT fname, lname
    -> FROM employee;
+----------+-----------+
| fname    | lname     |
+----------+-----------+
```

```
| James    | Hadley    |
| Susan    | Tingley   |
| Frank    | Tucker    |
| John     | Hayward   |
| Charles  | Frasier   |
| John     | Spencer   |
| Margaret | Young     |
| Louis    | Blake     |
| Richard  | Farley    |
| Michael  | Smith     |
| Susan    | Barker    |
| Robert   | Tyler     |
| Susan    | Hawthorne |
| John     | Gooding   |
| Helen    | Fleming   |
| Chris    | Tucker    |
| Sarah    | Parker    |
| Jane     | Grossman  |
| Paula    | Roberts   |
| Thomas   | Ziegler   |
| Samantha | Jameson   |
| John     | Blake     |
| Cindy    | Mason     |
| Frank    | Portman   |
| Theresa  | Markham   |
| Beth     | Fowler    |
| Rick     | Tulman    |
+----------+-----------+
27 rows in set (0.01 sec)
```

6.3

Отсортируйте результаты упражнения 6.2 по столбцу `lname`.

```
mysql> SELECT fname, lname
    -> FROM individual
    -> UNION ALL
    -> SELECT fname, name
    -> FROM employee
    -> ORDER BY lname;
+----------+-----------+
| fname    | lname     |
+----------+-----------+
| Susan    | Barker    |
| Louis    | Blake     |
| John     | Blake     |
| Richard  | Farley    |
| Helen    | Fleming   |
| Beth     | Fowler    |
| Charles  | Frasier   |
| John     | Gooding   |
```

```
| Jane     | Grossman  |
| James    | Hadley    |
| Susan    | Hawthorne |
| John     | Hayward   |
| Samantha | Jameson   |
| Theresa  | Markham   |
| Cindy    | Mason     |
| Sarah    | Parker    |
| Frank    | Portman   |
| Paula    | Roberts   |
| Michael  | Smith     |
| John     | Spencer   |
| Susan    | Tingley   |
| Chris    | Tucker    |
| Frank    | Tucker    |
| Rick     | Tulman    |
| Robert   | Tyler     |
| Margaret | Young     |
| Thomas   | Ziegler   |
+----------+-----------+
27 rows in set (0.01 sec)
```

Глава 7

7.1

Написать запрос, возвращающий 17–25 символы строки «Please find the substring in this string» (Пожалуйста, найдите подстроку в этой строке).

```
mysql> SELECT SUBSTRING('Please find the substring in this string',17,9);
+-----------------------------------------------------------+
| SUBSTRING('Please find the substring in this string',17,9) |
+-----------------------------------------------------------+
| substring                                                 |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

7.2

Напишите запрос, возвращающий абсолютную величину и знак (-1, 0 или 1) числа -25,768 23. Также возвратите число, округленное до сотых.

```
mysql> SELECT ABS(-25.76823), SIGN(-25.76823), ROUND(-25.76823, 2);
+----------------+-----------------+---------------------+
| ABS(-25.76823) | SIGN(-25.76823) | ROUND(-25.76823, 2) |
+----------------+-----------------+---------------------+
|       25.76823 |              -1 |              -25.77 |
+----------------+-----------------+---------------------+
1 row in set (0.00 sec)
```

7.3

Напишите запрос, возвращающий только значение месяца текущей даты.

```
mysql> SELECT EXTRACT(MONTH FROM CURRENT_DATE(  ));
+-----------------------------------+
| EXTRACT(MONTH FROM CURRENT_DATE) |
+-----------------------------------+
|                                 5 |
+-----------------------------------+
1 row in set (0.02 sec)
```

(Если это упражнение выполняется не в мае, полученный результат будет отличаться от приведенного.)

Глава 8

8.1

Создайте запрос для подсчета числа строк в таблице `account`.

```
mysql> SELECT COUNT(*)
    -> FROM account;
+----------+
| count(*) |
+----------+
|       24 |
+----------+
1 row in set (0.32 sec)
```

8.2

Измените свой запрос из упражнения 8.1 для подсчета числа счетов, имеющихся у каждого клиента. Для каждого клиента выведите ID клиента и количество счетов.

```
mysql> SELECT cust_id, COUNT(*)
    -> FROM account
    -> GROUP BY cust_id;
+---------+----------+
| cust_id | count(*) |
+---------+----------+
|       1 |        3 |
|       2 |        2 |
|       3 |        2 |
|       4 |        3 |
|       5 |        1 |
|       6 |        2 |
|       7 |        1 |
|       8 |        2 |
|       9 |        3 |
|      10 |        2 |
```

```
|      11 |        1 |
|      12 |        1 |
|      13 |        1 |
+---------+----------+
13 rows in set (0.00 sec)
```

8.3

Измените запрос из упражнения 8.2 так, чтобы в результирующий набор были включены только клиенты, имеющие не меньше двух счетов.

```
mysql> SELECT cust_id, COUNT(*)
    -> FROM account
    -> GROUP BY cust_id
    -> HAVING COUNT(*) >= 2;
+---------+----------+
| cust_id | COUNT(*) |
+---------+----------+
|       1 |        3 |
|       2 |        2 |
|       3 |        2 |
|       4 |        3 |
|       6 |        2 |
|       8 |        2 |
|       9 |        3 |
|      10 |        2 |
+---------+----------+
8 rows in set (0.04 sec)
```

8.4 (дополнительно)

Найдите общий доступный остаток по типу счета и отделению, где на каждый тип и отделение приходится более одного счета. Результаты должны быть упорядочены по общему остатку (от наибольшего к наименьшему).

```
mysql> SELECT product_cd, open_branch_id, SUM(avail_balance)
    -> FROM account
    -> GROUP BY product_cd, open_branch_id
    -> HAVING COUNT(*) > 1
    -> ORDER BY 3 DESC;
+------------+----------------+--------------------+
| product_cd | open_branch_id | SUM(avail_balance) |
+------------+----------------+--------------------+
| CHK        |              4 |           67852.33 |
| MM         |              1 |           14832.64 |
| CD         |              1 |           11500.00 |
| CD         |              2 |            8000.00 |
| CHK        |              2 |            3315.77 |
| CHK        |              1 |             782.16 |
| SAV        |              2 |             700.00 |
+------------+----------------+--------------------+
7 rows in set (0.01 sec)
```

Примечание: MySQL не принимает `ORDER BY SUM(avail_balance) DESC,`, поэтому я был вынужден обозначить столбец сортировки его порядковым номером.

Глава 9

9.1

Создайте запрос к таблице `account`, использующий условие фильтрации с несвязанным подзапросом к таблице `product` для поиска всех кредитных счетов (`product.product_type_cd = 'LOAN'`). Должны быть выбраны ID счета, код счета, ID клиента и доступный остаток.

```
mysql> SELECT account_id, product_cd, cust_id, avail_balance
    -> FROM account
    -> WHERE product_cd IN (SELECT product_cd
    ->   FROM product
    ->   WHERE product_type_cd = 'LOAN');
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|         21 | BUS        |      10 |          0.00 |
|         22 | BUS        |      11 |       9345.55 |
|         24 | SBL        |      13 |      50000.00 |
+------------+------------+---------+---------------+
3 rows in set (0.07 sec)
```

9.2

Переработайте запрос из упражнения 9.1, используя *связанный* подзапрос к таблице `product` для получения того же результата.

```
mysql> SELECT a.account_id, a.product_cd, a.cust_id, a.avail_balance
    -> FROM account a
    -> WHERE EXISTS (SELECT 1
    ->   FROM product p
    ->   WHERE p.product_cd = a.product_cd
    ->     AND p.product_type_cd = 'LOAN');
+------------+------------+---------+---------------+
| account_id | product_cd | cust_id | avail_balance |
+------------+------------+---------+---------------+
|         21 | BUS        |      10 |          0.00 |
|         22 | BUS        |      11 |       9345.55 |
|         24 | SBL        |      13 |      50000.00 |
+------------+------------+---------+---------------+
3 rows in set (0.01 sec)
```

9.3

Соедините следующий запрос с таблицей `employee`, чтобы показать уровень квалификации каждого сотрудника:

```
SELECT 'trainee' name, '2004-01-01' start_dt, '2005-12-31' end_dt
UNION ALL
SELECT 'worker' name, '2002-01-01' start_dt, '2003-12-31' end_dt
UNION ALL
SELECT 'mentor' name, '2000-01-01' start_dt, '2001-12-31' end_dt
```

Дайте подзапросу псевдоним `levels` (уровни) и включите ID сотрудника, имя, фамилию и квалификацию (`levels.name`). (Совет: в условии соединения определяйте диапазон, в который попадает столбец `employee.start_date`, с помощью условия неравенства.)

```
mysql> SELECT e.emp_id, e.fname, e.lname, levels.name
    -> FROM employee e INNER JOIN
    ->  (SELECT 'trainee' name, '2004-01-01' start_dt, '2005-12-31' end_dt
    ->   UNION ALL
    ->   SELECT 'worker' name, '2002-01-01' start_dt, '2003-12-31' end_dt
    ->   UNION ALL
    ->  SELECT 'mentor' name, '2000-01-01' start_dt, '2001-12-31' end_dt) levels
    ->   ON e.start_date BETWEEN levels.start_dt AND levels.end_dt;
+--------+----------+-----------+---------+
| emp_id | fname    | lname     | name    |
+--------+----------+-----------+---------+
|      6 | Helen    | Fleming   | trainee |
|      7 | Chris    | Tucker    | trainee |
|      2 | Susan    | Barker    | worker  |
|      4 | Susan    | Hawthorne | worker  |
|      5 | John     | Gooding   | worker  |
|      8 | Sarah    | Parker    | worker  |
|      9 | Jane     | Grossman  | worker  |
|     10 | Paula    | Roberts   | worker  |
|     12 | Samantha | Jameson   | worker  |
|     14 | Cindy    | Mason     | worker  |
|     15 | Frank    | Portman   | worker  |
|     17 | Beth     | Fowler    | worker  |
|     18 | Rick     | Tulman    | worker  |
|      1 | Michael  | Smith     | mentor  |
|      3 | Robert   | Tyler     | mentor  |
|     11 | Thomas   | Ziegler   | mentor  |
|     13 | John     | Blake     | mentor  |
|     16 | Theresa  | Markham   | mentor  |
+--------+----------+-----------+---------+
18 rows in set (0.00 sec)
```

9.4

Создайте запрос к таблице `employee` для получения ID, имени и фамилии сотрудника вместе с названиями отдела и отделения, к которым он приписан. Не используйте соединение таблиц.

```
mysql> SELECT e.emp_id, e.fname, e.lname,
    ->  (SELECT d.name FROM department d
    ->   WHERE d.dept_id = e.dept_id) dept_name,
```

```
    ->  (SELECT b.name FROM branch b
    ->   WHERE b. branch_id = e.assigned_branch_id) branch_name
    -> FROM employee e;
+--------+----------+-----------+----------------+---------------+
| emp_id | fname    | lname     | dept_name      | branch_name   |
+--------+----------+-----------+----------------+---------------+
|      1 | Michael  | Smith     | Administration | Headquarters  |
|      2 | Susan    | Barker    | Administration | Headquarters  |
|      3 | Robert   | Tyler     | Administration | Headquarters  |
|      4 | Susan    | Hawthorne | Operations     | Headquarters  |
|      5 | John     | Gooding   | Loans          | Headquarters  |
|      6 | Helen    | Fleming   | Operations     | Headquarters  |
|      7 | Chris    | Tucker    | Operations     | Headquarters  |
|      8 | Sarah    | Parker    | Operations     | Headquarters  |
|      9 | Jane     | Grossman  | Operations     | Headquarters  |
|     10 | Paula    | Roberts   | Operations     | Woburn Branch |
|     11 | Thomas   | Ziegler   | Operations     | Woburn Branch |
|     12 | Samantha | Jameson   | Operations     | Woburn Branch |
|     13 | John     | Blake     | Operations     | Quincy Branch |
|     14 | Cindy    | Mason     | Operations     | Quincy Branch |
|     15 | Frank    | Portman   | Operations     | Quincy Branch |
|     16 | Theresa  | Markham   | Operations     | So. NH Branch |
|     17 | Beth     | Fowler    | Operations     | So. NH Branch |
|     18 | Rick     | Tulman    | Operations     | So. NH Branch |
+--------+----------+-----------+----------------+---------------+
18 rows in set (0.12 sec)
```

Глава 10

10.1

Напишите запрос, возвращающий все типы счетов и открытые счета этих типов (для соединения с таблицей `product` используйте столбец `product_cd` таблицы `account`). Должны быть включены все типы счетов, даже если не был открыт ни один счет определенного типа.

```
mysql> SELECT p.product_cd, a.account_id, a.cust_id, a.avail_balance
    -> FROM product p LEFT OUTER JOIN account a
    ->   ON p.product_cd = a.product_cd;
+------------+------------+---------+---------------+
| product_cd | account_id | cust_id | avail_balance |
+------------+------------+---------+---------------+
| AUT        |       NULL |    NULL |          NULL |
| BUS        |         21 |      10 |          0.00 |
| BUS        |         22 |      11 |       9345.55 |
| CD         |          3 |       1 |       3000.00 |
| CD         |         13 |       6 |      10000.00 |
| CD         |         14 |       7 |       5000.00 |
| CD         |         19 |       9 |       1500.00 |
| CHK        |          1 |       1 |       1057.75 |
```

```
| CHK        |          4 |       2 |       2258.02 |
| CHK        |          6 |       3 |       1057.75 |
| CHK        |          8 |       4 |        534.12 |
| CHK        |         11 |       5 |       2237.97 |
| CHK        |         12 |       6 |        122.37 |
| CHK        |         15 |       8 |       3487.19 |
| CHK        |         17 |       9 |        125.67 |
| CHK        |         20 |      10 |      23575.12 |
| CHK        |         23 |      12 |      38552.05 |
| MM         |          7 |       3 |       2212.50 |
| MM         |         10 |       4 |       5487.09 |
| MM         |         18 |       9 |       9345.55 |
| MRT        |       NULL |    NULL |          NULL |
| SAV        |          2 |       1 |        500.00 |
| SAV        |          5 |       2 |        200.00 |
| SAV        |          9 |       4 |        767.77 |
| SAV        |         16 |       8 |        387.99 |
| SBL        |         24 |      13 |      50000.00 |
+------------+------------+---------+---------------+
26 rows in set (0.01 sec)
```

10.2

Переформулируйте запрос из упражнения 10.1 и примените другой тип внешнего соединения (т. е. если в упражнении 10.1 использовалось левостороннее внешнее соединение, используйте правостороннее), так чтобы результаты были, как в упражнении 10.1.

```
mysql> SELECT p.product_cd, a.account_id, a.cust_id, a.avail_balance
    -> FROM account a RIGHT OUTER JOIN product p
    ->   ON p.product_cd = a.product_cd;
+------------+------------+---------+---------------+
| product_cd | account_id | cust_id | avail_balance |
+------------+------------+---------+---------------+
| AUT        |       NULL |    NULL |          NULL |
| BUS        |         21 |      10 |          0.00 |
| BUS        |         22 |      11 |       9345.55 |
| CD         |          3 |       1 |       3000.00 |
| CD         |         13 |       6 |      10000.00 |
| CD         |         14 |       7 |       5000.00 |
| CD         |         19 |       9 |       1500.00 |
| CHK        |          1 |       1 |       1057.75 |
| CHK        |          4 |       2 |       2258.02 |
| CHK        |          6 |       3 |       1057.75 |
| CHK        |          8 |       4 |        534.12 |
| CHK        |         11 |       5 |       2237.97 |
| CHK        |         12 |       6 |        122.37 |
| CHK        |         15 |       8 |       3487.19 |
| CHK        |         17 |       9 |        125.67 |
| CHK        |         20 |      10 |      23575.12 |
| CHK        |         23 |      12 |      38552.05 |
```

```
| MM         |           7 |       3 |       2212.50 |
| MM         |          10 |       4 |       5487.09 |
| MM         |          18 |       9 |       9345.55 |
| MRT        |        NULL |    NULL |          NULL |
| SAV        |           2 |       1 |        500.00 |
| SAV        |           5 |       2 |        200.00 |
| SAV        |           9 |       4 |        767.77 |
| SAV        |          16 |       8 |        387.99 |
| SBL        |          24 |      13 |      50000.00 |
+------------+-------------+---------+---------------+
26 rows in set (0.02 sec)
```

10.3

Проведите внешнее соединение таблицы account **с таблицами** individual **и** business **(посредством столбца** account.cust_id**) таким образом, чтобы результирующий набор содержал по одной строке для каждого счета. Должны быть включены столбцы** count.account_id**,** account.product_cd**,** individual.fname**,** individual.lname **и** business.name**.**

```
mysql> SELECT a.account_id, a.product_cd,
    ->   i.fname, i.lname, b.name
    -> FROM account a LEFT OUTER JOIN business b
    ->   ON a.cust_id = b.cust_id
    ->   LEFT OUTER JOIN individual i
    ->   ON a.cust_id = i.cust_id;
+------------+------------+----------+---------+-----------------------+
| account_id | product_cd | fname    | lname   | name                  |
+------------+------------+----------+---------+-----------------------+
|          1 | CHK        | James    | Hadley  | NULL                  |
|          2 | SAV        | James    | Hadley  | NULL                  |
|          3 | CD         | James    | Hadley  | NULL                  |
|          4 | CHK        | Susan    | Tingley | NULL                  |
|          5 | SAV        | Susan    | Tingley | NULL                  |
|          6 | CHK        | Frank    | Tucker  | NULL                  |
|          7 | MM         | Frank    | Tucker  | NULL                  |
|          8 | CHK        | John     | Hayward | NULL                  |
|          9 | SAV        | John     | Hayward | NULL                  |
|         10 | MM         | John     | Hayward | NULL                  |
|         11 | CHK        | Charles  | Frasier | NULL                  |
|         12 | CHK        | John     | Spencer | NULL                  |
|         13 | CD         | John     | Spencer | NULL                  |
|         14 | CD         | Margaret | Young   | NULL                  |
|         15 | CHK        | Louis    | Blake   | NULL                  |
|         16 | SAV        | Louis    | Blake   | NULL                  |
|         17 | CHK        | Richard  | Farley  | NULL                  |
|         18 | MM         | Richard  | Farley  | NULL                  |
|         19 | CD         | Richard  | Farley  | NULL                  |
|         20 | CHK        | NULL     | NULL    | Chilton Engineering   |
|         21 | BUS        | NULL     | NULL    | Chilton Engineering   |
|         22 | BUS        | NULL     | NULL    | Northeast Cooling Inc. |
```

```
|          23 | CHK         | NULL      | NULL     | Superior Auto Body     |
|          24 | SBL         | NULL      | NULL     | AAA Insurance Inc.     |
+-------------+-------------+-----------+----------+------------------------+
24 rows in set (0.05 sec)
```

10.4 (дополнительно)

Разработайте запрос, который сформирует набор {1, 2, 3,..., 99, 100}. (Совет: используйте перекрестное соединение как минимум с двумя подзапросами в блоке `from`.)

```
SELECT ones.x + tens.x + 1
FROM
 (SELECT 0 x UNION ALL
  SELECT 1 x UNION ALL
  SELECT 2 x UNION ALL
  SELECT 3 x UNION ALL
  SELECT 4 x UNION ALL
  SELECT 5 x UNION ALL
  SELECT 6 x UNION ALL
  SELECT 7 x UNION ALL
  SELECT 8 x UNION ALL
  SELECT 9 x) ones
CROSS JOIN
(SELECT 0 x UNION ALL
  SELECT 10 x UNION ALL
  SELECT 20 x UNION ALL
  SELECT 30 x UNION ALL
  SELECT 40 x UNION ALL
  SELECT 50 x UNION ALL
  SELECT 60 x UNION ALL
  SELECT 70 x UNION ALL
  SELECT 80 x UNION ALL
  SELECT 90 x) tens;
```

Глава 11

11.1

Перепишите следующий запрос, использующий простое выражение `case`, таким образом, чтобы получить аналогичные результаты с помощью выражения `case` с перебором вариантов. Попытайтесь свести к минимуму количество блоков `when`.

```
SELECT emp_id,
  CASE title
    WHEN 'President' THEN 'Management'
    WHEN 'Vice President' THEN 'Management'
    WHEN 'Treasurer' THEN 'Management'
    WHEN 'Loan Manager' THEN 'Management'
    WHEN 'Operations Manager' THEN 'Operations'
```

```
    WHEN 'Head Teller' THEN 'Operations'
    WHEN 'Teller' THEN 'Operations'
    ELSE 'Unknown'
  END
FROM employee;

SELECT emp_id,
  CASE
    WHEN title LIKE '%President' OR title = 'Loan Manager'
      OR title = 'Treasurer'
      THEN 'Management'
    WHEN title LIKE '%Teller' OR title = 'Operations Manager'
      THEN 'Operations'
    ELSE 'Unknown'
  END
FROM employee;
```

11.2

Перепишите следующий запрос так, чтобы результирующий набор содержал всего одну строку и четыре столбца (по одному для каждого отделения). Назовите столбцы branch_1**,** branch_2 **и т. д.**

```
mysql> SELECT open_branch_id, COUNT(*)
    -> FROM account
    -> GROUP BY open_branch_id;
+----------------+----------+
| open_branch_id | COUNT(*) |
+----------------+----------+
|              1 |        8 |
|              2 |        7 |
|              3 |        3 |
|              4 |        6 |
+----------------+----------+
4 rows in set (0.00 sec)

mysql> SELECT
    ->   SUM(CASE WHEN open_branch_id = 1 THEN 1 ELSE 0 END) branch_1,
    ->   SUM(CASE WHEN open_branch_id = 2 THEN 1 ELSE 0 END) branch_2,
    ->   SUM(CASE WHEN open_branch_id = 3 THEN 1 ELSE 0 END) branch_3,
    ->   SUM(CASE WHEN open_branch_id = 4 THEN 1 ELSE 0 END) branch_4
    -> FROM account;
+----------+----------+----------+----------+
| branch_1 | branch_2 | branch_3 | branch_4 |
+----------+----------+----------+----------+
|        8 |        7 |        3 |        6 |
+----------+----------+----------+----------+
1 row in set (0.02 sec)
```

D

Дополнительные источники

Теперь, прочитав эту книгу, вы должны уверенно продвигаться к профессиональному владению языком SQL. Я набрался смелости и сделал книгу не просто ознакомительной, а более глубокой. Поэтому некоторые из рассмотренных тем могут по-прежнему вызывать определенные трудности. Это хорошо, потому что, по-моему, покупать техническую книгу, которую можно прочитать всего один раз, – все равно что бросать деньги на ветер. Надеюсь, что вы будете перечитывать отдельные главы и продолжать экспериментировать с примером БД до тех пор, пока не сформируете твердое представление об основных концепциях.

Следующий шаг этого путешествия зависит от конкретных целей, которые вы перед собой ставите. За годы работы я встречался с многими людьми и могу ориентировочно разделить читателей на следующие категории:

- Вы программист (или работаете над диссертацией для получения ученой степени по вычислительной технике) без или с небольшим предварительным знанием теории баз данных и хотите расширить свои навыки или вас попросили помочь разобраться с аспектами проекта, связанными с БД.
- Вы непрограммист, но получили задание разработать проект создания отчетов или управления интеллектуальными ресурсами компании, возможно, включая установку и администрирование сервера BI (Business Intelligence – интеллектуальные ресурсы), такого как Business Objects, Actuate, Microstrategy или Cognos.
- Вы системный администратор и хотите распространить свою деятельность и на администрирование БД.
- Вы владелец малого предприятия, которому требуется БД для отслеживания клиентов, материально-производственных запасов, заказов и т. д. или используете «коробочное» приложение и хотите создавать специальные отчеты.

- Вы не относитесь к вышеупомянутым категориям, но каким-то образом на вашем ноутбуке (надеюсь, это не рабочий компьютер) оказалась база данных.

В зависимости о того, какая из указанных категорий лучше всего описывает вашу ситуацию, вас заинтересует одна или несколько следующих тем:

- Углубленное изучение SQL
- Программирование баз данных
- Проектирование баз данных
- Настройка баз данных
- Администрирование баз данных
- Создание отчетов

Далее в этом приложении представлены все эти темы и предложены некоторые источники, помогающие овладеть этими дополнительными навыками.

Углубленное изучение SQL

Независимо от вашей роли в организации, как то администратор, программист, специалист по вопросам производительности, составитель отчетов или даже разработчик БД, для личного успеха и успеха вашего проекта нет более важного навыка, чем владение языком SQL в совершенстве. Все так, но довольно сложно понять, владеете ли вы языком, который включает лишь горстку команд (я имею в виду категорию SQL-выражений для работы с данными, включающую выражения `select`, `insert`, `update` и `delete`). Лучший способ определить, освоили ли вы SQL, – пройти курс углубленного изучения SQL или прочитать книгу для профессионалов. После этого большинство почувствует, что вопрос не настолько хорошо знаком, как это представлялось.

Вот чем вам может помочь совершенное владение языком SQL:

- Программисты, работающие с базами данных, в конечном счете отвечают за соответствующую реализацию спецификаций проекта и за общую производительность системы. Лучшие программисты могут создавать лаконичные и эффективные SQL-выражения, правильно реализующие спецификации проекта, не приводя к проблемам производительности.
- Администрирование БД часто объединяет несколько навыков, включая проектирование и реализацию БД, программирование БД и настройку БД. Чтобы выполнять повседневные задачи администрирования, администраторы должны в совершенстве владеть SQL-выражениями управления схемой данных (такими как `create table` и `alter index`), а превосходное знание SQL-выражений для работы с данными поможет находить наилучшие решения при проектировании, программировании и настройке.

- Специалисты по вопросам производительности, как правило, соглашаются, что большинство проблем производительности обусловлено плохо продуманными SQL-выражениями, а не недостатком ресурсов сервера. Однако при возникновении проблем с производительностью обычно проклинают сервер и затевают дорогостоящие и ненужные обновления оборудования.
- Хорошо спроектированная БД должна быть довольно простой в навигации, иначе те, кто ее реализует, будут делать ошибки. Чем больше вы знаете о том, как будут извлекаться данные из вашей БД, тем лучше будут решения, принимаемые вами на этапе проектирования.

При углубленном изучении SQL вы осмелитесь выйти за рамки базового языка, задействовав специализированные интерфейсы и наборы команд. Например, ваша реализация SQL может позволить генерировать данные в формате XML (Extensible Markup Language – расширяемый язык разметки) прямо из запроса к БД или сохранять, проводить синтаксический анализ и извлекать документы XML. Ваша реализация SQL может также включать специальную функциональность для хранилищ данных и запросов к интеллектуальным ресурсам, такую как возможность ранжирования (например: «покажите мне 10 лучших продавцов прошлого года»). Кроме того, вам может понадобиться взаимодействие с объектно-ориентированными языками программирования, включая применение специальных команд для хранения, извлечения и создания объектов и коллекций объектов. Ни одна из этих тем обычно не рассматривается в ознакомительных книгах по SQL.

Следующие книги помогут перейти на следующий уровень в изучении SQL:

Санжей Мишра (Sanjay Mishra) и Алан Бьюли (Alan Beaulieu) «Mastering Oracle SQL», Second Edition, O'Reilly, 2004.[1]

Поль Дюбуа (Paul DuBois) «MySQL Cookbook», O'Reilly, 2002.[2]

Пол Нильсен (Paul Nielsen) «Microsoft SQL Server 2000 Bible», Wiley, 2002.

Можно также пройти курс обучения в одном из следующих учебных центров:

- Oracle University (*http://education.oracle.com*)
- Learning Tree International (*http://www.learningtree.com*)
- Microsoft Learning (*learning*)
- MySQL Training (*training*)

---

[1] Санжей Мишра, Алан Бьюли «Секреты Oracle SQL», перевод 1-го издания, Символ-Плюс, 2003.

[2] Поль Дюбуа «MySQL. Сборник рецептов», Символ-Плюс, 2004.

Программирование баз данных

Если вы программист, желающий добавить к своему профессиональному багажу умение организовывать доступ к БД, изучение SQL – только один из кусочков этого пазла. Нужен еще язык программирования или API, позволяющий создавать сеансы с БД и взаимодействовать с ней посредством команд SQL. Может, вы уже работаете с языком, обладающим этими возможностями, а может, вам понадобится дополнительный API или драйвер. В табл. D.1 показаны некоторые возможности, доступные в основных языках программирования.

Таблица D.1. Возможности доступа к БД

Язык программирования	API	Описание
Java	Java Database Connectivity (JDBC)	Набор интерфейсов для взаимодействия с БД. Необходим JDBC-драйвер (реализация интерфейсов JDBC) производителя вашей БД или стороннего производителя
C++	Oracle Call Interface (OCI)	Набор библиотек C/C++ для соединения с базой данных Oracle и выполнения команд SQL
	MySQL++	Набор библиотек C++ для соединения с базой данных MySQL и выполнения команд SQL
	RogueWave SourcePro DB	Набор библиотек C++ для соединения с базой данных MySQL, Oracle или SQL Server (в том числе) и выполнения команд SQL
Perl	DBI	Модуль для доступа к MySQL, SQL Server, Oracle и нескольким другим СУБД посредством единого интерфейса
Visual C++ Visual C# Visual Basic	Microsoft ActiveX® Data Objects .NET (ADO.NET)	Набор интерфейсов, предоставляющих сервисы доступа к данным для платформы .NET

Все приведенные в табл. D.1 языки программирования являются языками общего назначения. И всем им для доступа к БД нужен дополнительный драйвер или набор библиотек. Однако есть другие языки, разработанные специально для работы с БД и включающие определенные команды SQL (как минимум `select`, `update`, `insert`, `delete`, `start transaction`, `commit` и `rollback`). В табл. D.2 приведены некоторые из этих языков и описаны среды, в которых они работают.

Таблица D.2. Специализированные языки программирования баз данных

Язык программирования	Разработчик	Среда выполнения
PL/SQL	Oracle	Oracle Database Oracle Application Server
Transact-SQL	Microsoft	SQL Server
SQL2003 Stored Procedure Language	MySQL	MySQL Server (версии 5.0 и выше)
PowerBuilder	Sybase	Sybase EAServer Клиенты для Windows и UNIX
PowerHouse 4GL	Cognos	PowerHouse Application Server
Progress 4GL	Progress Software	OpenEdge Database OpenEdge Application Server

Последние три языка в табл. D.2 – это языки общего назначения, предназначенные для создания бизнес-приложений, а первые три, предоставляемые тремя серверами БД, рассматриваемыми в данной книге, позволяют формировать модули следующих типов:

Хранимые процедуры

Именованные подпрограммы, принимающие параметры.

Хранимые функции

Именованные функции, принимающие параметры и возвращающие значение.

Триггеры

Модули, автоматически запускаемые сервером БД при определенном событии, например удалении данных из той или иной таблицы.

Триггеры выполняются только сервером БД, а хранимые процедуры и функции выполняются в рамках сеанса БД точно так же, как команды SQL. Поскольку хранимые функции возвращают значение, они могут вызываться из SQL-выражений во всех случаях, где может использоваться скалярный подзапрос. Для работы с Oracle Database, SQL Server или MySQL необходимо обязательно ознакомиться с одной из следующих книг:

Мэйдин Фишер (Maydene Fisher) и др. «JDBC API Tutorial and Reference», Third Edition (Учебное пособие и справочник по JDBC API, 3-е издание), Addison-Wesley, 2003.

Билл Гамильтон (Bill Hamilton) «ADO.NET Cookbook» O'Reilly, 2003.[1]

[1] Билл Гамильтон «ADO.NET. Сборник рецептов для профессионалов», СПб: Питер, 2004.

Аллигатор Декарт (Alligator Descartes), Тим Банс (Tim Bunce) «Programming the Perl DBI», O'Reilly, 2000.[1]

Стивен Фейерштейн (Steven Feuerstein) и Билл Прибыл (Bill Pribyl) «Oracle PL/SQL Programming», Third Edition, O'Reilly, 2002.[2]

Кен Хендерсон (Ken Henderson) «The Guru's Guide to Transact-SQL», Addison-Wesley, 2000.[3]

Также есть множество учебных курсов по программированию БД. Их можно найти в одном из следующих учебных центров:

- Oracle University (*http://education.oracle.com*)
- Learning Tree International (*http://www.learningtree.com*)
- Microsoft Learning (*learning*)
- MySQL Training (*training*)

Проектирование БД

Если вы новичок в SQL (а я предполагаю, что это так), то скорее всего будете работать с имеющимися БД, по крайней мере, поначалу. Однако если вы также отвечаете за разработку БД для своего проекта, рекомендую не просто ознакомиться с кратким обзором проектирования и нормализации БД, приведенным в главе 2, а рассмотреть этот вопрос более внимательно. На самом деле есть несколько разновидностей моделей БД, каждая из которых имеет специальное назначение:

Логические модели

Обычно это представление высокого уровня детализации организации и среды, в которой осуществляется деятельность.

Функциональные модели

Обычно это представление среднего уровня детализации отдельного сегмента деятельности организации; как правило, используется в дополнение к спецификации проекта.

Физические модели

Обычно используются для формирования БД.

Администратора БД, возможно, интересуют только физические модели, тогда как логические модели часто являются сферой интересов корпоративных архитекторов (если организации посчастливилось иметь команду разработки корпоративной архитекторы).

---

[1] Аллигатор Декарт и Тим Банс «Программирование на Perl DBI», Символ-Плюс, 2000.

[2] С. Фейерштейн, Б. Прибыл «Oracle PL/SQL для профессионалов», Питер, 2003.

[3] Кен Хендерсон «Профессиональное руководство по Transact-SQL», Питер, 2005.

Как бы то ни было, прежде чем браться за выражения `create table`, необходимо серьезно подумать об использовании инструмента моделирования для построения визуальных моделей. При создании моделей БД обычно используется одна из двух методик:

Моделирование сущностей и связей (Entity-relationship, ER)

Используется практически исключительно для моделирования БД.

Моделирование с использованием Унифицированного языка моделирования (Unified Modeling Language, UML)

Универсальный инструмент моделирования для разработки объектно-ориентированного программного обеспечения.

Если БД проектируется как часть проекта по разработке объектно-ориентированного программного обеспечения, команда разработки может приобрести инструмент моделирования UML для объектного моделирования, чтобы использовать его и для проектирования БД. Если вы вольны в выборе инструментария, более полезным может оказаться один из следующих ER-инструментов, способных формировать полнофункциональные схемы БД (включая таблицы, ограничения, индексы, представления и т. д.) по одному нажатию клавиши:

- ERwin Data Modeler (ER-моделирование)
- Computer Associates (*http://www.ca.com*)
- ER/Studio (ER-моделирование)
- Embarcadero Technologies (*http://www.embarcadero.com*)
- Rational Rose (UML-моделирование)
- IBM (*http://www.ibm.com*)
- Visio (и ER- моделирование, и UML-моделирование)
- Microsoft (*http://www.microsoft.com*)

Ниже приведены две хорошие книги по проектированию БД:

Майкл Дж. Хернандес (Michael J. Hernandez) «Database Design for Mere Mortals: A Hands-on Guide to Relational Database Design», Second Edition (Проектирование баз данных для простых смертных. Практическое руководство по проектированию реляционных баз данных), Addison-Wesley, 2003.

Эрик Дж. Нейбург (Eric J. Naiburg) и Роберт А. Максимчук (Robert A. Maksimchuk) «UML for Database Design», Addison-Wesley, 2001.[1]

[1] Эрик Дж. Нейбург и Роберт А. Максимчук «Проектирование баз данных с помощью UML», Вильямс, 2002.

Настройка баз данных

Настройка БД – это, по сути, искусство и наука выявления и устранения узких мест производительности в:

- Приложениях, организующих доступ к БД (SQL, блокировки, транзакции)
- Схемах БД (проектирование, индексация, сегментирование таблиц)
- Серверах БД (конфигурация серверов, журналирование, управление соединением)
- Дисковых массивах, на которых хранятся файлы БД (конфигурации RAID, определение «горячих» точек)
- Компьютерах, на которых располагаются серверы БД (конфигурация операционной системы, файловые системы)
- Сетях, распространяющих данные на/с серверов БД

Выявление и устранение узких мест в оборудовании и программном обеспечении такой широкой номенклатуры может показаться пугающе сложной задачей, но большая часть работы обычно сосредоточена на схеме БД и языке SQL, который используется приложениями для доступа к БД. Это никак не преуменьшает значимости конфигурирования операционной системы, установки и конфигурирования сервера БД, а также компоновки ресурсов данных в дисковом массиве, но схемы БД и используемый для доступа к ним SQL – гораздо более динамичные компоненты системы, т. е. чреваты большим количеством проблем.

Будь вы штатный специалист по вопросам производительности, программист БД или администратор БД, в круг вашей основной деятельности по настройке войдут:

- Просмотр плана выполнения SQL-выражений для поиска неэффективных моментов
- Разработка стратегий индексации для обеспечения эффективного доступа
- Доработка или переписывание SQL-выражений с целью повлиять на выбор плана выполнения

Как упоминалось в главе 3, каждая БД включает компонент под названием *оптимизатор запросов*, задачей которого является вычисление SQL-выражений и выбор эффективного пути доступа к информационным ресурсам для достижения желаемых результатов. Результат работы оптимизатора – *план выполнения*, показывающий, какие ресурсы в каком порядке используются. Каждая из трех БД, обсуждаемых в книге, включает инструменты получения и просмотра плана выполнения SQL-выражения. Вам понадобится научиться генерировать и расшифровывать планы выполнения для своей БД.

Просто чтобы дать вам понять, о чем идет речь, привожу созданный MySQL план выполнения для запроса, организующего доступ к двум таблицам:

```
mysql> EXPLAIN SELECT c.fed_id, a.account_id, a.avail_balance
    -> FROM account a INNER JOIN customer c
    ->   ON a.cust_id = c.cust_id
    -> WHERE c.cust_type_cd = 'I' \G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: c
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 13
        Extra: Using where
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: a
         type: ref
possible_keys: fk_a_cust_id
          key: fk_a_cust_id
      key_len: 4
          ref: bank.c.cust_id
         rows: 1
        Extra:
2 rows in set (0.00 sec)
```

Чтобы увидеть план выполнения этого запроса, я просто поставил перед выражением `select` ключевое слово `explain` (объяснить), т. е. сервер получил команду показать план выполнения, а не результирующий набор запроса. План включает два этапа. Первый показывает, как будет осуществляться доступ к таблице `customer` (выполняется доступ ко всем строкам, поскольку столбец `cust_type_cd` не имеет индекса). Второй показывает, как будет организован доступ к таблице `account` (посредством внешнего ключа `fk_a_cust_id`). Формирование и расшифровка планов выполнения не является предметом рассмотрения для вводной книги по SQL, поэтому в конце данного раздела приведены имеющиеся источники (книги и учебные курсы). Также в продаже есть несколько замечательных инструментов, которые помогут строить, вычислять и настраивать SQL-выражения. Вот некоторые из них:

- Quest Central for Oracle и Quest Central for SQL Server
- Quest Software (*http://www.quest.com*)
- Embarcadero SQL Profiler
- Embarcadero Technologies (*http://www.embarcadero.com*)
- Oracle Enterprise Manager Tuning Pack

- Oracle Corporation (*http://www.oracle.com*)

Можно порекомендовать несколько хороших книг по настройке производительности SQL:

Томас Кайт (Thomas Kyte) «Effective Oracle by Design», McGraw-Hill Osborne Media, 2003.[1]

Кэри Миллсап (Cary Millsap) и Джефф Хольт (Jeff Holt) «Optimizing Oracle Performance», O'Reilly, 2003.[2]

Джереми Заводный (Jeremy Zawodny) и Дерек Баллинг (Derek Balling) «High Performance MySQL» (Высокопроизводительный MySQL), O'Reilly, 2004.

Кен Ингланд (Ken England) «Microsoft SQL Server 2000 Performance Optimization and Tuning Handbook» (Справочник по оптимизации производительности и настройке Microsoft SQL Server 2000), Digital Press, 2001.

Также есть много курсов по настройке производительности, предоставляемых следующими учебными центрами:

- Oracle University (*http://education.oracle.com*)
- Learning Tree International (*http://www.learningtree.com*)
- Microsoft Learning (*learning*)
- MySQL Training (*training*)

Администрирование баз данных

Администрирование БД, на самом деле, – многогранная задача, которая может включать что угодно или все из перечисленного ниже:

- Установка и конфигурирование сервера БД
- Проектирование БД
- Программирование БД, особенно хранимых процедур, функций и триггеров
- Система безопасности БД
- Резервное копирование и восстановление
- Настройка производительности

Организации покрупнее могут нанять одного или нескольких специалистов в каждой из вышеупомянутых областей, а более мелкие часто ожидают, что со всем этим справится один администратор БД. Если вы будете заниматься проектированием, программированием и настройкой, пожалуйста, ознакомьтесь со списками источников, приведенны-

[1] Томас Кайт «Эффективное проектирование приложений Oracle», Лори, 2006.

[2] Кэри Миллсап «Oracle. Оптимизация производительности», Символ-Плюс, 2006.

ми в предыдущих разделах. Однако все администраторы БД должны прочитать общие книги по администрированию или пройти курсы, чтобы профессионально овладеть базовыми навыками администратора, такими как установка и конфигурирование БД, создание пользователей и назначение привилегий, стратегии резервного копирования и восстановления, формирование схем. Еще один основной источник для администраторов – справочное руководство по SQL для используемого сервера БД, где приведен синтаксис SQL-выражений управления схемой, таких как `create index` и `alter table`. Самые популярные справочники:

Кевин Лоуни (Kevin Loney), Боб Брила (Bob Bryla) «Oracle Database 10g DBA Handbook» (Справочник администратора Oracle Database 10g), McGraw-Hill Osborne Media, 2004.

«MySQL Administrator's Guide», MySQL Press, 2004.[1]

Энтони Секвейра (Anthony Sequeira), Брайан Алдерман (Brian Alderman) «The SQL Server 2000 Book», Paraglyph, 2003.

Также есть масса курсов по администрированию, предоставляемых следующими учебными центрами:

- Oracle University (*http://education.oracle.com*)
- Learning Tree International (*http://www.learningtree.com*)
- Microsoft Learning (*learning*)
- MySQL Training (*training*)

Формирование отчетов

Если вы отвечаете за разработку и формирование отчетов для своей организации, вот два самых важных навыка, которые стоит приобрести:

- Знание возможностей механизма создания отчетов, используемых в вашей организации
- Владение в совершенстве реализацией SQL, используемого вашим сервером БД

Хотя большинство инструментов создания отчетов претендует на формирование SQL на базе визуального представления отчета, настоятельно рекомендую игнорировать эту возможность и самостоятельно создавать SQL-выражения для всех нетривиальных отчетов. В этом случае вы точно будете знать, что отправляется серверу БД, и впоследствии сможете лучше поддерживать и настраивать отчеты.

Некоторые механизмы создания отчетов достаточно гибки в отношении получения данных для отчета, но многие инструменты требуют, чтобы все данные одного отчета были сформированы посредством одного запроса. Глубокое понимание SQL, особенно подзапросов (глава 9),

[1] «MySQL. Руководство администратора», Вильямс, 2005.

операций работы с множествами (глава 6) и условной логики (глава 11), позволит создавать гораздо более изощренные отчеты.

Среди хороших книг по составлению отчетов можно назвать:

Питер Блэкберн (Peter Blackburn), Уильям Вон (William Vaughn) «Hitchhiker's Guide to SQL Server 2000 Reporting Services» (Руководство по службам отчетов SQL Server 2000), Addison-Wesley, 2004.

Синди Хаусон (Cindi Howson) «Business Objects: The Complete Reference» (Business Objects. Полное руководство), McGraw-Hill Osborne Media, 2003.

Нейл Фитцджеральд (Neil FitzGerald) и др. «Special Edition Using Crystal Reports 10» (Использование Crystal Reports 10. Специальное издание), Pearson Education, 2004.

Алфавитный указатель

В

Г

Д

Е

З

И

К

Л

М

Н

О

П

P

C

Т

У

Ф

Х

Ц

Ч

Э

Я