

O'REILLY®

Третье
Издание

Изучаем SQL



Алан Болье

ТРЕТЬЕ ИЗДАНИЕ

Изучаем SQL

Генерация, выборка и обработка данных

THIRD EDITION

Learning SQL

Generate, Manipulate, and Retrieve Data

Alan Beaulieu

Beijing · Boston · Farnham · Sebastopol · Tokyo

O'REILLY®

ТРЕТЬЕ ИЗДАНИЕ

Изучаем SQL

Генерация, выборка и обработка данных

Алан Болье

Київ
Комп'ютерне видавництво
“ДІАЛЕКТИКА”
2021

УДК 004.655.3 (075.8)

Б79

Перевод с английского и редакция канд. техн. наук *И.В. Красикова*

Болье, А.

Б79 Изучаем SQL. Генерация, выборка и обработка данных, 3-е изд./
Алан Болье; пер. с англ. И.В. Красикова. — Киев. : “Диалектика”,
2021. — 402 с. : ил. — Парал. тит. англ.

ISBN 978-617-7987-01-6 (укр.)
ISBN 978-1-492-05761-1 (англ.)

Данная книга отличается широким охватом как тем (от азов SQL до таких сложных вопросов, как аналитические функции и работа с большими базами данных), так и конкретных баз данных (MySQL, Oracle Database, SQL Server) и особенностей реализации тех или иных функциональных возможностей SQL на этих серверах. Книга идеально подходит в качестве учебника для начинающего разработчика в области баз данных. В ней описаны все возможные применения языка SQL и наиболее распространенные серверы баз данных.

УДК 004.655.3 (075.8)

Все права защищены.

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства O'Reilly & Associates.

Authorized Russian translation of the English edition of *Learning SQL: Master SQL Fundamentals*, 3rd Edition (ISBN 978-1-492-05761-1) © 2020 Alan Beaulieu. All rights reserved.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the Publisher.

ISBN 978-617-7987-01-6 (укр.)
ISBN 978-1-492-05761-1 (англ.)

© “Диалектика”, перевод, 2021
© 2020 Alan Beaulieu

Оглавление

Предисловие	13
Глава 1. Небольшая предыстория	19
Глава 2. Создание и наполнение базы данных	37
Глава 3. Запросы	67
Глава 4. Фильтрация	91
Глава 5. Запросы к нескольким таблицам	113
Глава 6. Работа с множествами	129
Глава 7. Генерация, обработка и преобразование данных	145
Глава 8. Группировка и агрегация	179
Глава 9. Подзапросы	195
Глава 10. Соединения	223
Глава 11. Условная логика	239
Глава 12. Транзакции	251
Глава 13. Индексы и ограничения	263
Глава 14. Представления	281
Глава 15. Метаданные	295
Глава 16. Аналитические функции	311
Глава 17. Работа с большими базами данных	331
Глава 18. SQL и большие данные	349
Приложение А. Схема базы данных Sakila	367
Приложение Б. Ответы к упражнениям	369
Предметный указатель	397

Содержание

Предисловие	13
Зачем изучать SQL	13
Почему стоит использовать данную книгу	14
Структура книги	14
Соглашения, принятые в этой книге	16
Использование примеров из этой книги	17
Благодарности	18
Ждем ваших отзывов!	18
Глава 1. Небольшая предыстория	19
Введение в базы данных	19
Нереляционные СУБД	20
Реляционная модель	22
Немного терминологии	26
Что такое SQL	26
Классы инструкций SQL	27
SQL: непроцедурный язык	29
Примеры SQL	30
Что такое MySQL	33
Отказ от SQL	34
Что дальше	35
Глава 2. Создание и наполнение базы данных	37
Создание базы данных MySQL	37
Использование инструмента командной строки mysql	39
Типы данных MySQL	40
Символьные данные	41
Числовые данные	45
Временные данные	46

Создание таблицы	49
Шаг 1. Проектирование	49
Шаг 2. Уточнение	50
Шаг 3. Построение инструкции схемы SQL	52
Заполнение и изменение таблиц	56
Добавление данных	56
Изменение данных	61
Удаление данных	61
Когда хорошие инструкции становятся плохими	62
Не уникальный первичный ключ	62
Несуществующий внешний ключ	62
Нарушения значений столбцов	63
Некорректное преобразование данных	63
База данных Sakila	64
Глава 3. Запросы	67
Механика запросов	67
Части запроса	69
Предложение select	70
Псевдонимы столбцов	72
Удаление дубликатов	74
Предложение from	75
Таблицы	76
Связи таблиц	79
Определение псевдонимов таблиц	80
Предложение where	81
Предложения group by и having	84
Предложение order by	85
Сортировка по возрастанию и убыванию	87
Сортировка с помощью номера столбца	88
Проверьте свои знания	89
Глава 4. Фильтрация	91
Вычисление условий	91
Использование скобок	92
Использование оператора not	93
Построение условия	94
Типы условий	95
Условия равенства	95

Условия диапазона	97
Условия членства	102
Условия соответствия	104
Этот таинственный null	107
Проверьте свои знания	110
Глава 5. Запросы к нескольким таблицам	113
Что такое соединение	113
Декартово произведение	114
Внутренние соединения	115
Синтаксис соединения ANSI	117
Соединение трех и более таблиц	119
Использование подзапросов в качестве таблиц	122
Использование одной таблицы дважды	124
Самосоединение	125
Проверьте свои знания	126
Глава 6. Работа с множествами	129
Основы теории множеств	129
Теория множеств на практике	132
Операторы для работы с множествами	134
Оператор union	134
Оператор intersect	137
Оператор except	138
Правила применения операторов для работы с множествами	139
Сортировка результатов составного запроса	140
Приоритеты операций над множествами	141
Проверьте свои знания	143
Глава 7. Генерация, обработка и преобразование данных	145
Работа со строковыми данными	145
Генерация строк	146
Манипуляции строками	152
Работа с числовыми данными	160
Выполнение математических функций	161
Управление точностью чисел	163
Работа со знаковыми данными	165
Работа с временными данными	166
Часовые пояса	166

Генерация временных данных	168
Манипуляции временными данными	172
Функции преобразования	177
Проверьте свои знания	178
Глава 8. Группировка и агрегация	179
Концепции группировки	179
Агрегатные функции	183
Неявная и явная группировка	184
Использование выражений	186
Обработка значений null	186
Генерация групп	188
Группировка по одному столбцу	188
Многостолбцовая группировка	189
Группировка с помощью выражений	190
Генерация итоговых данных	190
Условия группового фильтра	192
Проверьте свои знания	194
Глава 9. Подзапросы	195
Что такое подзапрос	195
Типы подзапросов	197
Некоррелированные подзапросы	197
Подзапросы с несколькими строками и одним столбцом	199
Многостолбцовые подзапросы	204
Коррелированные подзапросы	205
Оператор exists	207
Работа с данными с помощью коррелированных подзапросов	209
Применение подзапросов	211
Подзапросы как источники данных	211
Подзапросы как генераторы выражений	218
В заключение	221
Проверьте свои знания	221
Глава 10. Соединения	223
Внешние соединения	223
Левое и правое внешние соединения	226
Трехсторонние внешние соединения	227
Перекрестные соединения	228

Естественные соединения	235
Проверьте свои знания	237
Глава 11. Условная логика	239
Что такое условная логика	239
Выражение case	240
Поисковые выражения case	240
Простые выражения case	242
Примеры выражений case	243
Преобразования результирующего набора	243
Проверка существования	244
Ошибки деления на нуль	247
Условные обновления	248
Обработка значений null	249
Проверьте свои знания	250
Глава 12. Транзакции	251
Многопользовательские базы данных	251
Блокировка	252
Гранулярность блокировок	253
Что такое транзакция	254
Запуск транзакции	256
Завершение транзакции	257
Точки сохранения транзакции	259
Проверьте свои знания	262
Глава 13. Индексы и ограничения	263
Индексы	263
Создание индекса	264
Типы индексов	269
Как используются индексы	272
Обратная сторона индексов	274
Ограничения	275
Создание ограничения	276
Проверьте свои знания	279
Глава 14. Представления	281
Что такое представление	281
Зачем использовать представления	284

Безопасность данных	284
Агрегация данных	285
Сокрытие сложности	286
Соединение разделенных данных	287
Обновляемые представления	288
Обновление простых представлений	288
Обновление сложных представлений	290
Проверьте свои знания	293
Глава 15. Метаданные	295
Данные о данных	295
information_schema	296
Работа с метаданными	302
Сценарии генерации схемы	302
Проверка базы данных	305
Динамическая генерация SQL	306
Проверьте свои знания	310
Глава 16. Аналитические функции	311
Концепции аналитических функций	311
Окна данных	312
Локализованная сортировка	313
Ранжирование	315
Функции ранжирования	315
Генерация нескольких рейтингов	318
Функции отчетности	321
Рамки окон	324
Запаздывание и опережение	327
Конкатенация значений в столбце	328
Проверьте свои знания	329
Глава 17. Работа с большими базами данных	331
Секционирование	331
Концепции секционирования	332
Секционирование таблицы	333
Секционирование индекса	333
Методы секционирования	334
Преимущества секционирования	343
Кластеризация	343

Шардинг	344
Большие данные	345
Hadoop	346
NoSQL и базы данных документов	347
Облачные вычисления	347
Заключение	348
Глава 18. SQL и большие данные	349
Введение в Apache Drill	349
Запрос файлов с помощью Apache Drill	350
Запрос MySQL с использованием Apache Drill	353
Запрос MongoDB с использованием Apache Drill	356
Apache Drill и несколько источников данных	363
Будущее SQL	365
Приложение А. Схема базы данных Sakila	367
Приложение Б. Ответы к упражнениям	369
Глава 3	369
Глава 4	371
Глава 5	373
Глава 6	375
Глава 7	378
Глава 8	378
Глава 9	380
Глава 10	384
Глава 11	386
Глава 12	387
Глава 13	388
Глава 14	389
Глава 15	391
Глава 16	393
Предметный указатель	397

Предисловие

Языки программирования постоянно появляются и исчезают, и сегодня используется очень мало языков, которые имеют корни, уходящие в прошлое более чем на десятилетие. Вот некоторые из них: COBOL, который все еще довольно активно используется в мейнфреймах; Java, который родился в середине 1990-х и стал одним из самых распространенных языков программирования; C, который по-прежнему довольно популярен при разработке операционных систем и серверов, а также для встроенных систем. В области баз данных у нас имеется SQL, который появился в 1970-х годах.

SQL был разработан как язык для создания, выборки и обработки данных из реляционных баз данных, которые существуют уже более 40 лет. Однако за последнее десятилетие или около того приобрели большую популярность другие платформы данных, такие как Hadoop, Spark и NoSQL, занявшие свои ниши на рынке реляционных баз данных. Однако, как описано в нескольких последних главах этой книги, язык SQL постоянно развивается, чтобы упростить выборку данных на различных платформах независимо от того, где хранятся данные — в таблицах, документах или простых “плоских” файлах.

Зачем изучать SQL

Независимо от того, будете ли вы использовать реляционную базу данных, если вы работаете в области науки о данных или бизнес-аналитиком либо сталкиваетесь с некоторым иным аспектом анализа данных, вам, вероятно, потребуется знание SQL, а также других языков/платформ, таких как Python и R. Данные окружают нас в огромных количествах (и их объемы продолжают возрастать быстрыми темпами), и люди, способные извлекать из них значимую информацию, пользуются большим спросом.

Почему стоит использовать данную книгу

Есть много книг, в которых к вам относятся как к чайнику, идиоту или какому-то иному простаку, но эти книги, как правило, поверхностны. На другом конце спектра — справочные руководства, в которых подробно описывается каждая перестановка каждой инструкции языка, что может быть полезно, только если у вас уже есть хорошее представление о том, что вы хотите сделать, и нужно только вспомнить синтаксис. В этой книге мы попытаемся найти золотую середину, начав с некоторых азов языка SQL, пройдя через его основы и продолжив более “продвинутыми” функциями, которые позволят вам по-настоящему проявить себя. Кроме того, в последней главе показано, как можно запрашивать информацию из нереляционных баз данных, — вопрос, достаточно редко рассматриваемый в книгах, являющихся вводным курсом.

Структура книги

Книга состоит из 18 глав и двух приложений.

Глава 1, “Небольшая предыстория”

История компьютеризированных баз данных, в том числе реляционной модели и языка SQL.

Глава 2, “Создание и наполнение базы данных”

Создание базы данных MySQL, а также таблиц, используемых в качестве примеров в этой книге, и заполнение таблиц данными.

Глава 3, “Запросы”

Инструкция select и наиболее распространенные конструкции (select, from, where).

Глава 4, “Фильтрация”

Типы условий, которые можно использовать в конструкции where инструкции select, update или delete.

Глава 5, “Запросы к нескольким таблицам”

Использование запросов нескольких таблиц посредством их соединения.

Глава 6, “Работа с множествами”

Наборы данных и их взаимодействие в рамках запросов.

Глава 7, “Генерация, обработка и преобразование данных”

Встроенные функции, используемые для управления данными или их преобразования.

Глава 8, “Группировка и агрегация”

Способы агрегирования данных.

Глава 9, “Подзапросы”

Подзапросы и их использование.

Глава 10, “Соединения”

Дальнейшее исследование типов соединения таблиц.

Глава 11, “Условная логика”

Применение условной логики в инструкциях select, insert, update и delete.

Глава 12, “Транзакции”

Транзакции и их использование.

Глава 13, “Индексы и ограничения”

Индексы и ограничения.

Глава 14, “Представления”

Создание интерфейса для защиты пользователей от сложных данных.

Глава 15, “Метаданные”

Словарь данных, его преимущества и работа с ним.

Глава 16, “Аналитические функции”

Функциональность, используемая для создания рейтингов, промежуточных итогов и других значений, активно используемых в отчетности и анализе.

Глава 17, “Работа с большими базами данных”

Технологии, упрощающие управление и обход очень больших баз данных.

Глава 18, “SQL и большие данные”

Вариации языка SQL, позволяющие извлекать данные из нереляционных платформ данных.

Приложение А, “Схема базы данных Sakila”

Схема базы данных, используемая для всех примеров в книге.

Приложение Б, “Ответы к упражнениям”

Ответы к упражнениям.

Соглашения, принятые в этой книге

В этой книге используются следующие типографские соглашения.

Курсив

Используется для обозначения новых терминов.

Моноширинный шрифт

Используется для оформления листингов программ, а также внутри абзацев для ссылки на такие элементы программ, как имена переменных или функций, базы данных, типы данных, переменные среды, инструкции и ключевые слова, а также URL-адреса, адреса электронной почты, имена и расширения файлов.

Моноширинный курсив

Используется для оформления текста, который следует заменить значениями, вводимыми пользователем, или значениями, определенными из контекста.

Полужирный моноширинный шрифт

Используется для оформления команд или другого текста, который пользователь должен набирать буквально.



Обозначает совет, предложение или общее примечание. Например, здесь такие примечания используются, чтобы указать новые полезные функции в Oracle9i.



Обозначает предупреждение или предостережение, например предупреждение о том, что некоторая инструкция SQL может иметь непредвиденные последствия, если не использовать ее с осторожностью.

Использование примеров из этой книги

Чтобы экспериментировать с данными, использованными в примерах в этой книге, у вас есть два варианта.

- Загрузите и установите сервер MySQL версии 8.0 (или более поздней) и загрузите пример базы данных Sakila по адресу <https://dev.mysql.com/doc/index-other.html>.
- Перейдите по адресу <https://www.katacoda.com/mysql-db-sandbox/scenarios/mysql-sandbox>, чтобы получить доступ к “песочнице” MySQL, в которой база данных Sakila загружена в экземпляр MySQL. Вам потребуется создать (бесплатную) учетную запись Katacoda, а затем щелкнуть на кнопке **Start Scenario**.

Если вы выберете второй вариант, то, как только вы запустите сценарий, будет установлен и запущен сервер MySQL, а затем загружены схема и данные Sakila. Когда все будет готово, появится стандартное приглашение `mysql>`, и вы сможете начать посылать запросы в базу данных. Это, безусловно, самый простой вариант, и я думаю, что большинство читателей выберет именно его; если он вам нравится, смело переходите к следующему разделу.

Если вы предпочитаете иметь собственную копию данных и хотите, чтобы внесенные вами изменения были постоянными, или если вы просто заинтересованы в установке сервера MySQL на собственной машине, то, возможно, вы предпочтете первый вариант. Вы можете также использовать сервер MySQL, размещенный в такой среде, как Amazon Web Services или Google Cloud. В любом случае вам нужно будет выполнить установку и настройку самостоятельно, так как этот материал выходит за рамки данной книги. Как только ваша база данных станет доступной, вам нужно будет выполнить еще несколько шагов, чтобы загрузить базу данных Sakila.

Сначала запустите клиент командной строки `mysql` и укажите пароль, а затем выполните следующие действия.

1. Перейдите по адресу <https://dev.mysql.com/doc/index-other.html> и загрузите файлы “`sakila database`” в разделе **Example Databases**.
2. Поместите файлы в локальный каталог, например в `C:\temp\sakila-db` (используется в следующих двух шагах; воспользуйтесь путем к вашему реальному каталогу).
3. Введите `source c:\temp\sakila-db\sakila-schema.sql`; и нажмите `<Enter>`.

4. Введите source c:\temp\sakila-db\sakila-data.sql; и нажмите <Enter>.

Теперь у вас должна быть рабочая база данных, заполненная всеми данными, необходимыми для примеров в этой книге.

Благодарности

Я хотел бы поблагодарить моего редактора Джеффа Блейла (Jeff Bleiel) за то, что он помог сделать это издание реальностью, а также Томаса Нилда (Thomas Nield), Энн Уайт-Уоткинс (Ann White-Watkins) и Чарльза Живра (Charles Givre), которые были столь любезны, что просмотрели мою книгу и высказали свои замечания. Спасибо также Деб Бейкер (Deb Baker), Джесс Хаберман (Jess Haberman) и всему персоналу O'Reilly Media, которые участвовали в создании этой книги. Наконец, я благодарю свою жену Нэнси (Nancy) и моих дочерей Мишель (Michelle) и Николь (Nicole) за поддержку и терпение.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info.dialektika@gmail.com
WWW: <http://www.dialektika.com>

Небольшая предыстория

Прежде чем мы засучим рукава и приступим к работе, было бы полезно изучить историю технологий баз данных, чтобы лучше понимать, как эволюционировали реляционные базы данных и язык SQL. Поэтому я хотел бы начать с представления некоторых основных концепций баз данных и изучения истории компьютеризированного хранения и поиска данных.



Читателям, желающим поскорее начать писать запросы, можно пропустить все до главы 3, “Запросы”, но я рекомендую позже вернуться к первым двум главам, чтобы лучше понять историю и преимущества языка SQL.

Введение в базы данных

База данных — это не что иное, как набор связанный информации. Например, телефонная книга — это база данных имен, номеров телефонов и адресов всех людей, живущих в конкретном районе. Хотя телефонная книга, безусловно, встречается повсеместно и является часто используемой базой данных, она страдает следующими недостатками.

- Поиск телефонного номера человека может занять много времени, особенно если книга содержит большое количество записей.
- Телефонная книга индексируется только по фамилии/имени, поэтому поиск имен людей, живущих по определенному адресу, хотя и возможен теоретически, на практике для этой базы данных не реален.
- С момента печати телефонной книги информация в ней становится все менее и менее точной, так как люди переезжают в этот район или уезжают из него, меняют свой номер телефона или переезжают в другое место в этом же районе.

Те же недостатки, что и у телефонной книги, присущи любым ручным системам хранения данных, например историям болезней, хранящимся в картотеке. Из-за громоздкости бумажных баз данных одними из первых компьютерных приложений были системы управления базами данных (СУБД), которые представляли собой компьютеризированные механизмы хранения и поиска данных. Поскольку СУБД хранит данные в электронном виде, а не на бумаге, такая компьютеризированная система может гораздо быстрее извлекать данные, индексировать их разными способами и предоставлять пользователям самую последнюю информацию.

Ранние СУБД управляли данными, хранящимися на магнитных лентах. Поскольку, как правило, кассет было гораздо больше, чем считывателей, техническим специалистам приходилось постоянно загружать и выгружать ленты при запросе конкретных данных. Поскольку компьютеры той эпохи имели очень мало памяти, несколько запросов к одним и тем же данным обычно требовали многократного чтения ленты. Хотя такие СУБД были значительным улучшением по сравнению с бумажными базами данных, они очень далеки от современных технологий. (Современные СУБД могут управлять петабайтами данных, обращаться к кластерам серверов, каждый из которых кеширует десятки гигабайт данных в высокоскоростной памяти... но я несколько забегаю вперед.)

Нереляционные СУБД



В этом разделе содержится базовая информация о СУБД, предваряющих реляционные. Читатели, которые хотят поскорее погрузиться в SQL, могут сразу перейти к следующему разделу.

В течение нескольких первых десятилетий компьютеризированных СУБД хранение и предоставление информации пользователям производилось различными способами. В иерархической СУБД, например, данные представлены в виде одной или нескольких древовидных структур. На рис. 1.1 показано, как могут быть представлены в виде деревьев данные, относящиеся к банковским счетам Джорджа Блейка и Сью Смит.

У Джорджа и Сью есть собственные деревья, содержащие их счета и транзакции, проведенные с этими счетами. Иерархическая СУБД предоставляет инструменты для поиска дерева конкретного клиента, а затем выполняет обход дерева в поисках нужных счетов и/или транзакций. У каждого узла в дереве может быть либо нуль, либо один родительский узел и нуль, один или

много дочерних узлов. Такая конфигурация известна как *иерархия с одним родителем*.

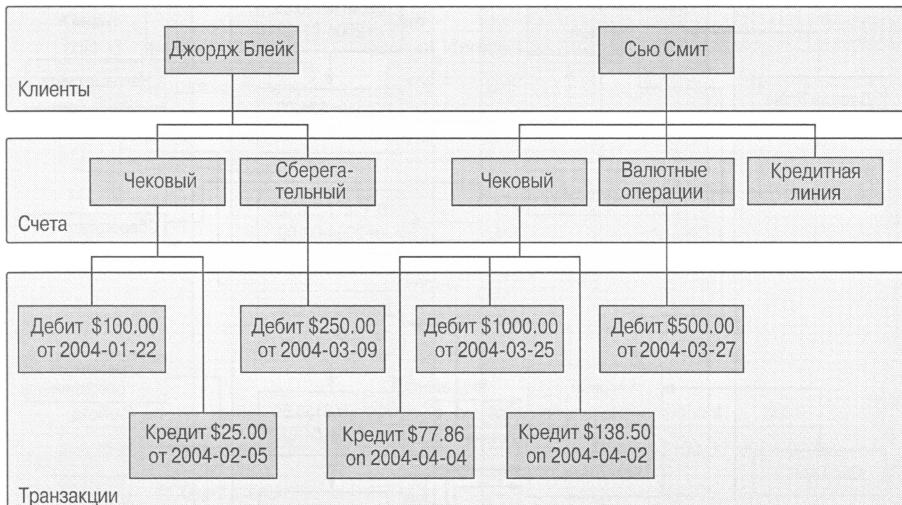


Рис. 1.1. Иерархическое представление данных

Другой распространенный подход, называемый *сетевой СУБД*, предоставляет наборы записей и наборы ссылок, которые определяют отношения между различными записями. На рис. 1.2 показано, как те же счета Джорджа и Сью могут выглядеть в такой системе.

Чтобы найти транзакции валютных операций на счете Сью, необходимо выполнить следующие действия.

1. Найти запись о клиенте Сью Смит.
2. Перейти по ссылке из записи клиента Сью Смит к списку ее счетов.
3. Пройти по цепочке счетов, пока не будет найден валютный счет.
4. Перейти по ссылке из записи валютного счета к списку транзакций.

Одна интересная особенность сетевых СУБД демонстрируется набором записей Услуги в правом ряду на рис. 1.2. Обратите внимание, что каждая запись об услуге (чековые счета, сберегательные счета и т.д.) указывает на список учетных записей, относящихся к этому типу услуг. Таким образом, к учетным записям можно получить доступ из нескольких мест (как из записей о клиентах, так и из записей об услугах), что позволяет сетевой базе данных действовать в качестве *иерархии со многими родителями*.

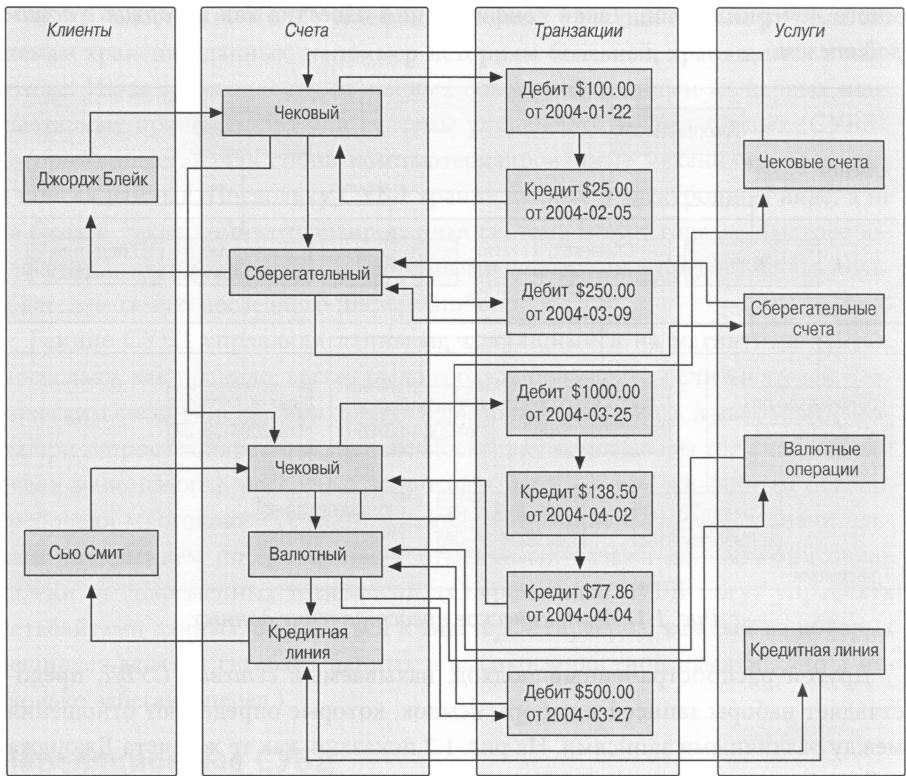


Рис. 1.2. Сетевое представление данных

И иерархические, и сетевые СУБД и сегодня живы и здоровы, хотя в основном используются в мире мейнфреймов. Кроме того, иерархические СУБД пережили второе рождение в сфере служб каталогов, таких как Microsoft Active Directory и Apache Directory Server с открытым исходным кодом. Однако, начиная с 1970-х годов, начал укореняться новый способ представления данных, более строгий, но простой для понимания и реализации.

Реляционная модель

В 1970 году д-р Э.Ф. Кодд (E.F. Codd) из исследовательской лаборатории IBM опубликовал статью под названием *Реляционная модель данных для больших общих банков данных*, в которой было предложено представление данных в виде наборов таблиц. Вместо указателей для перехода между связанными сущностями для связывания записей в разных таблицах используются

избыточные данные. На рис. 1.3 показано, как в этом контексте будет отображаться информация о клиентах Джордже и Сью.

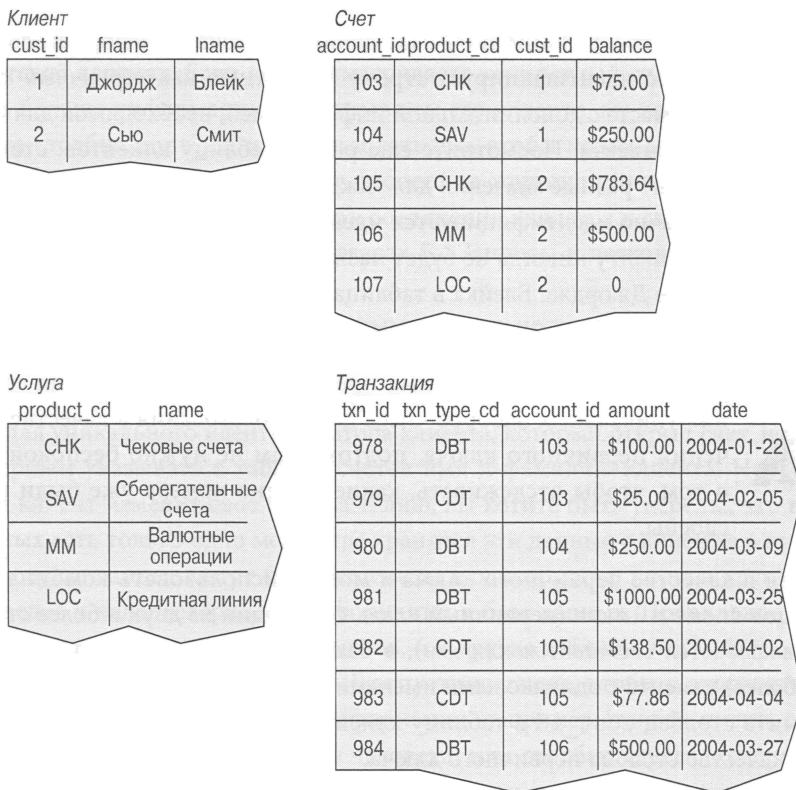


Рис. 1.3. Реляционное представление данных

Четыре таблицы на рис. 1.3 представляют четыре объекта, которые обсуждались до сих пор: клиент, услуга, счет и транзакция. Просматривая верхнюю часть таблицы клиентов на рис. 1.3, вы видите три столбца: `cust_id` (номер идентификатора клиента), `fname` (имя клиента) и `lname` (фамилия клиента). Взглянув на таблицу клиентов, вы можете увидеть две строки, одна из которых содержит данные Джорджа Блейка, а другая — данные Сью Смит. Количество столбцов, которые может содержать таблица, отличается от сервера к серверу, но обычно оно достаточно велико, чтобы не создавать проблем (например, Microsoft SQL Server позволяет содержать в таблице до 1024 столбцов). Количество строк, которые может содержать таблица, определяется в большей степени физическими ограничениями (доступность

дискового пространства) и возможностью поддержки (насколько большой может стать таблица, прежде чем с ней станет трудно работать), а не ограничениями сервера базы данных.

Каждая таблица в реляционной базе данных включает информацию, которая однозначно идентифицирует строку (известную как *первичный ключ* (primary key)), вместе с дополнительной информацией, необходимой для полного описания объекта. Посмотрите еще раз на таблицу клиентов: столбец `cust_id` содержит разные значения для каждого клиента; Джордж Блейк, например, однозначно идентифицируется идентификатором клиента 1. Ни одному другому клиенту никогда не будет назначен этот идентификатор, и чтобы найти данные Джорджа Блейка в таблице клиентов, не требуется никакой другой информации.



Каждый сервер базы данных предоставляет механизм для создания уникальных наборов чисел для использования в качестве значений первичного ключа, поэтому вам не нужно беспокоиться о том, чтобы отслеживать, какие именно номера уже были присвоены.

Хотя в качестве первичного ключа я мог бы использовать комбинацию столбцов `fname` и `lname` (первичный ключ, состоящий из двух и более столбцов, называется *составным ключом*), в банке запросто могут найтись два (или более) клиента с одинаковыми именами и фамилиями. Поэтому я решил включить столбец `cust_id` в таблицу клиентов специально для использования в качестве столбца первичного ключа.



В этом примере выбор `fname/lname` в качестве первичного ключа приведет к тому, что именуется *естественным ключом*, в то время как выбор `cust_id` в качестве такового будет называться *суррогатным ключом*. Решение о том, следует ли использовать естественные или суррогатные ключи, остается на усмотрение разработчика базы данных, но в данном случае выбор очевиден, поскольку фамилия человека может измениться (например, при принятии фамилии супруга), а столбцы первичного ключа после присваивания значения никогда не должны изменяться.

Некоторые таблицы также содержат информацию, используемую для перехода к другой таблице (это и есть упомянутые ранее “избыточные данные”). Например, таблица счетов включает столбец `cust_id`, который содержит

уникальный идентификатор клиента, открывшего счет, вместе со столбцом `product_cd`, который содержит уникальный идентификатор услуги, которой соответствует счет. Эти столбцы известны как *внешние ключи* (foreign key) и служат той же цели, что и линии, соединяющие объекты в иерархической и сетевой версиях. Если вы просматриваете конкретный счет и хотите получить больше информации об открывшем его клиенте, вы должны взять значение столбца `cust_id` и использовать его, чтобы найти соответствующую строку в таблице клиентов (этот процесс на жаргоне реляционных баз данных известен как *соединение* (join); соединения вводятся в главе 3, “Запросы”, и подробно рассматриваются в главах 5, “Запросы к нескольким таблицам”, и 10, “Соединения”).

Может показаться расточительным хранить несколько копий одних и тех же данных, но в реляционной модели совершенно ясно, какие избыточные данные могут храниться. Например, в таблицу счетов можно включить столбец для уникального идентификатора клиента, который открыл счет, но будет неверным указывать в таблице счетов имя и фамилию клиента. Например, если клиент изменит свое имя/фамилию, вы хотите быть уверены, что в базе данных есть только одно место, где хранятся эти данные о клиенте; в противном случае данные могут быть изменены в одном месте, но не в другом, что приведет к недостоверности информации в базе данных. Правильное место для этих данных — таблица клиентов, и в другие таблицы могут быть включены только значения `cust_id`. Недопустимо также, чтобы в одном столбце содержалось несколько частей информации, например столбец имени, содержащий как имя, так и фамилию человека, или столбец адреса, который содержит информацию об улице, городе, штате и почтовом индексе. Процесс доработки дизайна базы данных, гарантирующий, что каждая независимая часть информации находится только в одном месте (кроме внешних ключей), называется *нормализацией*.

Возвращаясь к четырем таблицам на рис. 1.3, вы можете задаться вопросом, как использовать эти таблицы, чтобы найти транзакции Джорджа Блейка по его чековому счету. Сначала вы находите уникальный идентификатор Джорджа Блейка в таблице клиентов. Затем вы находите строку в таблице счетов, столбец `cust_id` которой содержит значение уникального идентификатора Джорджа, а столбец `product_cd` которой соответствует строке в таблице услуг, столбец `name` которой равен “Чековые счета”. Наконец, нужно найти строки в таблице транзакций, столбец `account_id` которых соответствует уникальному идентификатору из таблицы счетов. Это может

показаться сложным, но, как вы вскоре увидите, все это можно сделать с помощью одной команды языка SQL.

Немного терминологии

В предыдущих разделах я ввел новую терминологию, так что, возможно, пришло время дать некоторые формальные определения. В табл. 1.1 показаны термины, которые мы используем в оставшейся части книги, а также их определения.

Таблица 1.1. Термины и определения

Термин	Определение
Объект, сущность (entity)	Что интересное для сообщества пользователей баз данных, например клиенты, запчасти, географические местоположения и т.д.
Столбец (column)	Отдельная часть данных, хранящаяся в таблице
Строка (row)	Набор столбцов, которые вместе взятые полностью описывают объект или какое-либо действие с ним. Именуется также <i>записью</i>
Таблица (table)	Набор строк, хранящихся либо в оперативной памяти (непостоянная), либо в энергонезависимом хранилище (постоянная)
Результатирующий набор (result set)	Другое название непостоянной таблицы, обычно являющейся результатом SQL-запроса
Первичный ключ (primary key)	Один или несколько столбцов, которые можно использовать как уникальный идентификатор для каждой строки таблицы
Внешний ключ (foreign key)	Один или несколько столбцов, которые можно использовать совместно для идентификации одной строки в другой таблице

Что такое SQL

Кроме определения реляционной модели, Кодд предложил язык, названный “DSL/Alpha”, для управления данными в реляционных таблицах. Вскоре после выхода статьи Кодда IBM поручила группе разработчиков создать прототип базы данных, основанный на идеях Кодда. Эта группа создала упрощенную версию DSL/Alpha, которую они назвали “SQUARE”. Усовершенствования в SQUARE привели к созданию языка под названием “SEQUEL”, которое в конечном итоге было сокращено до “SQL”. SQL начался как язык, используемый для управления данными в реляционных базах данных, но постепенно превратился (как вы увидите в конце этой книги) в язык для управления данными в различных технологиях баз данных.

SQL уже более 40 лет, и за это время он претерпел большие изменения. В середине 1980-х годов Американский национальный институт стандартов

(ANSI) начал работу над первым стандартом языка SQL, опубликованным в 1986 году. Последующие усовершенствования привели к появлению новых выпусков стандарта SQL в 1989, 1992, 1999, 2003, 2006, 2008, 2011 и 2016 годах. Наряду с усовершенствованиями основного языка, в язык SQL были добавлены новые возможности, среди прочего — для возможности объектно-ориентированной функциональности. Более поздние стандарты ориентированы на интеграцию родственных технологий, таких как расширяемый язык разметки (XML) и объектная нотация JavaScript (JSON).

SQL тесно связан с реляционной моделью, потому что результат SQL-запроса представляет собой таблицу (в этом контексте также именуемую *результатирующим набором*). Таким образом, в реляционной базе данных новая постоянная таблица может быть создана просто путем сохранения результатов запроса. Аналогично запрос может использовать в качестве входных данных как постоянные таблицы, так и результатирующие наборы других запросов (мы подробно рассмотрим этот вопрос в главе 9, “Подзапросы”).

И последнее замечание: SQL не является аббревиатурой для чего-либо (хотя многие люди будут настаивать на том, что это означает *Structured Query Language* (язык структурированных запросов)). Что касается названия языка, то одинаково приемлемо как произносить буквы по отдельности (S.Q.L.), так и использовать слово *sequel*.

Классы инструкций SQL

Язык SQL делится на несколько отдельных частей: части, которые мы рассматриваем в этой книге, включают *инструкции схемы SQL*, которые используются для определения структур данных, хранящихся в базе данных; *инструкции данных SQL*, которые используются для управления структурами данных, ранее определенными с использованием инструкций схемы SQL; и *инструкции транзакций SQL*, которые используются для начала, завершения и отката транзакций (концепции, рассматриваемые в главе 12, “Транзакции”). Например, чтобы создать новую таблицу в своей базе данных, необходимо использовать инструкцию схемы SQL `create table`, в то время как процесс наполнения новой таблицы данными требует инструкции данных SQL `insert`.

Вот как выглядит инструкция схемы SQL, которая создает таблицу с именем `corporation`:

```
CREATE TABLE corporation
  (corp_id SMALLINT,
```

```
    name VARCHAR(30),  
    CONSTRAINT pk_corporation PRIMARY KEY (corp_id)  
);
```

Этот оператор создает таблицу с двумя столбцами, `corp_id` и `name`, при этом столбец `corp_id` указан как первичный ключ таблицы. Мы исследуем детали этой инструкции, например различные типы данных, доступные в MySQL, в главе 2, “Создание и наполнение базы данных”. А вот как выглядит инструкция данных SQL, которая вставляет строку в таблицу `corporation` для Acme Paper Corporation:

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

Эта инструкция добавляет в таблицу `corporation` строку со значением 27 для столбца `corp_id` и значением *Acme Paper Corporation* для столбца `name`.

Наконец, вот какой вид имеет простой оператор `select` для извлечения только что созданных данных:

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

Все элементы базы данных, созданные с помощью инструкций схемы SQL, хранятся в специальном наборе таблиц, называемом *словарем данных*. Эти “данные о базе данных” известны под общим названием *метаданные* и рассматриваются в главе 15, “Метаданные”. Как и таблицы, созданные вами, таблицы словаря данных можно запрашивать с помощью оператора `select`, что позволяет вам обнаруживать текущие структуры данных, развернутые в базе данных во время выполнения. Например, если вас попросят написать отчет, показывающий новые учетные записи, созданные в прошлом месяце, вы можете либо жестко закодировать имена столбцов в таблице `account`, которые известны при написании отчета, либо запросить словарь данных, чтобы определить текущий набор столбцов и динамически генерировать отчет при каждом его выполнении.

Большая часть этой книги посвящена части языка SQL, относящейся к данным, которая состоит из команд `select`, `update`, `insert` и `delete`. Инструкции схемы SQL показаны в главе 2, “Создание и наполнение базы данных”, которая проведет вас через проектирование и создание некоторых простых

таблиц. В целом инструкции схемы SQL не требуют особого обсуждения, кроме их синтаксиса, тогда как инструкции данных SQL, хотя сами по себе и немногочисленны, имеют массу возможностей, которые следует внимательно изучить. Поэтому, хотя я пытаюсь познакомить вас со многими инструкциями схемы SQL, большинство глав этой книги сосредоточено на инструкциях данных SQL.

SQL: непроцедурный язык

Если вы работали с языками программирования, то привыкли определять переменные и структуры данных, использовать условную логику (*if-then-else*), циклы (*do while ... end*) и разбиение кода на мелкие многократно используемые части (объекты, функции, процедуры). Ваш код обрабатывается компилятором, а получившийся выполнимый файл выполняет в точности (хорошо, не всегда *в точности*) то, что вы запрограммировали. Работаете ли вы с Java, Python, Scala или каким-либо иным *процедурным* языком, вы полностью контролируете то, что делает программа.



Процедурный язык определяет как желаемые результаты, так и механизм, или процесс, с помощью которого эти результаты достигаются. Непроцедурный язык также определяет желаемые результаты, но процесс, с помощью которого достигаются результаты, остается на усмотрение внешнего агента.

При использовании SQL вам придется отказаться от части контроля, к которому вы привыкли, потому что операторы SQL определяют необходимые входные и выходные данные, но способ выполнения инструкции оставлен на усмотрение компонента вашего механизма базы данных, известного как *оптимизатор*. Задача оптимизатора — просматривать ваши инструкции SQL и, принимая во внимание, как именно настроены таблицы и какие индексы доступны, определять самый эффективный способ выполнения (хорошо, не всегда *самый эффективный*). Большинство механизмов баз данных позволяют вам влиять на решения оптимизатора, указывая для него подсказки, например предлагая использовать определенный индекс. Однако большинство пользователей SQL никогда не достигнут этого уровня сложности и оставят такую настройку администратору базы данных или эксперту по производительности.

Таким образом, с помощью SQL вы не сможете писать полноценные приложения. Если только вы не пишете простой сценарий для управления

определенными данными, вам необходимо интегрировать SQL с вашим языком программирования. Некоторые поставщики баз данных сделали это вместо вас, — например, язык PL/SQL Oracle, язык хранимых процедур MySQL и язык Transact-SQL от Microsoft. В этих языках инструкции данных SQL являются частью грамматики языка, что позволяет легко интегрировать запросы к базе данных с процедурными командами. Однако, если вы используете язык, не зависящий от базы данных, такой как Java или Python, вам нужно будет использовать набор инструментов/API для выполнения инструкций SQL из вашего кода. Одни из этих наборов инструментов предоставляются поставщиком вашей базы данных, тогда как другие созданы сторонними поставщиками или поставщиками с открытым исходным кодом. В табл. 1.2 показаны некоторые из доступных вариантов интеграции SQL в конкретный язык.

Таблица 1.2. Инструменты интеграции SQL

Язык	Инструментарий
Java	JDBC (Java Database Connectivity)
C#	ADO.NET (Microsoft)
Ruby	Ruby DBI
Python	Python DB
Go	Пакет database/sql

Если вам нужно только интерактивно выполнять команды SQL, то каждый поставщик баз данных предоставляет как минимум простой инструмент командной строки, обеспечивающий отправку команд SQL базе данных и просмотр результатов. Большинство поставщиков также предоставляют графический инструментарий, который включает в себя окно, показывающее ваши команды SQL, и другое окно — с результатами выполнения этих команд SQL. Дополнительно имеются сторонние инструменты, например SQuirrel, который может подключаться через JDBC ко многим различным серверам баз данных. Поскольку примеры в этой книге выполняются для базы данных MySQL, для выполнения примеров и форматирования результатов я использую инструмент командной строки mysql, который включен в установку MySQL.

Примеры SQL

Ранее в этой главе я обещал показать вам инструкцию SQL, которая вернет все транзакции чекового счета Джорджа Блейка. Вот она:

```

SELECT t.txn_id, t.txn_type_cd, t.txn_date, t.amount
FROM individual i
    INNER JOIN account a ON i.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 i.fname = 'George' AND i.lname = 'Blake'
    AND p.name = 'checking account';

```

txnid	txntypecd	txndate	amount
11	DBT	2008-01-05 00:00:00	100.00

1 row in set (0.00 sec)

Не вдаваясь пока что в подробности, замечу, что этот запрос идентифицирует строку в таблице `individual` для Джорджа Блейка и строку в таблице `product` для "checking account", находит строку в таблице `account` для данной комбинации человека/услуги и возвращает четыре столбца из таблицы `transaction` для всех транзакций этого счета. Если вы знаете, что идентификатор клиента Джорджа Блейка равен 8, а текущие счета обозначены кодом 'CHK', то, чтобы найти соответствующие транзакции, вы можете найти чековый счет Джорджа Блейка в таблице `account` на основании идентификатора клиента и использовать идентификатор счета:

```

SELECT t.txn_id, t.txn_type_cd, t.txn_date, t.amount
FROM account a
    INNER JOIN transaction t ON t.account_id = a.account_id
WHERE a.cust_id = 8 AND a.product_cd = 'CHK';

```

Я рассмотрю все концепции в этих запросах (и многое другое) в следующих главах, но я просто хотел показать, как будут выглядеть эти запросы.

Предыдущие запросы содержат три разных *предложения* (*clause*): `select`, `from` и `where`. Почти каждый запрос, с которым вы будете сталкиваться, будет включать как минимум эти три предложения, хотя есть и еще несколько, которые можно использовать для более специализированных целей. Роль каждого из этих предложений демонстрируется в следующем фрагменте:

```

SELECT /* Одна или несколько вещей */ ...
FROM  /* Одно или несколько мест */ ...
WHERE /* Одно или несколько условий */ ...

```



Большинство реализаций SQL обрабатывают любой текст между `/*` и `*/` как комментарии.

При построении запроса ваша первая задача обычно состоит в том, чтобы определить, какая таблица (или таблицы) потребуется, а затем добавить ее в предложение `from`. Далее вам понадобится добавить условия в предложение `where`, чтобы отфильтровать данные из этих таблиц, которые вас не интересуют. Наконец, вы решаете, какие столбцы из разных таблиц необходимо получить, и добавляете их в предложение `select`. Вот простой пример, который демонстрирует, как найти всех клиентов с фамилией "Smith":

```
SELECT cust_id, fname  
FROM individual  
WHERE lname = 'Smith';
```

Этот запрос ищет в таблице `individual` все строки, столбец `lname` которых соответствует строке 'Smith', и возвращает столбцы `cust_id` и `fname` этих строк.

Наряду с запросами к базе данных вы, вероятно, будете заполнять базу данных новыми данными и изменять имеющиеся. Вот простой пример вставки новой строки в таблицу `product`:

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

Ой, похоже, мы неправильно написали слово `Deposit`. Никаких проблем — ситуацию можно исправить с помощью инструкции `update`:

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

Обратите внимание, что инструкция `update` так же, как и `select`, содержит предложение `where`. Это связано с тем, что инструкция `update` должна идентифицировать строки, которые необходимо изменить; в данном случае вы указываете, что следует изменить только те строки, значение столбца `product_cd` которых соответствует строке 'CD'. Поскольку столбец `product_cd` является первичным ключом таблицы `product`, следует ожидать, что ваша инструкция обновления изменит ровно одну строку (или ни одной, если такое значение в таблице отсутствует). Всякий раз, выполняя инструкцию данных SQL, вы получаете ответ от механизма базы данных о том, на какое количество строк она подействовала. Если вы используете интерактивный инструмент наподобие упомянутого ранее инструмента командной строки `mysql`, то в ответе вы получите информацию о том, сколько строк

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

Если вы используете с одним из упоминавшихся ранее наборов инструментов процедурный язык, то этот инструментарий будет включать вызов для запроса такой информации о количестве строк после выполнения вашей инструкции данных SQL. В общем случае рекомендуется проверить эту информацию, чтобы убедиться, что ваша инструкция не сделала ничего неожиданного (например, если вы забудете добавить предложение `where` в инструкцию `delete`, то удалите все строки в таблице!).

Что такое MySQL

Реляционные базы данных доступны коммерчески уже более трех десятилетий. Вот некоторые из наиболее зрелых и популярных коммерческих продуктов:

- Oracle Database от Oracle Corporation;
- SQL Server от Microsoft;
- DB2 Universal Database от IBM.

Все эти серверы баз данных делают примерно одно и то же, хотя некоторые из них лучше подходят для работы с очень большими базами данных или с базами данных с очень высокой пропускной способностью. У других лучше выполняется обработка объектов, очень больших файлов или XML-документов и т.д. Кроме того, все эти серверы довольно точно соответствуют последнему стандарту ANSI SQL. Следование стандарту — всегда хорошо, и я хочу показать вам, как писать инструкции SQL, которые будут работать на любой из этих платформ с минимальными изменениями или вовсе без них.

Последние два десятилетия с целью создания жизнеспособной альтернативы коммерческим серверам баз данных большую активность проявляло сообщество разработчиков программного обеспечения с открытым кодом. Двумя наиболее часто используемыми серверами баз данных с открытым исходным кодом являются PostgreSQL и MySQL. Сервер MySQL бесплатен и к тому же чрезвычайно прост в загрузке и установке. По этим причинам я решил, что все примеры для данной книги должны выполняться с базой данных MySQL.

(версия 8.0) и что для форматирования результатов запроса может использоваться инструмент командной строки `mysql`. Даже если вы уже пользуетесь другим сервером и никогда не планируете работать с MySQL, я настоятельно рекомендую вам установить последнюю версию сервера MySQL, загрузить образец схемы и данных и поэкспериментировать с данными и примерами из этой книги.

Однако помните о следующем предостережении:

Это не книга о реализации SQL в MySQL!

Данная книга предназначена для того, чтобы научить вас создавать инструкции SQL, которые без изменений будут выполняться на MySQL, а также будут работать с последними выпусками Oracle Database, DB2 и SQL Server с небольшими изменениями или без них.

Отказ от SQL

За десятилетие между вторым и третьим изданиями этой книги в мире баз данных многое изменилось. Хотя реляционные базы данных по-прежнему широко используются и будут использоваться еще некоторое время, для удовлетворения потребностей таких компаний, как Amazon и Google, появились новые технологии баз данных. Эти технологии включают Hadoop, Spark, NoSQL и NewSQL, которые представляют собой распределенные масштабируемые системы, обычно развертываемые на кластерах стандартных серверов. Подробное изучение этих технологий выходит за рамки данной книги, но все они имеют нечто общее с реляционными базами данных: SQL.

Поскольку организации часто хранят данные с использованием нескольких технологий, необходимы службы, которые в состоянии охватить несколько баз данных. Например, в отчете может потребоваться объединить данные, хранящиеся в файлах Oracle, Hadoop, JSON, CSV и файлах журналов Unix. Для решения этой задачи было создано новое поколение инструментов, и одним из наиболее многообещающих из них является Apache Drill — механизм запросов с открытым исходным кодом, который позволяет пользователям писать запросы, могущие получить доступ к данным, хранящимся практически в любой базе данных или файловой системе. Мы рассмотрим Apache Drill в главе 18, “SQL и большие данные”.

Что дальше

Общая цель следующих четырех глав — познакомить вас с инструкциями данных SQL, с особым акцентом на трех основных предложениях инструкции `select`. Кроме того, вы увидите множество примеров, в которых используется схема `Sakila` (представленная в следующей главе), которая будет применяться для всех примеров в книге. Я надеюсь, что знакомство с единственной базой данных позволит вам добраться до сути примера без того, чтобы постоянно останавливаться и проверять используемые таблицы. Если для вас окажется немного утомительной работа с одним и тем же набором таблиц, не стесняйтесь дополнить образец базы данных новыми таблицами или создать собственную базу данных, с которой можно будет экспериментировать.

После того как вы хорошо усвоите основы, в оставшихся главах книги мы углубимся в дополнительные концепции, большинство из которых не зависят одна от другой. Таким образом, если вы обнаружите, что запутались, то вы всегда сможете двигаться дальше, а позже вернуться, чтобы еще раз прочесть непонятную главу. Изучив эту книгу и проработав все примеры, вы будете на пути к тому, чтобы стать опытным практиком SQL.

Для читателей, желающих получить больше информации о реляционных базах данных, истории компьютеризированных СУБД и языке SQL, чем приведено в этом кратком введении, представляю несколько ресурсов, на которые стоит обратить внимание.

- C.J. Date. *Database in Depth: Relational Theory for Practitioners* (O'Reilly)
- К.Дж. Дейт. *Введение в системы баз данных*, 8-е изд. (пер. с англ., ООО “Диалектика”, 2020)
- C.J. Date *The Database Relational Model: A Retrospective Review and Analysis* (Addison-Wesley)
- Статья в Wikipedia *Database Management System* (<https://oreil.ly/sj2xR>)

Создание и наполнение базы данных

В этой главе представлена информация, необходимая для создания вашей первой базы данных, а также таблиц и связанных данных, используемых в примерах в этой книге. Вы также узнаете о различных типах данных и научитесь с их помощью создавать таблицы. Поскольку примеры в этой книге выполняются для базы данных MySQL, в этой главе имеется определенный уклон в сторону функций и синтаксиса MySQL, но большинство описываемых концепций применимо к любому серверу.

Создание базы данных MySQL

Если вы хотите иметь возможность экспериментировать с данными, использованными в примерах в этой книге, у вас есть два варианта.

- Загрузите и установите сервер MySQL версии 8.0 (или новее) и загрузите образец базы данных Sakila по адресу <https://dev.mysql.com/doc/index-other.html>.
- Перейдите по адресу <https://www.katacoda.com/mysql-db-sandbox/scenarios/mysql-sandbox>, чтобы получить доступ к “песочнице” MySQL, в которой образец базы данных Sakila загружен в экземпляр MySQL. Вам необходимо создать (бесплатную) учетную запись Katacoda и щелкнуть на кнопке Start Scenario (Начать сценарий).

Если вы выберете второй вариант, то, как только вы запустите сценарий, сервер MySQL будет установлен и запущен, а затем будут загружены схема и данные Sakila. Когда все будет готово, появится стандартное приглашение `mysql>`, и вы сможете начать выполнять запросы к базе данных. Это, безусловно, самый простой вариант, и я ожидаю, что большинство читателей

выберет именно его. Если вы так и поступили — можете сразу переходить к следующему разделу.

Если же вы предпочитаете иметь собственную копию данных и хотите, чтобы любые вносимые вами изменения были постоянными, или если вы просто заинтересованы в установке сервера MySQL на своем компьютере, то можете предпочтеть первый вариант. Вы также можете выбрать использование сервера MySQL, размещенного в такой среде, как Amazon Web Services или Google Cloud. В любом случае вам нужно будет выполнить установку и настройку самостоятельно, поскольку эта тема выходит за рамки данной книги. Как только ваша база данных станет доступной, вам нужно будет выполнить несколько шагов, чтобы загрузить образец базы данных Sakila.

Сначала нужно будет запустить клиент командной строки mysql и указать пароль, а затем выполнить следующие шаги.

1. Перейти по адресу <https://dev.mysql.com/doc/index-other.html> и загрузить файлы базы данных Sakila из раздела Example Databases.
2. Поместить загруженные файлы в свой каталог наподобие C:\temp\sakila-db (он использован и в следующих двух шагах; просто замените его своим реальным каталогом).
3. Введите source c:\temp\sakila-db\sakila-schema.sql; и нажмите <Enter>.
4. Введите source c:\temp\sakila-db\sakila-data.sql; и нажмите <Enter>.

Теперь у вас должна быть рабочая база данных, заполненная всеми данными, необходимыми для примеров в этой книге.



Образец базы данных Sakila предоставляется MySQL и имеет лицензию New BSD. Sakila содержит данные для фиктивной компании по прокату фильмов и включает в себя различные таблицы с информацией об инвентаре, фильмах, клиентах и др. Реальные пункты проката фильмов в основном ушли в прошлое, так что при небольшой фантазии вы можете переименовать свою компанию в компанию по потоковому воспроизведению фильмов и переделать соответственно базу данных, но примеры в этой книге соответствуют оригинальной базе данных.

Использование инструмента командной строки mysql

Если только вы не используете временный сеанс работы с базой данных (второй вариант работы с базой данных в предыдущем разделе), вам нужно запустить инструмент командной строки mysql, чтобы взаимодействовать с базой данных. Для этого вам нужно будет открыть оболочку Windows или Unix и запустить утилиту mysql. Например, если вы входите в систему, используя учетную запись root, выполните следующее:

```
mysql -u root -p;
```

Затем вас попросят ввести пароль, после чего вы увидите приглашение mysql>. Чтобы увидеть все доступные базы данных, можно использовать следующую команду:

```
mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| mysql          |
| performance_schema |
| sakila         |
| sys            |
+-----+
5 rows in set (0.01 sec)
```

Поскольку вы будете использовать базу данных Sakila, вам нужно указать ее с помощью команды use:

```
mysql> use sakila;
Database changed
```

Всякий раз, вызывая инструмент командной строки mysql, вы можете указать как имя пользователя, так и используемую базу данных, как показано ниже:

```
mysql -u root -p sakila;
```

Это избавит вас от необходимости набирать use sakila; при каждом запуске инструмента mysql. Теперь, когда вы установили сеанс и указали базу данных, можете выполнять инструкции SQL и просматривать результаты. Например, если вы хотите узнать текущие дату и время, можете ввести следующий запрос:

```
mysql> SELECT now();
+-----+
| now()        |
+-----+
```

```
+-----+
| 2019-04-04 20:44:26 |
+-----+
1 row in set (0.01 sec)
```

Функция now() — это встроенная функция MySQL, которая возвращает текущие дату и время. Как видите, инструмент командной строки mysql форматирует результаты ваших запросов внутри прямоугольника, ограниченного символами +, - и |. После того как результаты исчерпаны (в данном случае имеется только одна строка результатов), инструмент командной строки mysql показывает, сколько строк было возвращено и сколько времени потребовалось для выполнения инструкции SQL.

Об отсутствии предложения from

К некоторым серверам баз данных вы не сможете отправить запрос без предложения from, в котором указана хотя бы одна таблица. Одним из таких серверов является Oracle Database. Для случаев, когда вам нужно только вызвать функцию, Oracle предоставляет таблицу с именем dual, которая состоит из единственного столбца с именем dummy, который содержит единственную строку данных. Для совместимости с Oracle Database MySQL также предоставляет таблицу dual. Таким образом, предыдущий запрос для определения текущей даты и времени можно было бы записать как

```
mysql> SELECT now()
       FROM dual;
+-----+
| now()           |
+-----+
| 2019-04-04 20:44:26 |
+-----+
1 row in set (0.01 sec)
```

Если вы не используете Oracle и вам не нужна совместимость с Oracle, можете просто игнорировать таблицу dual и использовать только select (без from).

Завершив работу с инструментом командной строки mysql, просто введите quit; или exit;, чтобы вернуться в командную оболочку Unix или Windows.

Типы данных MySQL

В целом все популярные серверы баз данных могут хранить одни и те же типы данных, такие как строки, даты и числа. Обычно они различаются

поддержкой специализированных типов данных, таких как документы XML и JSON, или пространственных данных. Поскольку это вводная книга по SQL и поскольку 98% столбцов, с которыми вы столкнетесь, будут простыми типами данных, в этой главе рассматриваются только символьные данные, даты (также известные как временные данные) и числовые данные. Использование SQL для запроса документов JSON будет рассмотрено в главе 18, “SQL и большие данные”.

Символьные данные

Символьные данные могут храниться как строки фиксированной или переменной длины; разница в том, что строки фиксированной длины дополняются пробелами справа и всегда используют одинаковое количество байтов, а строки переменной длины не дополняются справа пробелами и не всегда занимают одинаковое количество байтов. При определении символьного столбца вы должны указать максимальный размер строки, которая может храниться в столбце. Например, чтобы хранить строки длиной до 20 символов, можно использовать любое из следующих определений:

```
char(20)    /* Постоянная длина */  
varchar(20) /* Переменная длина */
```

Максимальная длина столбцов `char` в настоящее время равна 255 байтов, тогда как столбцы `varchar` могут иметь размер до 65 535 байтов. Если вам нужно хранить более длинные строки (например, электронные письма, XML-документы и т.д.), можно использовать один из текстовых типов (`mediumtext` и `longtext`), о которых я расскажу позже в этом разделе. В общем случае вы должны использовать тип `char`, когда все строки, которые должны храниться в столбце, имеют одинаковую длину (например, аббревиатуры состояний), и тип `varchar`, когда строки, которые должны храниться в столбце, имеют разную длину. И `char`, и `varchar` одинаково используются на всех основных серверах баз данных.



Исключением является использование `varchar` в Oracle Database. Пользователи Oracle при определении столбцов символов переменной длины должны использовать тип `varchar2`.

Наборы символов

В языках, использующих латинский алфавит, например в английском языке, имеется достаточно малое количество символов, при котором для

хранения каждого символа достаточно одного байта. Другие языки, например японский и корейский, содержат большое количество символов, так что для каждого символа требуется несколько байтов памяти. Поэтому такие наборы символов называются *многобайтовыми наборами символов*.

MySQL может хранить данные с использованием различных наборов символов, как однобайтовых, так и многобайтовых. Чтобы просмотреть поддерживаемые наборы символов на своем сервере, можно использовать команду show, как показано в следующем примере:

```
mysql> SHOW CHARACTER SET;
```

Charset	Description	Default collation	Maxlen
armSCII8	ARMSCII-8 Armenian	armSCII8_general_ci	1
ascii	US ASCII	ascii_general_ci	1
big5	Big5 Traditional Chinese	big5_chinese_ci	2
binary	Binary pseudo charset	binary	1
cp1250	Windows Central European	cp1250_general_ci	1
cp1251	Windows Cyrillic	cp1251_general_ci	1
cp1256	Windows Arabic	cp1256_general_ci	1
cp1257	Windows Baltic	cp1257_general_ci	1
cp850	DOS West European	cp850_general_ci	1
cp852	DOS Central European	cp852_general_ci	1
cp866	DOS Russian	cp866_general_ci	1
cp932	SJIS for Windows Japanese	cp932_japanese_ci	2
dec8	DEC West European	dec8_swedish_ci	1
eucjpm	UJIS for Windows Japanese	eucjpm_s_japanese_ci	3
euckr	EUC-KR Korean	euckr_korean_ci	2
gb18030	China National Standard		
	GB18030	gb18030_chinese_ci	4
gb2312	GB2312 Simplified Chinese	gb2312_chinese_ci	2
gbk	GBK Simplified Chinese	gbk_chinese_ci	2
geostd8	GEOSTD8 Georgian	geostd8_general_ci	1
greek	ISO 8859-7 Greek	greek_general_ci	1
hebrew	ISO 8859-8 Hebrew	hebrew_general_ci	1
hp8	IHP West European	hp8_english_ci	1
keybcs2	DOS Kamenicky Czech-Slovak	keybcs2_general_ci	1
koi8r	KOI8-R Relcom Russian	koi8r_general_ci	1
koi8u	KOI8-U Ukrainian	koi8u_general_ci	1
latin1	cp1252 West European	latin1_swedish_ci	1
latin2	ISO 8859-2 Central European	latin2_general_ci	1
latin5	ISO 8859-9 Turkish	latin5_turkish_ci	1
latin7	ISO 8859-13 Baltic	latin7_general_ci	1
macce	Mac Central European	macce_general_ci	1
macroman	Mac West European	macroman_general_ci	1
sjis	Shift-JIS Japanese	sjis_japanese_ci	2
swe7	7bit Swedish	swe7_swedish_ci	1
tis620	TIS620 Thai	tis620_thai_ci	1

```

|ucs2      |UCS-2 Unicode
|ujis      |EUC-JP Japanese
|utf16     |UTF-16 Unicode
|utf16le   |UTF-16LE Unicode
|utf32     |UTF-32 Unicode
|utf8      |UTF-8 Unicode
|utf8mb4   |UTF-8 Unicode
+-----+
41 rows in set (0.00 sec)

```

Если значение в четвертом столбце, maxlen, больше 1, то набор символов является многобайтовым.

В предыдущих версиях сервера MySQL в качестве набора символов по умолчанию автоматически выбирался набор символов latin1, но версия 8 по умолчанию использует utf8mb4. Однако вы можете выбрать использование разных наборов символов для каждого символьного столбца в своей базе данных и даже хранить разные наборы символов в одной таблице. Чтобы при определении столбца выбрать набор символов, отличный от значения по умолчанию, просто укажите один из поддерживаемых наборов символов после определения типа, например:

```
varchar(20) character set latin1
```

В случае MySQL вы можете установить набор символов по умолчанию для всей базы данных:

```
create database european_sales character set latin1;
```

Здесь я не могу разместить больше информации о наборах символов, чем допустимо для введения в SQL, но на самом деле интернационализация — это очень большая тема. Если вы планируете иметь дело с несколькими (или с незнакомыми) наборами символов, можете обратиться к такой книге, как Jukka Korpela. *Unicode Explained: Internationalize Documents, Programs, and Web Sites* (O'Reilly).

Текстовые данные

Для того чтобы хранить данные, которые могут превышать ограничение в 64 Кбайта для столбцов varchar, следует использовать один из текстовых типов. В табл. 2.1 показаны доступные текстовые типы и их максимальные размеры.

Таблица 2.1. Текстовые типы MySQL

Текстовый тип	Максимальное количество байтов
tinytext	255
text	65 535
mediumtext	16 777 215
longtext	4 294 967 295

Выбирая для использования один из текстовых типов, необходимо учитывать следующее.

- Если данные, загружаемые в текстовый столбец, превышают максимальный размер для этого типа, они будут усечены.
- Конечные пробелы при загрузке данных в столбец не удаляются.
- При сортировке или группировке текстовых столбцов используются только первые 1024 байта данных, хотя при необходимости этот предел может быть увеличен.
- Различные текстовые типы уникальны для MySQL. SQL Server имеет единственный текстовый тип для больших символьных данных, в то время как DB2 и Oracle используют тип данных, именуемый clob (Character Large Object — большой символьный объект).
- Теперь, когда MySQL позволяет использовать до 65 535 байт для столбцов varchar (в версии 4 было ограничение в 255 байтов), нет никакой необходимости использовать типы tinytext и text.

Если вы создаете столбец для ввода данных произвольного вида, например столбец примечаний для хранения данных о взаимодействии клиентов с отделом обслуживания вашей компании, то varchar, вероятно, будет адекватным выбором. Однако, если вы храните документы, следует выбрать тип mediumtext или longtext.



Oracle Database позволяет использовать до 2000 байтов для столбцов типа char и 4000 байтов — для столбцов типа varchar2. Для больших документов вы можете использовать тип clob. SQL Server может обрабатывать до 8000 байтов как для char, так и для varchar, но вы можете хранить до 2 Гбайтов данных в столбце, определенном как varchar (max).

Числовые данные

Хотя может показаться разумным иметь единственный числовой тип данных, на самом деле существует несколько различных числовых типов данных, которые отражают различные способы использования чисел, как показано далее.

Столбец, показывающий, отправлен ли заказ клиента.

Этот тип столбца, именуемый **логическим** (Boolean), будет содержать значение 0, чтобы указать ложь, и 1 — для истины.

Сгенерированный системой первичный ключ для таблицы транзакций.

Эти данные обычно начинаются с 1 и увеличиваются на единицу для каждого очередного значения до потенциально очень большого числа.

Номер позиции электронной корзины покупателя.

Значения для этого типа столбца — положительные целые числа от 1 и до, возможно, 200 (для шопоголиков).

Позиционные данные для сверлильного станка для печатных плат.

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

Для обработки этих (и других) типов данных MySQL имеет несколько различных числовых типов данных. Чаще всего используются числовые типы, используемые для хранения целочисленных значений, или **целых чисел**. При указании одного из этих типов вы также можете указать, что данные будут без знака, что сообщает серверу, что все данные, хранящиеся в столбце, будут не меньше нуля. В табл. 2.2 показаны пять различных типов данных, используемых для хранения целых чисел.

Таблица 2.2. Целочисленные типы MySQL

Тип	Знаковый диапазон	Беззнаковый диапазон
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 -2^{63} до $2^{63}-1$	От 0 до $2^{64}-1$

Когда вы создаете столбец с использованием одного из целочисленных типов, MySQL выделяет соответствующий объем пространства для хранения данных, который варьируется от одного байта для `tinyint` до восьми байтов для `bigint`. Следовательно, вы должны попытаться выбрать тип, который будет достаточно большим, чтобы вместить самое большое число, которое может быть в вашей задаче, при этом без излишних затрат места для хранения.

Для чисел с плавающей точкой (например, 3.1415927) вы можете выбирать числовые типы, показанные в табл. 2.3.

Таблица 2.3. Типы с плавающей точкой MySQL

Тип	Числовой диапазон
<code>float (p, s)</code>	От $-3.402823466E+38$ до $-1.175494351E-38$ и от $1.175494351E-38$ до $3.402823466E+38$
<code>double (p, s)</code>	От $-1.7976931348623157E+308$ до $-2.2250738585072014E-308$ и от $2.2250738585072014E-308$ до $1.7976931348623157E+308$

При использовании типа с плавающей точкой можно указать его *точность* (precision — общее количество допустимых цифр слева и справа от десятичной точки) и *масштаб* (scale — количество допустимых цифр справа от десятичной точки), но это необязательно. Эти значения представлены в табл. 2.3 как *p* и *s*. Если вы укажете точность и масштаб столбца с плавающей точкой, помните, что данные, хранящиеся в столбце, будут округлены, если количество цифр превышает указанный масштаб и/или точность столбца. Например, столбец, определенный как `float (4, 2)`, будет хранить в общей сложности четыре цифры, две слева от десятичной точки и две справа. Следовательно, такой столбец отлично справится с числами 27,44 и 8,19, но число 17,8675 будет округлено до 17,87, а попытка сохранить число 178,375 в столбец `float (4, 2)` вызовет ошибку.

Как и целочисленные типы, столбцы с плавающей запятой могут быть определены как беззнаковые, но это указание всего лишь предотвращает сохранение в столбце отрицательных чисел и не влияет на диапазон данных, которые могут в нем храниться.

Временные данные

Вы почти наверняка будете работать не только со строками и числами, но и с информацией о датах и/или времени. Этот тип данных называется *временными*, а некоторые примеры временных данных в базе данных включают следующее.

- Дата в будущем, когда ожидается конкретное событие, например отправка заказ клиенту.
- Дата отправки заказа клиента.
- Дата и время, когда пользователь изменил определенную строку в таблице.
- Дата рождения сотрудника.
- Год, соответствующий строке в таблице `yearly_sales` данных склада.
- Время, необходимое для выполнения определенных работ на автомобильном конвейере.

MySQL включает типы данных для обработки всех этих ситуаций. В табл. 2.4 показаны временные типы данных, поддерживаемые MySQL.

Таблица 2.4. Временные типы MySQL

Тип	Формат по умолчанию	Допустимые значения
<code>date</code>	<code>YYYY-MM-DD</code>	От 1000-01-01 до 9999-12-31
<code>datetime</code>	<code>YYYY-MM-DD HH:MI:SS</code>	От 1000-01-01 00:00:00.000000 до 9999-12-31 23:59:59.999999
<code>timestamp</code>	<code>YYYY-MM-DD HH:MI:SS</code>	От 1970-01-01 00:00:00.000000 до 2038-01-18 22:14:07.999999
<code>year</code>	<code>YYYY</code>	От 1901 до 2155
<code>time</code>	<code>HHH:MI:SSS</code>	От -838:59:59.000000 до 838:59:59.000000

Хотя серверы баз данных хранят временные данные различными способами, назначение строки формата (второй столбец табл. 2.4) — показать, как данные будут представлены при выборке, а также как должна быть построена строка даты при вставке или обновлении временного столбца. Таким образом, если вы хотите внести дату “23 марта 2020 года” в столбец `date`, используя формат по умолчанию `YYY-MM-DD`, вы должны использовать строку `2020-03-23`. В главе 7, “Генерация, обработка и преобразование данных”, подробно исследуется, как создаются и отображаются временные данные.

Типы `datetime`, `timestamp` и `time` также допускают до 6 десятичных разрядов дробной части секунд (микросекунды). При определении столбцов с использованием одного из этих типов данных вы можете указать значение от 0 до 6; например, `datetime(2)` позволит включать в ваши значения времени сотые доли секунды.



Серверы баз данных допускают различные диапазоны дат для временных столбцов. Oracle Database принимает даты от 4712 года до н.э. до 9999 года н.э., в то время как SQL Server обрабатывает только даты от 1753 года н.э. до 9999 года н.э. (если только вы не используете тип данных datetime2 SQL Server 2008, который допускает даты в диапазоне от 1 года н.э. до 9999 года н.э.). MySQL находится между Oracle и SQL Server и может хранить даты с 1000 года н.э. до 9999 года н.э. Хотя для большинства систем, отслеживающих текущие и будущие события, это может не иметь никакого значения, следует помнить об этом при хранении исторических дат.

В табл. 2.5 описаны различные компоненты форматов даты, представленных в табл. 2.4.

Таблица 2.5. Компоненты формата даты

Компонент	Определение	Диапазон
YYYY	Год, включая век	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	Минуты	От 00 до 59
SS	Секунды	От 00 до 59

Вот как можно использовать различные временные типы для реализации приведенных выше примеров.

- Столбцы для хранения ожидаемой будущей даты отгрузки заказа клиента или дня рождения сотрудника могут использовать тип date, так как нереально запланировать будущую отгрузку с точностью до секунды и необязательно знать, в какое именно время человек родился.
- Столбец для хранения информации о том, когда заказ клиента был отправлен на самом деле, может использовать тип datetime, так как важно отслеживать не только дату отправки, но и ее время.
- Столбец, который отслеживает, когда пользователь в последний раз изменял определенную строку в таблице, может использовать тип timestamp. Тип timestamp содержит ту же информацию, что и тип datetime (год, месяц, день, час, минута, секунда), но столбец timestamp будет автоматически заполняться текущим значением даты/времени.

времени сервером MySQL при добавлении строки в таблицу или при ее последующем изменении.

- Столбец, содержащий только данные о году, может использовать тип `year`.
- Столбцы, содержащие данные о продолжительности времени, необходимого для выполнения задания, могут использовать тип `time`. Для этого типа данных компонент даты не нужен и только запускает, поскольку нас интересует только количество часов/минут/секунд, необходимых для выполнения задания. Эта информация может быть получена с использованием двух столбцов `datetime` (один — для даты/времени начала выполнения задания, другой — для даты/времени его завершения) путем вычитания одного из другого, но проще использовать единственный столбец `time`.

В главе 7, “Генерация, обработка и преобразование данных”, рассказывается, как работать с каждым из этих временных типов данных.

Создание таблицы

Теперь, когда у вас есть четкое представление о том, какие типы данных могут храниться в базе данных MySQL, пора рассмотреть, как использовать эти типы в определениях таблиц. Начнем с определения таблицы для хранения информации о человеке.

Шаг 1. Проектирование

Хороший способ начать разработку таблицы — провести мозговой штурм, чтобы увидеть, какую информацию было бы полезно в нее включить. Вот что я придумал, подумав вкратце о типах информации, которые описывают человека.

- Имя
- Цвет глаз
- Дата рождения
- Адрес
- Любимые блюда

Это, конечно, не исчерпывающий список, но на данный момент его достаточно. Следующий шаг состоит в присвоении имен столбцам и выборе типов данных. В табл. 2.6 показана моя первая попытка.

Таблица 2.6. Таблица person (первая попытка)

Столбец	Тип	Допустимые значения
name	varchar(40)	
eye_color	char(2)	BL, BR, GR
birth_date	date	
address	varchar(100)	
favorite_foods	varchar(200)	

Столбцы name, address и favorite_foods относятся к типу varchar и допускают ввод данных в свободной форме. Столбец eye_color позволяет использовать два символа, которые должны представлять собой только BR, BL или GR. Столбец birth_date имеет тип date, поскольку компонент времени не нужен.

Шаг 2. Уточнение

В главе 1, “Небольшая предыстория”, вы познакомились с понятием нормализации, которая представляет собой процесс обеспечения отсутствия дубликатов (кроме внешних ключей) или составных столбцов в дизайне базы данных. При втором взгляде на столбцы в таблице person возникают следующие вопросы.

- Столбец name фактически представляет собой составной объект, состоящий из имени и фамилии.
- Поскольку несколько человек могут иметь одинаковое имя, цвет глаз, дату рождения и так далее, в таблице person нет столбцов, гарантирующих уникальность.
- Столбец address также является составным объектом, состоящим из улицы, города, штата/области, страны и почтового индекса.
- Столбец favorite_foods — это список, содержащий нуль, одно или несколько независимых блюд. Лучше всего для этих данных создать отдельную таблицу, в которую будет входить внешний ключ к таблице person, чтобы вы знали, к какому человеку можно отнести то или иное конкретное блюдо.

С учетом этих замечаний в табл. 2.7 приведена нормализованная версия таблицы person.

Таблица 2.7. Таблица person (вторая попытка)

Столбец	Тип	Допустимые значения
person_id	smallint (unsigned)	
first_name	varchar(20)	
last_name	varchar(20)	
eye_color	char(2)	BL, BR, GR
birth_date	date	
street	varchar(30)	
city	varchar(20)	
state	varchar(20)	
country	varchar(20)	
postal_code	varchar(20)	

Теперь, когда у таблицы person есть первичный ключ (person_id), чтобы гарантировать уникальность, следующим шагом будет создание таблицы favorite_food, которая включает внешний ключ к таблице person. Результат показан в табл. 2.8.

Таблица 2.8. Таблица favorite_food

Столбец	Тип
person_id	smallint (unsigned)
food	varchar(20)

Столбцы person_id и food составляют первичный ключ таблицы favorite_food, а столбец person_id, кроме того, является внешним ключом для таблицы person.

Чего достаточно?

Перемещение столбца favorite_food из таблицы person было, определенно, хорошей идеей, но все ли мы сделали? Что произойдет, например, если один человек укажет в качестве любимой еды “макароны”, в то время как другой — “спагетти”? Это одно и то же? Чтобы предотвратить эту проблему, вы можете решить, что хотите, чтобы люди выбирали свои любимые продукты из списка вариантов, и в этом случае вы должны создать таблицу food со столбцами food_id и food_name, а затем изменить таблицу favorite_food так, чтобы она содержала внешний ключ к таблице food. Хотя такой дизайн будет полностью нормализован, вы можете решить, что будете просто хранить введенные пользователем значения, и в этом случае вы можете оставить таблицу такой, как она есть.

Шаг 3. Построение инструкции схемы SQL

Теперь, когда дизайн двух таблиц, содержащих информацию о людях и их любимых блюдах, готов, следующий шаг состоит в генерации инструкций SQL для создания таблиц в базе данных. Вот инструкция для создания таблицы people:

```
CREATE TABLE person
(person_id SMALLINT UNSIGNED,
 fname VARCHAR(20),
 lname VARCHAR(20),
 eye_color CHAR(2),
 birth_date DATE,
 street VARCHAR(30),
 city VARCHAR(20),
 state VARCHAR(20),
 country VARCHAR(20),
 postal_code VARCHAR(20),
 CONSTRAINT pk_person PRIMARY KEY (person_id)
);
```

В этой инструкции все должно быть достаточно очевидным, за исключением последнего пункта; когда вы определяете свою таблицу, вам нужно сообщить серверу базы данных, какой столбец (или столбцы) будет служить первичным ключом для таблицы. Вы делаете это, создавая *ограничение* таблицы. В определение таблицы можно добавить несколько типов ограничений. Данное ограничение является *ограничением первичного ключа*. Он создается в столбце person_id и получает имя pk_person.

Говоря об ограничениях, следует упомянуть еще один тип ограничения, который был бы полезен для таблицы person. В табл. 2.6 я добавил третий столбец, чтобы показать допустимые значения для определенных столбцов (например, BR и BL для столбца eye_color). Еще один тип ограничения, имеющий *проверочным ограничением*, приводит к проверке допустимых значений для конкретного столбца. MySQL позволяет присоединить к столбцу проверочное ограничение:

```
eye_color CHAR(2) CHECK (eye_color IN ('BR','BL','GR')),
```

Хотя проверочные ограничения должным образом работают на большинстве серверов баз данных, сервер MySQL позволяет определять проверочные ограничения, но не применяет их. Однако MySQL предоставляет еще один тип символьных данных, именуемый enum, который вводит проверочное ограничение в определение типа данных. Вот как это выглядит в определении столбца eye_color:

```
eye_color ENUM('BR','BL','GR'),
```

А вот как выглядит определение таблицы person с типом данных enum для столбца eye_color:

```
CREATE TABLE person
(person_id SMALLINT UNSIGNED,
 fname VARCHAR(20),
 lname VARCHAR(20),
 eye_color ENUM('BR','BL','GR'),
 birth_date DATE,
 street VARCHAR(30),
 city VARCHAR(20),
 state VARCHAR(20),
 country VARCHAR(20),
 postal_code VARCHAR(20),
 CONSTRAINT pk_person PRIMARY KEY (person_id)
);
```

Позже в этой главе вы увидите, что произойдет, если вы попытаетесь добавить в столбец данные, которые нарушают проверочное ограничение (или, в случае MySQL, его значения перечисления).

Теперь вы готовы выполнить инструкцию создания таблицы с помощью командной строки mysql. Вот как это выглядит:

```
mysql> CREATE TABLE person
-> (person_id SMALLINT UNSIGNED,
-> fname VARCHAR(20),
-> lname VARCHAR(20),
-> eye_color ENUM('BR','BL','GR'),
-> birth_date DATE,
-> street 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 (0.37 sec)
```

После выполнения инструкции create table сервер MySQL возвращает сообщение Query OK, 0 rows affected (Запрос успешен, затронуты 0 строк), что говорит о том, что в инструкции нет синтаксических ошибок.

Если вы хотите убедиться, что таблица person действительно существует, можете использовать команду describe (или для краткости — desc), чтобы посмотреть определение таблицы:

```
mysql> desc person;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| person_id | smallint unsigned | NO | PRI | NULL | |
| fname | varchar(20) | YES | | NULL | |
| lname | varchar(20) | YES | | NULL | |
| eye_color | enum('BR','BL','GR') | YES | | NULL | |
| birth_date | date | YES | | NULL | |
| street | 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 (0.00 sec)
```

Столбцы 1 и 2 выходных данных не требуют пояснений. Столбец 3 показывает, можно ли пропустить конкретный столбец при добавлении данных в таблицу. В данный момент я намеренно опустил эту тему из обсуждения, но мы подробно рассмотрим ее в главе 4, “Фильтрация”. В четвертом столбце указано, участвует ли данный столбец в каких-либо ключах (первичных или внешних); в нашем случае столбец `person_id` указан как первичный ключ. В пятом столбце показано значение по умолчанию, которым будет заполнен столбец, если его значение при добавлении данных в таблицу будет опущено. Шестой столбец показывает иную информацию, которая может иметь отношение к столбцу.

Что такое `NULL`

В некоторых случаях невозможно указать значение для определенного столбца таблицы. Например, при добавлении данных о новом заказе клиента столбец `ship_date` (дата отгрузки) не может быть определен. В этом случае говорят, что столбец *нулевой (null)* (обратите внимание, что я не говорю, что он *равен нулю!*), что указывает на отсутствие в нем значения. `NULL` используется в различных случаях, когда значение не может быть предоставлено из-за того, что оно *неизвестно*, представляет собой пустое множество или неприменимо в некотором конкретном случае.

При разработке таблицы вы можете указать, какие столбцы могут быть нулевыми (по умолчанию), а какие не могут (что указывается добавлением ключевых слов `not null` после определения типа).

Теперь, когда мы создали таблицу `person`, перейдем к созданию таблицы `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_fav_food_person_id FOREIGN KEY (person_id)
->      REFERENCES person (person_id)
->  );
Query OK, 0 rows affected (0.10 sec)
```

Все это выглядит очень похоже на инструкцию `create table` для таблицы `person`, но со следующими отличиями.

- Поскольку у человека может быть более одного любимого блюда (в первую очередь из-за этого и была создана эта таблица), таблице требуется больше, чем единственный столбец `person_id`, чтобы гарантировать уникальность записи в ней. Таким образом, эта таблица имеет составной ключ из двух столбцов: `person_id` и `food`.
- Таблица `favorite_food` содержит еще один тип ограничения, который называется *ограничением внешнего ключа*. Он ограничивает значения столбца `person_id` в `favorite_food` таким образом, что этот столбец может включать *только* те значения, которые имеются в таблице `person`. При таком ограничении я не смогу добавить в таблицу `favorite_food` строку, которая указывает, что некто со значением `person_id`, равным 27, любит пиццу, если в таблице `person` еще нет строки с записью, значение `person_id` которой равно 27.



Если вы забыли указать ограничение внешнего ключа при первоначальном создании таблицы, добавьте его позже с помощью инструкции `alter table`.

Вот что показывает инструкция `desc` после выполнения инструкции `create table`:

```
mysql> desc favorite_food;
+-----+-----+-----+-----+-----+
| Field | Type            | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| person_id | smallint unsigned | NO | PRI | NULL |          |
| food | varchar(20) | NO | PRI | NULL |          |
+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

Теперь, когда таблицы созданы, следующий логический шаг состоит в добавлении в них некоторых данных.

Заполнение и изменение таблиц

Создав таблицы `person` и `favorite_food`, мы можем приступить к изучению четырех инструкций данных SQL: `insert`, `update`, `delete` и `select`.

Добавление данных

Поскольку данных в таблицах `person` и `favorite_food` еще нет, первая из четырех инструкций данных SQL, которую мы изучим, будет инструкцией `insert`. У нее есть три основных компонента:

- имя таблицы, в которую нужно добавить данные;
- имена столбцов в таблице, которые необходимо заполнить;
- значения, которыми заполняются столбцы.

Вы не обязаны предоставлять данные для каждого столбца таблицы (если только все столбцы в таблице не определены как `not null`). В некоторых случаях тем столбцам, которые не включены в начальную инструкцию `insert`, значение будет присвоено позже с помощью инструкции `update`. В других случаях в некоторой строке данных столбец может так никогда и не получить значение (например, заказ клиента, который отменен до отправки; таким образом, никакое значение столбца `ship_date` не применимо).

Генерация данных числовых ключей

Прежде чем добавлять данные в таблицу `person`, было бы полезно обсудить, как генерируются значения для числовых первичных ключей. Помимо высасывания чисел из пальца, у нас есть несколько более разумных вариантов:

- найти самое большое значение в таблице и увеличить его;
- позволить серверу базы данных предоставить нам требуемое значение.

Хотя первый вариант может показаться вполне допустимым, в многопользовательской среде он оказывается проблематичным, поскольку два пользователя могут одновременно просматривать таблицу и генерировать одно и то же значение для первичного ключа. Вместо этого все серверы баз данных, имеющиеся на рынке в настоящее время, предоставляют безопасный и надежный метод создания цифровых ключей. На некоторых серверах, таких как `Oracle Database`, используется отдельный объект схемы (именуемый *последовательностью*); в случае же `MySQL` нужно просто включить функцию *автоинкремента* (автоматического увеличения) для столбца первичного ключа.

Обычно это делается при создании таблицы, но сейчас мы получаем возможность изучить другую инструкцию схемы SQL, `alter table`, которая используется для изменения определения существующей таблицы:

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

Эта инструкция, по существу, переопределяет столбец `person_id` в таблице `person`. Если теперь выполнить инструкцию `desc` для таблицы, можно увидеть автоинкремент в столбце “Extra” для `person_id`:

```
mysql> DESC person;
+-----+-----+-----+-----+-----+
| Field | Type            | Null | Key | Default | Extra          |
+-----+-----+-----+-----+-----+
| person_id | smallint(5) unsigned | NO   | PRI | NULL    | auto_increment |
| .         |                   |      |     |          |                |
| :         |                   |      |     |          |                |
| .         |                   |      |     |          |                |
```

Добавляя данные в таблицу `person`, вы просто предоставляеме `person_id` значение `null`, и MySQL самостоятельно заполнит столбец следующим доступным числом (по умолчанию MySQL для столбцов с автоматическим увеличением начинает отсчет с 1).

Инструкция `insert`

Теперь, когда все готово, пора заняться добавлением данных. Следующая инструкция создает строку в таблице `person` для Уильяма Тернера (William Turner):

```
mysql> INSERT INTO person
->   (person_id, fname, lname, eye_color, birth_date)
-> VALUES (null, 'William', 'Turner', 'BR', '1972-05-27');
Query OK, 1 row affected (0.22 sec)
```

Сообщение “`Query OK, 1 row affected`” означает, что синтаксис инструкции верен и в таблицу была добавлена одна строка (поскольку это была инструкция `insert`). Вы можете просмотреть только что добавленные в таблицу данные, выполнив инструкцию `select`:

```
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 сгенерировал для первичного ключа значение 1. Так как в таблице `person` есть только одна строка, я не указал, какая именно строка меня интересует, и просто получил все строки таблицы. Если бы в таблице было больше одной строки, я мог бы добавить предложение `where`, чтобы указать, что я хочу получить данные только для строки, имеющей значение 1 в столбце `person_id`:

```
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)
```

Хотя этот запрос указывает конкретное значение первичного ключа, вы можете использовать для поиска строк любой столбец таблицы, как показано в следующем запросе, который находит все строки со значением `Turner` в столбце `lname`:

```
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, стоит сказать еще пару слов о рассмотренной выше инструкции `insert`.

- Значения не были предоставлены ни для одного из столбцов адреса. Это нормально, поскольку для этих столбцов разрешены значения `null`.
- Значение, предоставленное для столбца `birth_date`, было строкой. Пока эта строка соответствует требуемому формату, показанному в табл. 2.4, MySQL сам преобразует строку в дату.
- Имена столбцов и указанные значения должны совпадать по количеству и типу. Если вы укажете семь столбцов и предоставите только шесть значений или если вы предоставите значения, которые не могут быть преобразованы в соответствующий тип данных для соответствующего столбца, вы получите сообщение об ошибке.

Уильям Тернер предоставил также информацию о трех своих любимых блюдах, так что вот три инструкции `insert` для сохранения его предпочтений в еде:

```
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, который извлекает любимые блюда Уильяма в алфавитном порядке с помощью предложения `order by`:

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

Предложение `order by` сообщает серверу, как следует сортировать данные, возвращаемые запросом. Без `order by` нет гарантии, что данные из таблицы будут извлекаться в каком-то определенном порядке.

Чтобы Уильям не чувствовал себя одиноким, выполним еще одну инструкцию `insert`, чтобы добавить в таблицу `person` Сьюзен Смит:

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

Поскольку Сьюзен любезно предоставила свой адрес, мы включили в запрос на пять столбцов больше, чем при добавлении данных Уильяма. Снова запросив таблицу, вы увидите, что строке Сьюзен в качестве значения первичного ключа 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)
```

Как получить информацию в виде XML

Всем, кто планирует работать с данными XML, будет приятно узнать, что большинство серверов баз данных предоставляют простой способ генерации выходных данных запроса в виде XML. Например, с MySQL вы можете использовать параметр `--xml` при вызове инструмента `mysql`, и весь ваш вывод будет автоматически отформатирован в виде XML. Вот как выглядят в указанном виде данные о любимой еде:

```
C:\database> mysql -u lrngsql -p --xml bank
Enter password: xxxxxx
Welcome to the MySQL Monitor...

Mysql> SELECT * FROM favorite_food;
<?xml version="1.0"?>

<resultset statement="select * from favorite_food"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<row>
  <field name="person_id">1</field>
  <field name="food">cookies</field>
</row>
<row>
  <field name="person_id">1</field>
  <field name="food">nachos</field>
</row>
<row>
  <field name="person_id">1</field>
  <field name="food">pizza</field>
</row>
</resultset>
3 rows in set (0.00 sec)
```

В SQL Server вам не нужно настраивать инструмент командной строки; надо просто добавить предложение `for xml` в конец вашего запроса:

```
SELECT * FROM favorite_food
FOR XML AUTO, ELEMENTS
```

Изменение данных

Когда мы добавляли в таблицу данные для Уильяма Тернера, в инструкцию `insert` не были включены данные для различных столбцов адреса. Следующая инструкция показывает, как эти столбцы можно заполнить позже с помощью инструкции `update`:

```
mysql> UPDATE person
->   SET street = '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 двухстрочным сообщением: `Rows matched: 1` сообщает, что условие в предложении `where` соответствует одной строке таблицы, а `Changed: 1` — что в таблице была изменена одна строка. Поскольку предложение `where` указывает первичный ключ строки Уильяма, произошло именно то, чего мы и ожидали.

В зависимости от условий в предложении `where` можно изменить несколько строк с помощью одной инструкции. Рассмотрим, например, что было бы, если бы ваше предложение `where` выглядело следующим образом:

```
WHERE person_id < 10
```

Поскольку и Уильям, и Сьюзен имеют значение `person_id`, меньшее 10, обе строки будут изменены. Если вы полностью опустите предложение `where`, ваша инструкция `update` изменит каждую строку таблицы.

Удаление данных

Похоже, что Уильям и Сьюзен не очень хорошо ладят между собой, так что одному из них нужно уйти. Поскольку Уильям был первым, информация о Сьюзен удаляется из таблицы с помощью инструкции `delete`:

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

И вновь для указания интересующей нас строки использован первичный ключ, поэтому из таблицы удалена ровно одна строка. Как и в случае с оператором `update`, можно удалить более одной строки — в зависимости

от условий в предложении `where` (если предложение `where` опустить, будут удалены все строки).

Когда хорошие инструкции становятся плохими

До сих пор все инструкции данных SQL, показанные в этой главе, были правильно сформированы и играли по правилам. Однако исходя из определений таблиц `person` и `favorite_food` имеется много способов ошибиться при добавлении и изменении данных. В этом разделе показаны некоторые из распространенных ошибок, с которыми вы можете столкнуться, и реакция на них сервера MySQL.

Не уникальный первичный ключ

Поскольку определения таблиц включают ограничение первичного ключа, MySQL будет следить за тем, чтобы в таблицах не было повторяющихся значений первичных ключей. Следующая инструкция пытается обойти функцию автоинкремента столбца `person_id` и создать еще одну строку в таблице `person` с `person_id`, равным 1:

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

Как видите, произошла ошибка, связанная с тем, что такое значение первичного ключа уже имеется в таблице. Однако ничто не мешает вам (по крайней мере, с текущей схемой) создать две строки с одинаковыми именами, адресами, датами рождения и так далее, если они имеют разные значения столбца `person_id`.

Несуществующий внешний ключ

Определение 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`, существуют в таблице `person`. Вот что произойдет, если вы попытаетесь создать строку, нарушающую это ограничение:

```
mysql> INSERT INTO favorite_food (person_id, food)
-> VALUES (999, 'lasagna');
ERROR 1452 (23000): Cannot add or update a child row:
a foreign key constraint fails ('sakila'.'favorite_food',
```

```
CONSTRAINT 'fk_fav_food_person_id' FOREIGN KEY  
('person_id') REFERENCES 'person' ('person_id'))
```

В этом случае таблица `favorite_food` считается *дочерней*, а таблица `person` — *родительской*, так как таблица `favorite_food` зависит от некоторых данных таблицы `person`. Если вы планируете вводить данные в обе таблицы, создайте строку в родительской таблице `person`, прежде чем вводить соответствующие данные в `favorite_food`.



Ограничения внешнего ключа применяются только в том случае, если ваши таблицы созданы с использованием механизма хранения InnoDB. Мы обсудим механизм хранения MySQL в главе 12, “Транзакции”.

Нарушения значений столбцов

Столбец `eye_color` в таблице `person` ограничен значениями BR для карих глаз, BL — для голубых и GR — для зеленых глаз. Если вы по ошибке попытаетесь использовать любое другое значение, то получите следующий ответ:

```
mysql> UPDATE person  
    -> SET eye_color = 'ZZ'  
    -> WHERE person_id = 1;  
ERROR 1265 (01000): Data truncated for column 'eye_color' at row 1
```

Сообщение об ошибке немножко сбивает с толку, но дает общее представление о том, что сервер недоволен значением, указанным для столбца `eye_color`.

Некорректное преобразование данных

Если вы создаете строку для заполнения столбца даты и эта строка не соответствует ожидаемому формату, вы получите еще одно сообщение об ошибке. Вот пример использования формата даты, который не соответствует формату даты по умолчанию YYYY-MM-DD:

```
mysql> UPDATE person  
    -> SET birth_date = 'DEC-21-1980'  
    -> WHERE person_id = 1;  
ERROR 1292 (22007): Incorrect date value: 'DEC-21-1980'  
for column 'birth_date' at row 1
```

В общем случае всегда рекомендуется явно указывать строку формата, а не полагаться на формат по умолчанию. Вот еще одна версия инструкции, в которой используется функция `str_to_date`, чтобы указать, какую строку формата использовать:

```
mysql> UPDATE person
-> SET birth_date = str_to_date('DEC-21-1980', '%b-%d-%Y')
-> WHERE person_id = 1;
Query OK, 1 row affected (0.12 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

Доволен не только сервер базы данных, но и сам Уильям (мы только что сделали его на восемь лет моложе безо всяких дорогостоящих косметических операций!).



Ранее в этой главе, рассматривая различные временные типы данных, я показывал строки форматирования дат, такие как YYYY-MM-DD. В то время как многие серверы баз данных используют этот стиль форматирования, MySQL использует для обозначения года из четырех цифр %Y. Вот еще несколько строк форматирования, которые могут понадобиться вам при преобразовании строк в дату и время в MySQL:

%a	Краткое имя дня недели – Sun, Mon, ...
%b	Краткое имя месяца – Jan, Feb, ...
%c	Числовое значение месяца (0..11)
%d	Числовое значение дня месяца (00..31)
%f	Число микросекунд (000000..999999)
%H	Час дня в 24-часовом формате (00..23)
%h	Час дня в 12-часовом формате (01..12)
%i	Минуты в часе (00..59)
%j	День года (001..366)
%M	Полное имя месяца (January..December)
%m	Числовое значение месяца
%p	AM или PM
%s	Число секунд (00..59)
%W	Полное имя дня недели (Sunday..Saturday)
%w	Числовое значение дня недели (0=Sunday..6=Saturday)
%Y	Значение года (четыре цифры)

База данных Sakila

В оставшейся части книги в большинстве примеров будет использоваться образец базы данных под названием “Sakila”, созданный добрыми людьми из MySQL. Эта база данных моделирует сеть магазинов по прокату DVD (которая несколько устарела, но может быть переделана в компанию, занимающуюся потоковым видео). Среди таблиц базы данных — customer, film, actor, payment, rental и category. Схема и образец данных должны были быть созданы при выполнении вами последних шагов в начале главы, где описывалась загрузка сервера MySQL и генерация образцов данных. Диаграммы

таблиц, их столбцов и взаимосвязей приведены в приложении А, “Схема базы данных Sakila”.

В табл. 2.9 показаны некоторые таблицы, используемые в схеме Sakila, а также их краткие описания.

Таблица 2.9. Определения схемы Sakila

Таблица	Описание
film	Уже вышедший фильм, который можно взять напрокат
actor	Персонаж фильма
customer	Клиент, берущий фильм напрокат
category	Жанр фильма
payment	Прокат фильма клиентом
language	Язык фильма
film_actor	Артист, играющий в фильме
inventory	Доступность фильма для проката

Не стесняйтесь экспериментировать с таблицами сколько угодно, включая добавление собственных таблиц для расширения бизнес-функций базы данных. Вы всегда можете удалить базу данных и воссоздать ее из загруженного файла, если хотите получить нетронутые исходные данные. Если вы используете временный сеанс, любые внесенные вами изменения будут потеряны, когда сеанс закроется. Поэтому вы можете сохранить сценарий своих изменений, чтобы иметь возможность воссоздать все внесенные вами изменения.

Чтобы увидеть таблицы, доступные в вашей базе данных, можете использовать команду `show tables`:

```
+-----+  
| Tables_in_sakila      |  
+-----+  
| actor                |  
| actor_info            |  
| address               |  
| category              |  
| city                  |  
| country               |  
| customer              |  
| customer_list          |  
| film                 |  
| film_actor             |  
| film_category          |  
| film_list              |  
| film_text              |  
| inventory              |  
| language               |
```

```
| nicer_but_slower_film_list |
| payment |
| rental |
| sales_by_film_category |
| sales_by_store |
| staff |
| staff_list |
| store |
+-----+
23 rows in set (0.02 sec)
```

Помимо 23 таблиц в схеме Sakila, ваш список таблиц может включать соз-данные в этой главе таблицы person и favorite_food. Они не будут использо-ваться в следующих главах, поэтому можете смело их удалить, выполнив следующие команды:

```
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)
```

Если вы хотите просмотреть столбцы в таблице, можете использовать ко-манду describe. Вот пример вывода describe для таблицы customer:

```
mysql> desc customer;
+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| customer_id | smallint(5) | NO | PRI | NULL | auto_increment |
| | unsigned | | | | |
| store_id | tinyint(3) | NO | MUL | NULL | |
| | unsigned | | | | |
| first_name | varchar(45) | NO | | NULL | |
| last_name | varchar(45) | NO | MUL | NULL | |
| email | varchar(50) | YES | | NULL | |
| address_id | smallint(5) | NO | MUL | NULL | |
| | unsigned | | | | |
| active | tinyint(1) | NO | | 1 | |
| create_date | datetime | NO | | NULL | |
| last_update | timestamp | YES | CURRENT_TIMESTAMP | DEFAULT_GENERATED on update CURRENT_TIMESTAMP |
```

Чем комфортнее вы будете чувствовать себя с примером базы данных, тем лучше вы поймете рассматриваемые примеры (а следовательно, и концепции) в следующих главах книги.

Запросы

До сих пор вы видели лишь несколько примеров запросов к базе данных (известных как инструкции `select`), разбросанных на протяжении первых двух глав. Пришло время более пристально взглянуть на различные части инструкции `select` и их взаимодействие. Изучив эту главу, вы будете иметь общее представление о том, как данные выбираются, соединяются, фильтруются, группируются и сортируются; более подробно эти темы будут рассмотрены в главах 4–10.

Механика запросов

Прежде чем переходить к инструкции `select`, было бы интересно посмотреть, как запросы выполняются сервером MySQL (или, если уж на то пошло, любым сервером базы данных). Если вы используете инструмент командной строки `mysql` (что, как я предполагаю, так и есть), то вы уже вошли на сервер MySQL, указав свое имя пользователя и пароль (а возможно, и имя хоста, если сервер MySQL работает на другом компьютере). Сервер уже проверил правильность вашего имени пользователя и пароля и создал для вас *подключение к базе данных*. Это подключение поддерживается приложением, которое его запросило (в данном случае это инструмент `mysql`), до тех пор, пока приложение не освободит подключение (в результате ввода команды `quit`) или сервер не закроет соединение при выключении. Каждому подключению к серверу MySQL назначается идентификатор, который отображается для вас при первом входе в систему:

```
Welcome to the MySQL monitor. Commands end with ; or \g.  
Your MySQL connection id is 11  
Server version: 8.0.15 MySQL Community Server - GPL  
Copyright (c) 2000, 2019, Oracle and/or its affiliates.  
All rights reserved. Oracle is a registered trademark of  
Oracle Corporation and/or its affiliates. Other names
```

may be trademarks of their respective owners.
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

В данном случае мой идентификатор подключения — 11. Эта информация может быть полезна для администратора вашей базы данных, если что-то пойдет не так (например, неправильно сформированный запрос, который выполняется часами), так что вы можете записать это значение.

Как только сервер проверит ваше имя пользователя и пароль и установит соединение, вы будете готовы выполнять запросы (и другие инструкции SQL). Каждый раз, когда запрос отправляется на сервер, сервер перед выполнением запроса проверяет следующее.

- Есть ли у вас разрешение на выполнение инструкции.
- Есть ли у вас разрешение на доступ к нужным данным.
- Правильность синтаксиса вашей инструкции.

Если ваша инструкция успешно проходит эти три теста, запрос передается *оптимизатору запросов*, задача которого — определить наиболее эффективный способ выполнения вашего запроса. Оптимизатор изучает, например, порядок соединения таблиц, указанных в конструкции `from`, и какие индексы доступны, а затем выбирает *план выполнения* запроса, который сервер и осуществляет при выполнении вашего запроса.



Как сервер базы данных выбирает план выполнения — увлекательная тема, и многие из вас захотят дополнительно ее изучить. Те, кто использует MySQL, могут прочитать книгу Барона Шварца (Baron Schwartz) и других *High Performance MySQL* (*Высокопроизводительный MySQL*, издательство O'Reilly). Помимо прочего, вы изучите, как создавать индексы, анализировать планы выполнения, влиять на оптимизатор с помощью подсказок в запросах и настраивать параметры запуска вашего сервера. Если вы работаете с Oracle Database или SQL Server, то примите к сведению, что по настройке этих СУБД доступны десятки книг.

Как только сервер завершает выполнение вашего запроса, вызывающему приложению (которым в нашем случае является инструмент `mysql`) возвращается *результатирующий набор*. Как я уже упоминал в главе 1, “Небольшая предыстория”, результатирующий набор — это просто еще одна таблица, содержащая строки и столбцы. Если ваш запрос не в состоянии выдать какие-либо

результаты, инструмент mysql покажет вам сообщение, которое вы найдете в конце следующего примера:

```
mysql> SELECT first_name, last_name  
-> FROM customer  
-> WHERE last_name = 'ZIEGLER';  
Empty set (0.02 sec)
```

Если запрос возвращает одну или несколько строк, инструмент mysql форматирует результаты, добавляя заголовки столбцов и строя рамки вокруг столбцов с помощью символов -, | и +, как показано в следующем примере:

```
mysql> SELECT *  
-> FROM category;  
+-----+-----+-----+  
| category_id | name | last_update |  
+-----+-----+-----+  
| 1 | Action | 2006-02-15 04:46:27 |  
| 2 | Animation | 2006-02-15 04:46:27 |  
| 3 | Children | 2006-02-15 04:46:27 |  
| 4 | Classics | 2006-02-15 04:46:27 |  
| 5 | Comedy | 2006-02-15 04:46:27 |  
| 6 | Documentary | 2006-02-15 04:46:27 |  
| 7 | Drama | 2006-02-15 04:46:27 |  
| 8 | Family | 2006-02-15 04:46:27 |  
| 9 | Foreign | 2006-02-15 04:46:27 |  
| 10 | Games | 2006-02-15 04:46:27 |  
| 11 | Horror | 2006-02-15 04:46:27 |  
| 12 | Music | 2006-02-15 04:46:27 |  
| 13 | New | 2006-02-15 04:46:27 |  
| 14 | Sci-Fi | 2006-02-15 04:46:27 |  
| 15 | Sports | 2006-02-15 04:46:27 |  
| 16 | Travel | 2006-02-15 04:46:27 |  
+-----+-----+-----+  
16 rows in set (0.02 sec)
```

Этот запрос возвращает все три столбца для всех строк таблицы category. После вывода последней строки данных mysql выводит сообщение о том, как много строк было возвращено (в данном случае — 16).

Части запроса

Инструкция select состоит из нескольких компонентов, или *предложений*. Хотя только одно из них является обязательным при использовании MySQL (а именно, предложение select), обычно в запрос включается по крайней мере два или три из шести доступных предложений. В табл. 3.1 показаны различные предложения и их назначение.

Таблица 3.1. Предложения запроса

Имя	Назначение
select	Определяет, какие столбцы следует включить в результирующий набор запроса
from	Определяет таблицы, из которых следует выбирать данные, а также таблицы, которые должны быть соединены
where	Отсеивает ненужные данные
group by	Используется для группировки строк по общим значениям столбцов
having	Отсеивает ненужные данные
order by	Сортирует строки окончательного результирующего набора по одному или нескольким столбцам

Все предложения, приведенные в табл. 3.1, включены в ANSI-спецификацию языка. В следующих разделах подробно рассказывается об использовании шести основных предложений запроса.

Предложение **select**

Несмотря на то что предложение **select** является первым предложением инструкции **select**, оно является одним из последних предложений, которые вычисляет сервер базы данных. Причина в том, что прежде, чем можно будет определить, что включать в окончательный результирующий набор, нужно знать все возможные столбцы, которые *могут* быть включены в окончательный результирующий набор. Поэтому, чтобы понять назначение предложения **select**, нужно понять назначение предложения **from**. Рассмотрим для начала такой запрос:

```
mysql> SELECT *  
      -> FROM language;  
+-----+-----+-----+  
| language_id | name      | last_update          |  
+-----+-----+-----+  
|          1  | English    | 2006-02-15 05:02:19 |  
|          2  | Italian    | 2006-02-15 05:02:19 |  
|          3  | Japanese   | 2006-02-15 05:02:19 |  
|          4  | Mandarin   | 2006-02-15 05:02:19 |  
|          5  | French     | 2006-02-15 05:02:19 |  
|          6  | German     | 2006-02-15 05:02:19 |  
+-----+-----+-----+  
6 rows in set (0.03 sec)
```

В этом запросе предложение **from** перечисляет одну таблицу (**language**), а предложение **select** указывает, что в результирующий набор должны быть

включены *все* столбцы (что обозначено с помощью *) таблицы. Этот запрос на человеческом языке можно описать следующим образом:

Покажи мне все столбцы и все строки таблицы language.

Помимо указания всех столбцов с помощью символа звездочки (*), можно явно назвать интересующие вас столбцы, например:

```
mysql> SELECT language_id, name, last_update  
-> FROM language;
```

language_id	name	last_update
1	English	2006-02-15 05:02:19
2	Italian	2006-02-15 05:02:19
3	Japanese	2006-02-15 05:02:19
4	Mandarin	2006-02-15 05:02:19
5	French	2006-02-15 05:02:19
6	German	2006-02-15 05:02:19

6 rows in set (0.00 sec)

Результаты идентичны первому запросу, поскольку в предложении `select` указаны *все* столбцы таблицы `language` (`language_id`, `name` и `last_update`). Но можно выбрать только часть столбцов таблицы:

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

name
English
Italian
Japanese
Mandarin
French
German

6 rows in set (0.00 sec)

Таким образом, работа предложения `select` заключается в следующем:

Предложение select определяет, какие из всех возможных столбцов следует включить в результатирующий набор запроса.

Если бы вы ограничивались включением только столбцов из таблицы или таблиц, указанных в предложении `from`, все было бы довольно скучно. Однако вы можете включать в предложение `select` следующее:

- литералы, например числа или строки;
- выражения, такие как `transaction.amount*-1`;

- вызов встроенных функций, например `ROUND(transaction.amount, 2);`
- вызовы пользовательских функций.

В следующем запросе демонстрируется использование столбца таблицы, литерала, выражения и вызова встроенной функции в одном запросе к таблице `employee`:

```
mysql> SELECT language_id,
->   'COMMON' language_usage,
->   language_id * 3.1415927 lang_pi_value,
->   upper(name) language_name
-> FROM language;
+-----+-----+-----+-----+
| language_id | language_usage | lang_pi_value | language_name |
+-----+-----+-----+-----+
|       1 | COMMON      |    3.1415927 | ENGLISH      |
|       2 | COMMON      |     6.2831854 | ITALIAN      |
|       3 | COMMON      |    9.4247781 | JAPANESE     |
|       4 | COMMON      |   12.5663708 | MANDARIN     |
|       5 | COMMON      |   15.7079635 | FRENCH       |
|       6 | COMMON      |   18.8495562 | GERMAN       |
+-----+-----+-----+-----+
6 rows in set (0.04 sec)
```

Мы подробно рассмотрим выражения и встроенные функции позже; я просто хотел дать вам представление о том, что может быть включено в предложение `select`. Если вам нужно только лишь выполнить встроенную функцию или вычислить простое выражение, можете полностью опустить предложение `from`:

```
mysql> SELECT version(),
->   user(),
->   database();
+-----+-----+-----+
| version() | user()        | database()  |
+-----+-----+-----+
| 8.0.15   | root@localhost | sakila     |
+-----+-----+-----+
1 row in set (0.00 sec)
```

Поскольку этот запрос просто вызывает три встроенные функции и не выбирает никакие данные из таблиц, в предложении `from` нет необходимости.

Псевдонимы столбцов

Хотя `mysql` генерирует метки для столбцов, возвращаемых вашими запросами, вы можете назначить им свои названия. Вы можете захотеть назначить

как новую метку столбцу из таблицы (если он плохо или неоднозначно назван), а также, определенно, захотите присвоить собственные метки тем столбцам в результирующем наборе, которые генерируются выражениями или вызовами встроенных функций. Вы можете сделать это, добавив *псевдоимя столбца* после каждого элемента вашего предложения select. Вот как выглядит предыдущий запрос к таблице language, который включает псевдонимы для трех столбцов:

```
mysql> SELECT language_id,
->      'COMMON' language_usage,
->      language_id * 3.1415927 lang_pi_value,
->      upper(name) language_name
->  FROM language;
+-----+-----+-----+-----+
| language_id | language_usage | lang_pi_value | language_name |
+-----+-----+-----+-----+
|       1 | COMMON        |      3.1415927 | ENGLISH      |
|       2 | COMMON        |      6.2831854 | ITALIAN      |
|       3 | COMMON        |      9.4247781 | JAPANESE     |
|       4 | COMMON        |     12.5663708 | MANDARIN    |
|       5 | COMMON        |     15.7079635 | FRENCH      |
|       6 | COMMON        |     18.8495562 | GERMAN      |
+-----+-----+-----+-----+
6 rows in set (0.04 sec)
```

Взглянув на предложение select, вы увидите, как после второго, третьего и четвертого столбцов добавляются псевдонимы столбцов language_usage, lang_pi_value и language_name. Я думаю, вы согласитесь с тем, что при применении псевдонимов столбцов вывод более понятен, и с ними было бы легче работать программно, если бы вы отправляли запрос из Java или Python, а не интерактивно через инструмент mysql. Чтобы подчеркнуть применение псевдонимов столбцов, можно использовать ключевое слово as перед именем псевдонима, например:

```
mysql> SELECT language_id,
->      'COMMON' AS language_usage,
->      language_id * 3.1415927 AS lang_pi_value,
->      upper(name) AS language_name
->  FROM language;
```

Хотя многие считают, что добавление необязательного ключевого слова as улучшает читаемость, я решил не использовать его в примерах в этой книге.

Удаление дубликатов

В некоторых случаях запрос может возвращать повторяющиеся строки данных. Например, если бы вы захотели получить идентификаторы всех актеров, снимавшихся в фильмах, вы бы увидели следующее:

Поскольку некоторые актеры снимались более чем в одном фильме, вы встречаете одни и те же идентификаторы актеров многократно. На самом деле вам, вероятно, нужно множество *различных* актеров, вместо того чтобы видеть их идентификаторы, повторяющиеся в разных фильмах, в которых они снимались. Этого можно добиться, добавив ключевое слово `distinct` непосредственно после ключевого слова `select`:

```
mysql> SELECT DISTINCT actor_id FROM film_actor ORDER BY actor_id;
+-----+
| actor_id |
+-----+
|      1 |
|      2 |
|      3 |
|      4 |
|      5 |
```

```
|   6 |
|   7 |
|   8 |
|   9 |
| 10 |
...
| 192 |
| 193 |
| 194 |
| 195 |
| 196 |
| 197 |
| 198 |
| 199 |
| 200 |
+-----+
200 rows in set (0.01 sec)
```

Результирующий набор теперь содержит 200 строк, по одной для каждого отдельного актера, а не 5462 строки — по одной для каждого появления актера в том или ином фильме.



Если вам нужен просто список всех актеров, можете запросить таблицу `actor` вместо того, чтобы читать все строки таблицы `film_actor` и удалять дубликаты.

Если вы не хотите, чтобы сервер удалял повторяющиеся данные, или вы уверены, что в вашем результирующем наборе дубликатов нет, можете указать ключевое слово `all` вместо `distinct`. Однако ключевое слово `all` используется по умолчанию, так что явно указывать его никогда не нужно. Поэтому большинство программистов не включают `all` в свои запросы.



Помните, что для получения результирующего набора без дубликатов данные требуется отсортировать, что для большого результирующего набора может занять много времени. Не попадайтесь в ловушку `distinct` просто для того, чтобы быть уверенным в отсутствии дубликатов; вместо этого потратьте время на понимание данных, с которыми вы работаете, и разберитесь, возможно ли в вашем конкретном запросе появление дубликатов.

Предложение `from`

До сих пор мы видели запросы, в которых предложения `from` содержат одну таблицу. Несмотря на то, что в большинстве книг по SQL предложение

`from` определяется просто как список из одной или нескольких таблиц, я хотел бы расширить определение следующим образом:

Предложение `from` определяет таблицы, используемые запросом, наряду со средствами связывания таблиц вместе.

Это определение состоит из двух отдельных, но связанных понятий, которые мы исследуем в следующих разделах.

Таблицы

Столкнувшись с термином *таблица*, большинство людей думает о ней как о наборе связанных строк, которые хранятся в базе данных. Хотя такое представление и соответствует одному из типов таблиц, я все же хотел бы использовать это слово в более общем смысле, удалив из понятия любое упоминание о том, как могут храниться данные, и сосредоточиться только на наборе связанных строк. Этому ослабленному определению соответствуют четыре разных типа таблиц.

- Постоянные таблицы (т.е. созданные с помощью инструкции `create table`)
- Производные таблицы (т.е. строки, возвращаемые подзапросом и хранящиеся в памяти)
- Временные таблицы (т.е. изменяемые данные, хранящиеся в памяти)
- Виртуальные таблицы (т.е. созданные с помощью инструкции `create view`).

Каждый из этих типов таблиц может быть включен в предложение `from` запроса. К настоящему времени вы уже должны привыкнуть к включению в предложение `from` постоянных таблиц, поэтому я кратко опишу другие типы таблиц, которые могут использоваться в предложении `from`.

Производные таблицы (генерируемые подзапросами)

Подзапрос — это запрос, содержащийся в другом запросе. Подзапросы окружены круглыми скобками и могут встречаться в различных частях инструкции `select`; однако в предложении `from` подзапрос служит для создания производной таблицы, видимой из всех других предложений запроса и могущей взаимодействовать с другими таблицами, указанными в предложении `from`. Вот простой пример:

```
mysql> SELECT concat(cust.last_name, ' ', cust.first_name) full_name  
-> FROM
```

```

->  (SELECT first_name, last_name, email
->   FROM customer
->   WHERE first_name = 'JESSIE'
-> ) cust;
+-----+
| full_name      |
+-----+
| BANKS, JESSIE |
| MILAM, JESSIE |
+-----+
2 rows in set (0.00 sec)

```

В этом примере подзапрос к таблице клиентов возвращает три столбца, а *содержащий запрос* обращается к двум из этих трех доступных столбцов. Обращение к подзапросу содержащим запросом осуществляется через псевдоним, которым в данном случае является *cust*. Данные таблицы *cust* на время запроса сохраняются в памяти, а затем отбрасываются. Это упрощенный и не особо полезный пример подзапроса в предложении *from*; подробное описание подзапросов представлено в главе 9, “Подзапросы”.

Временные таблицы

Хотя реализации могут быть различными, каждая реляционная база данных позволяет определить изменчивые, или временные, таблицы. Эти таблицы выглядят так же, как и постоянные, но любые данные, добавленные во временную таблицу, в какой-то момент исчезают (обычно в конце транзакции или при закрытии сеанса базы данных). Вот простой пример, как можно временно сохранить актеров, чьи фамилии начинаются с буквы *J*:

```

mysql> CREATE TEMPORARY TABLE actors_j
->   (actor_id smallint(5),
->    first_name varchar(45),
->    last_name varchar(45)
-> );
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO actors_j
->   SELECT actor_id, first_name, last_name
->   FROM actor
->   WHERE last_name LIKE 'J%';
Query OK, 7 rows affected (0.03 sec)
Records: 7 Duplicates: 0 Warnings: 0

```

```

mysql> SELECT * FROM actors_j;
+-----+-----+-----+
| actor_id | first_name | last_name |
+-----+-----+-----+
|     119 | WARREN    | JACKMAN   |

```

```

| 131 | JANE      | JACKMAN   |
| 8  | MATTHEW   | JOHANSSON |
| 64 | RAY       | JOHANSSON |
| 146 | ALBERT    | JOHANSSON |
| 82 | WOODY     | JOLIE     |
| 43 | KIRK      | JOVOVICH  |
+-----+
7 rows in set (0.00 sec)

```

Эти семь строк временно хранятся в памяти и исчезнут после закрытия подключения к базе данных.



Большинство серверов баз данных удаляют временную таблицу, когда заканчивается сеанс работы с базой данных. Исключением является база данных Oracle, в которой определение временной таблицы сохраняется доступным для будущих сеансов.

Представления

Представление — это запрос, который хранится в словаре данных. Он выглядит и действует как таблица, однако данных, связанных с представлением, нет (именно поэтому я называю его *виртуальной* таблицей). При выполнении запроса к представлению этот запрос объединяется с определением представления для создания окончательного запроса, который и будет выполнен.

Чтобы продемонстрировать работу представлений, создадим представление, которое запрашивает таблицу employee и включает четыре из доступных столбцов:

```
mysql> CREATE VIEW cust_vw AS
-> SELECT customer_id, first_name, last_name, active
-> FROM customer;
Query OK, 0 rows affected (0.12 sec)
```

При создании представления никакие дополнительные данные не создаются и не сохраняются: сервер просто откладывает инструкцию select для использования в будущем. Теперь, когда представление существует, вы можете использовать его в запросах, например:

```
mysql> SELECT first_name, last_name
-> FROM cust_vw
-> WHERE active = 0;
+-----+-----+
| first_name | last_name |
+-----+-----+
| SANDRA    | MARTIN   |
| JUDITH    | COX       |
| SHEILA    | WELLS    |
+-----+-----+
```

```

| ERICA      | MATTHEWS   |
| HEIDI      | LARSON     |
| PENNY      | NEAL        |
| KENNETH    | GOODEN     |
| HARRY      | ARCE        |
| NATHAN     | RUNYON     |
| THEODORE   | CULP        |
| MAURICE    | CRAWLEY    |
| BEN         | EASTER      |
| CHRISTIAN  | JUNG        |
| JIMMIE     | EGGLESTON  |
| TERRANCE   | ROUSH       |
+-----+

```

15 rows in set (0.00 sec)

Представления создаются по разным причинам, в том числе для того, чтобы скрыть столбцы от пользователей и упростить сложные конструкции баз данных.

Связи таблиц

Второе отклонение от определения простого предложения `from` заключается в том, что если в предложении `from` указано более одной таблицы, то должны быть включены и условия, используемые для *связи* таблиц. Это не требование сервера MySQL или любой другой базы данных, а одобренный ANSI метод соединения нескольких таблиц, наиболее переносимый между различными серверами баз данных. Мы исследуем соединение нескольких таблиц более подробно в главах 5, “Запросы к нескольким таблицам”, и 10, “Соединения”, а пока простой пример на случай, если мне удалось вас заинтриговать:

```

mysql> SELECT customer.first_name, customer.last_name,
->      time(rental.rental_date) rental_time
->  FROM customer
->    INNER JOIN rental
->      ON customer.customer_id = rental.customer_id
->  WHERE date(rental.rental_date) = '2005-06-14';
+-----+-----+-----+
| first_name | last_name | rental_time |
+-----+-----+-----+
| JEFFERY    | PINSON    | 22:53:33   |
| ELMER      | NOE        | 22:55:13   |
| MINNIE     | ROMERO    | 23:00:34   |
| MIRIAM     | MCKINNEY  | 23:07:08   |
| DANIEL     | CABRAL    | 23:09:38   |
| TERRANCE   | ROUSH     | 23:12:46   |
| JOYCE      | EDWARDS   | 23:16:26   |

```

```

| GWENDOLYN | MAY      | 23:16:27   |
| CATHERINE  | CAMPBELL | 23:17:03   |
| MATTHEW    | MAHAN    | 23:25:58   |
| HERMAN     | DEVORE   | 23:35:09   |
| AMBER      | DIXON    | 23:42:56   |
| TERENCE    | GUNDERSON | 23:47:35   |
| SONIA      | GREGORY  | 23:50:11   |
| CHARLES    | KOWALSKI | 23:54:34   |
| JEANETTE   | GREENE   | 23:54:46   |
+-----+-----+-----+
16 rows in set (0.01 sec)

```

Предыдущий запрос отображает данные как из таблицы `customer` (`first_name`, `last_name`), так и из таблицы `rental` (`rental_date`), поэтому в предложение `from` включены обе таблицы. Механизм связывания двух таблиц (называемый *соединением* (*join*)) — это хранящийся как в таблице `customer`, так и в таблице `rental` идентификатор клиента. Таким образом, серверу базы данных дана инструкция использовать значение столбца `customer_id` в таблице `customer`, чтобы найти все оплаты клиента в таблице `rental`. Условие соединения для двух таблиц находится в подпредложении `on` предложения `from`; в данном случае условие соединения имеет вид `ON customer.customer_id = rental.customer_id`. Предложение `where` не является частью соединения и включено только для того, чтобы набор результатов был небольшим (в таблице `rental` более 16000 строк). Подробно соединения таблиц рассматриваются в главе 5, “Запросы к нескольким таблицам”.

Определение псевдонимов таблиц

Когда несколько таблиц соединяются в один запрос, нужен способ определения, к какой таблице вы обращаетесь, когда указываете столбцы в предложении `select`, `where`, `group by`, `having` и `order by`. Имеется два варианта обращения к таблице за пределами предложения `from`.

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

В предыдущем запросе я решил использовать полное имя таблицы в предложении `select` и `on`. Вот как выглядит тот же запрос с использованием псевдонимов таблиц:

```

SELECT c.first_name, c.last_name,
       time(r.rental_date) rental_time
FROM customer c

```

```
INNER JOIN rental r
ON c.customer_id = r.customer_id
WHERE date(r.rental_date) = '2005-06-14';
```

Если внимательно рассмотреть предложение `from`, то видно, что таблице `customer` назначен псевдоним `c`, а таблице `rental` — псевдоним `r`. Эти псевдонимы затем используются в предложении `on` при определении условия соединения, а также в предложении `select` при указании столбцов для включения в результирующий набор. Надеюсь, вы согласитесь, что использование псевдонимов делает предложения более компактными, не вызывая путаницы (до тех пор, пока выбор псевдонимов является разумным). Кроме того, с псевдонимами таблиц можно использовать ключевое слово `as` аналогично показанному ранее для псевдонимов столбцов:

```
SELECT c.first_name, c.last_name,
       time(r.rental_date) rental_time
  FROM customer AS c
 INNER JOIN rental AS r
   ON c.customer_id = r.customer_id
 WHERE date(r.rental_date) = '2005-06-14';
```

Я обнаружил, что примерно половина разработчиков баз данных, с которыми я работал, используют ключевое слово `as` со своими псевдонимами столбцов и таблиц, а половина обходится без него.

Предложение `where`

В некоторых случаях может потребоваться получить все строки из таблицы, особенно для небольших таблиц, таких как `language`. Однако в большинстве случаев выбирать все строки из таблицы не требуется, а потому необходим способ отфильтровывать строки, которые не представляют интереса. В этом состоит работа предложения `where`.

Предложение `where` — это механизм для фильтрации нежелательных строк из вашего результирующего набора.

Предположим, вы хотите взять фильм напрокат, но вас интересуют только фильмы с рейтингом G, которые можно держать у себя не менее недели. В следующем запросе используется предложение `where` для выборки только тех фильмов, которые соответствуют указанным критериям:

```
mysql> SELECT title
    -> FROM film
    -> WHERE rating = 'G' AND rental_duration >= 7;
```

```
+-----+  
| title |  
+-----+  
| BLANKET BEVERLY |  
| BORROWERS BEDAZZLED |  
| BRIDE INTRIGUE |  
| CATCH AMISTAD |  
| CITIZEN SHREK |  
| COLDBLOODED DARLING |  
| CONTROL ANTHEM |  
| CRUELTY UNFORGIVEN |  
| DARN FORRESTER |  
| DESPERATE TRAINSPOTTING |  
| DIARY PANIC |  
| DRACULA CRYSTAL |  
| EMPIRE MALKOVICH |  
| FIREHOUSE VIETNAM |  
| GILBERT PELICAN |  
| GRADUATE LORD |  
| GREASE YOUTH |  
| GUN BONNIE |  
| HOOK CHARIOTS |  
| MARRIED GO |  
| MENAGERIE RUSHMORE |  
| MUSCLE BRIGHT |  
| OPERATION OPERATION |  
| PRIMARY GLASS |  
| REBEL AIRPORT |  
| SPIKING ELEMENT |  
| TRUMAN CRAZY |  
| WAKE JAWS |  
| WAR NOTTING |  
+-----+  
29 rows in set (0.00 sec)
```

В этом случае предложение `where` отфильтровало и отбросило 971 из 1000 строк в таблице `film`. Это предложение `where` содержит два условия фильтрации, но вы можете включить столько условий, сколько необходимо. Отдельные условия разделяются с помощью таких операторов, как `and`, `or` и `not` (предложение `where` и условия фильтрации рассматриваются в главе 4, “Фильтрация”).

Давайте посмотрим, что произойдет, если изменить оператор, разделяющий два условия, заменив `and` оператором `or`:

```
mysql> SELECT title  
-> FROM film  
-> WHERE rating = 'G' OR rental_duration >= 7;
```

```
+-----+
| title
+-----+
| ACE GOLDFINGER
| ADAPTATION HOLES
| AFFAIR PREJUDICE
| AFRICAN EGG
| ALAMO VIDEOTAPE
| AMISTAD MIDSUMMER
| ANGELS LIFE
| ANNIE IDENTITY
| ...
| WATERSHIP FRONTIER
| WEREWOLF LOLA
| WEST LION
| WESTWARD SEABISCUIT
| WOLVES DESIRE
| WON DARES
| WORKER TARZAN
| YOUNG LANGUAGE
+-----+
```

340 rows in set (0.00 sec)

При разделении условия с использованием оператора `and`, чтобы строка вошла в результирующий набор, *все* условия должны вычисляться как `true`; при использовании `or` для включения строки достаточно, чтобы значение `true` давало только одно из условий. Это объясняет, почему размер результирующего набора увеличился с 29 до 340 строк.

А что делать, если нужно использовать в предложении `where` и оператор `and`, и оператор `or`? Рад, что вы спросили об этом. Вы должны использовать круглые скобки для группировки условий. Следующий запрос указывает, что в результирующий набор будут включены только те фильмы, которые имеют рейтинг `G` и которые можно хранить не менее недели, или те, которые имеют рейтинг `PG-13` и которые можно хранить не более 3 дней:

```
mysql> SELECT title, rating, rental_duration
->   FROM film
-> WHERE (rating = 'G' AND rental_duration >= 7)
->   OR (rating = 'PG-13' AND rental_duration < 4);
+-----+-----+-----+
| title          | rating | rental_duration |
+-----+-----+-----+
| ALABAMA DEVIL | PG-13 |            3 |
| BACKLASH UNDEFEATED | PG-13 |            3 |
| BILKO ANONYMOUS | PG-13 |            3 |
| BLANKET BEVERLY | G     |            7 |
| BORROWERS BEDAZZLED | G     |            7 |
| BRIDE INTRIGUE  | G     |            7 |
```

CASPER DRAGONFLY	PG-13	3
CATCH AMISTAD	G	7
CITIZEN SHREK	G	7
COLDBLOODED DARLING	G	7
...		
TREASURE COMMAND	PG-13	3
TRUMAN CRAZY	G	7
WAIT CIDER	PG-13	3
WAKE JAWS	G	7
WAR NOTTING	G	7
WORLD LEATHERNECKS	PG-13	3

+-----+-----+-----+

68 rows in set (0.00 sec)

Всегда следует использовать круглые скобки для разделения групп условий при смещивании различных операторов, чтобы вы, сервер базы данных и все, кто придет позже и будет изменять ваш код, могли легко его понять.

Предложения group by и having

До сих пор все запросы выполняли выборку данных без какой-либо их обработки. Иногда, однако, требуется увидеть тенденции в данных, для чего сервер базы данных должен их немного подготовить для лучшего восприятия человеком, прежде чем выдать результирующий набор. Один из способов обработки — предложение `group by`, которое используется для группировки данных по значениям столбцов. Предположим, например, что вы захотели найти всех клиентов, которые брали напрокат 40 или более фильмов. Вместо того чтобы просматривать все 16 044 строки в таблице `rental`, вы можете написать запрос, который дает указание серверу сгруппировать все записи проката по клиентам, подсчитать количество для каждого клиента и вернуть только тех клиентов, у которых это количество имеет значение не меньше 40. При использовании предложения `group by` для создания групп строк можно также использовать предложение `having`, которое позволяет фильтровать сгруппированные данные так же, как предложение `where` позволяет фильтровать необработанные данные.

Вот как выглядит такой запрос:

```
mysql> SELECT c.first_name, c.last_name, count(*)
-> FROM customer c
->     INNER JOIN rental r
->     ON c.customer_id = r.customer_id
-> GROUP BY c.first_name, c.last_name
-> HAVING count(*) >= 40;
```

```
+-----+-----+-----+
| first_name | last_name | count(*) |
+-----+-----+-----+
| TAMMY     | SANDERS   |      41 |
| CLARA     | SHAW       |      42 |
| ELEANOR   | HUNT       |      46 |
| SUE        | PETERS    |      40 |
| MARCIA    | DEAN       |      42 |
| WESLEY    | BULL       |      40 |
| KARL      | SEAL       |      45 |
+-----+-----+-----+
7 rows in set (0.03 sec)
```

Я вкратце упомянул эти два предложения, чтобы они не застали вас врасплох далее в этой книге. Они немного сложнее, чем другие четыре предложения `select`, поэтому я прошу вас дождаться главы 8, “Группировка и агрегация”, в которой описано, как и когда использовать предложения `group by` и `having`.

Предложение `order by`

Как правило, строки в результирующем наборе, возвращаемом запросом, не находятся ни в каком конкретном порядке. Если же вы хотите, чтобы ваш результирующий набор был отсортирован, укажите серверу о необходимости сортировать результаты с помощью предложения `order by`:

Предложение `order by` — это механизм для сортировки вашего результирующего набора с использованием любого необработанного столбца данных или выражения на основе данных столбца.

Например, вот еще один вариант предыдущего запроса, который возвращает всех клиентов, бравших фильмы 14 июня 2005 года:

```
mysql> SELECT c.first_name, c.last_name,
->      time(r.rental_date) rental_time
->   FROM customer c
->   INNER JOIN rental r
->     ON c.customer_id = r.customer_id
->   WHERE date(r.rental_date) = '2005-06-14';
+-----+-----+-----+
| first_name | last_name | rental_time |
+-----+-----+-----+
| JEFFERY    | PINSON    | 22:53:33   |
| ELMER      | NOE        | 22:55:13   |
| MINNIE     | ROMERO    | 23:00:34   |
| MIRIAM     | MCKINNEY  | 23:07:08   |
| DANIEL     | CABRAL    | 23:09:38   |
| TERRANCE   | ROUSH     | 23:12:46   |
```

```

| JOYCE      | EDWARDS    | 23:16:26   |
| GWENDOLYN | MAY         | 23:16:27   |
| CATHERINE  | CAMPBELL   | 23:17:03   |
| MATTHEW    | MAHAN       | 23:25:58   |
| HERMAN     | DEVORE      | 23:35:09   |
| AMBER      | DIXON       | 23:42:56   |
| TERRENCE   | GUNDERSON  | 23:47:35   |
| SONIA      | GREGORY    | 23:50:11   |
| CHARLES    | KOWALSKI   | 23:54:34   |
| JEANETTE   | GREENE      | 23:54:46   |
+-----+-----+-----+
16 rows in set (0.01 sec)

```

Для того чтобы результаты находились в алфавитном порядке фамилий клиентов, в предложение `order by` можно добавить столбец `last_name`:

```

mysql> SELECT c.first_name, c.last_name,
->   time(r.rental_date) rental_time
->   FROM customer c
->   INNER JOIN rental r
->   ON c.customer_id = r.customer_id
->   WHERE date(r.rental_date) = '2005-06-14'
->   ORDER BY c.last_name;
+-----+-----+-----+
| first_name | last_name | rental_time |
+-----+-----+-----+

```

```

| DANIEL     | CABRAL     | 23:09:38   |
| CATHERINE  | CAMPBELL   | 23:17:03   |
| HERMAN     | DEVORE     | 23:35:09   |
| AMBER      | DIXON      | 23:42:56   |
| JOYCE      | EDWARDS    | 23:16:26   |
| JEANETTE   | GREENE     | 23:54:46   |
| SONIA      | GREGORY   | 23:50:11   |
| TERRENCE   | GUNDERSON | 23:47:35   |
| CHARLES    | KOWALSKI  | 23:54:34   |
| MATTHEW    | MAHAN      | 23:25:58   |
| GWENDOLYN | MAY        | 23:16:27   |
| MIRIAM     | MCKINNEY   | 23:07:08   |
| ELMER      | NOE        | 22:55:13   |
| JEFFERY    | PINSON     | 22:53:33   |
| MINNIE     | ROMERO    | 23:00:34   |
| TERRANCE   | ROUSH      | 23:12:46   |
+-----+-----+-----+

```

16 rows in set (0.01 sec)

Хотя в данном примере этого и нет, в больших списках клиентов часто содержатся несколько однофамильцев, поэтому вы можете расширить критерии сортировки, указав в ней и имя человека.

Для этого надо просто добавить столбец `first_name` после столбца `last_name` в предложении `order by`:

```

mysql> SELECT c.first_name, c.last_name,
->     time(r.rental_date) rental_time
->   FROM customer c
->   INNER JOIN rental r
->     ON c.customer_id = r.customer_id
->   WHERE date(r.rental_date) = '2005-06-14'
->   ORDER BY c.last_name, c.first_name;
+-----+-----+-----+
| first_name | last_name | rental_time |
+-----+-----+-----+
| DANIEL      | CABRAL    | 23:09:38   |
| CATHERINE   | CAMPBELL  | 23:17:03   |
| HERMAN      | DEVORE    | 23:35:09   |
| AMBER        | DIXON     | 23:42:56   |
| JOYCE        | EDWARDS   | 23:16:26   |
| JEANETTE    | GREENE    | 23:54:46   |
| SONIA        | GREGORY   | 23:50:11   |
| TERENCE     | GUNDERSON | 23:47:35   |
| CHARLES     | KOWALSKI  | 23:54:34   |
| MATTHEW     | MAHAN     | 23:25:58   |
| GWENDOLYN   | MAY       | 23:16:27   |
| MIRIAM      | MCKINNEY  | 23:07:08   |
| ELMER        | NOE       | 22:55:13   |
| JEFFERY     | PINSON    | 22:53:33   |
| MINNIE      | ROMERO   | 23:00:34   |
| TERRANCE    | ROUSH     | 23:12:46   |
+-----+-----+-----+
16 rows in set (0.01 sec)

```

Порядок, в котором столбцы появляются в вашем предложении `order by`, имеет значение, когда вы включаете более одного столбца. Если бы вы поменяли порядок двух столбцов в предложении `order by`, первой в результирующем наборе была бы указана Amber Dixon.

Сортировка по возрастанию и убыванию

При сортировке у вас есть возможность указать порядок сортировки — по возрастанию или по убыванию — с помощью ключевых слов `asc` и `desc`. По умолчанию используется сортировка по возрастанию, поэтому, если вы хотите использовать сортировку по убыванию, добавьте ключевое слово `desc`. Например, в следующем запросе показаны все клиенты, взявшие фильмы на прокат 14 июня 2005 года, в порядке убывания времени проката:

```

mysql> SELECT c.first_name, c.last_name,
->     time(r.rental_date) rental_time
->   FROM customer c
->   INNER JOIN rental r
->     ON c.customer_id = r.customer_id

```

```

-> WHERE date(r.rental_date) = '2005-06-14'
-> ORDER BY time(r.rental_date) desc;
+-----+-----+-----+
| first_name | last_name | rental_time |
+-----+-----+-----+
| JEANETTE   | GREENE    | 23:54:46   |
| CHARLES    | KOWALSKI  | 23:54:34   |
| SONIA      | GREGORY   | 23:50:11   |
| TERENCE    | GUNDERSON| 23:47:35   |
| AMBER      | DIXON     | 23:42:56   |
| HERMAN     | DEVORE    | 23:35:09   |
| MATTHEW    | MAHAN     | 23:25:58   |
| CATHERINE  | CAMPBELL  | 23:17:03   |
| GWENDOLYN  | MAY       | 23:16:27   |
| JOYCE      | EDWARDS   | 23:16:26   |
| TERRANCE   | ROUSH     | 23:12:46   |
| DANIEL     | CABRAL    | 23:09:38   |
| MIRIAM     | MCKINNEY  | 23:07:08   |
| MINNIE     | ROMERO    | 23:00:34   |
| ELMER      | NOE        | 22:55:13   |
| JEFFERY    | PINSON    | 22:53:33   |
+-----+-----+-----+

```

16 rows in set (0.01 sec)

Сортировка по убыванию обычно используется для ранжирования запросов наподобие “покажи мне пять наибольших остатков на счетах”. MySQL включает предложение `limit`, которое позволяет отсортировать данные, а затем отбросить все, кроме первых X строк.

Сортировка с помощью номера столбца

Если выполняется сортировка по столбцам в предложении `select`, можно указывать столбцы для сортировки по их *позиции* в предложении `select`, а не по имени. Это может быть особенно полезно, если выполняется сортировка по выражению, как в предыдущем примере. Вот как выглядит предыдущий пример, если в предложении `order by` указывается сортировка по убыванию с использованием третьего элемента в предложении `select`:

```

mysql> SELECT c.first_name, c.last_name,
->      time(r.rental_date) rental_time
->   FROM customer c
-> INNER JOIN rental r
->   ON c.customer_id = r.customer_id
-> WHERE date(r.rental_date) = '2005-06-14'
-> ORDER BY 3 desc;
+-----+-----+-----+
| first_name | last_name | rental_time |
+-----+-----+-----+

```

JEANETTE	GREENE	23:54:46	
CHARLES	KOWALSKI	23:54:34	
SONIA	GREGORY	23:50:11	
TERENCE	GUNDERSON	23:47:35	
AMBER	DIXON	23:42:56	
HERMAN	DEVORE	23:35:09	
MATTHEW	MAHAN	23:25:58	
CATHERINE	CAMPBELL	23:17:03	
GWENDOLYN	MAY	23:16:27	
JOYCE	EDWARDS	23:16:26	
TERRANCE	ROUSH	23:12:46	
DANIEL	CABRAL	23:09:38	
MIRIAM	MCKINNEY	23:07:08	
MINNIE	ROMERO	23:00:34	
ELMER	NOE	22:55:13	
JEFFERY	PINSON	22:53:33	

16 rows in set (0.01 sec)

Возможно, вы не захотите широко применять эту возможность, так как добавление столбца к предложению `select` без соответствующего изменения номера в предложении `order by` может привести к неожиданным результатам. Лично я ссылаюсь на позиции столбцов только при написании специализированных разовых запросов, но при разработке серьезного кода я всегда указываю столбцы по имени.

Проверьте свои знания

Предлагаемые здесь упражнения призваны закрепить понимание вами инструкции `select` и различных ее предложений. Ответы к упражнениям представлены в приложении Б.

УПРАЖНЕНИЕ 3.1

Получите идентификатор актера, а также имя и фамилию для всех актеров. Отсортируйте вывод сначала по фамилии, а затем — по имени.

УПРАЖНЕНИЕ 3.2

Получите идентификатор, имя и фамилию актера для всех актеров, чьи фамилии — 'WILLIAMS' или 'DAVIS'.

УПРАЖНЕНИЕ 3.3

Напишите запрос к таблице `rental`, который возвращает идентификаторы клиентов, бравших фильмы напрокат 5 июля 2005 года (используйте столбец `rental.rental_date`; можете также использовать функцию `date()`, чтобы

игнорировать компонент времени). Выведите по одной строке для каждого уникального идентификатора клиента.

УПРАЖНЕНИЕ 3.4

Заполните пропущенные места (обозначенные как <#>) в следующем многострочном запросе, чтобы получить показанные результаты:

```
mysql> SELECT c.email, r.return_date
-> FROM customer c
-> INNER JOIN rental <1>
-> ON c.customer_id = <2>
-> WHERE date(r.rental_date) = '2005-06-14'
-> ORDER BY <3> <4>;
+-----+-----+
| email | return_date |
+-----+-----+
| DANIEL.CABRAL@sakilacustomer.org | 2005-06-23 22:00:38 |
| TERRANCE.ROUSH@sakilacustomer.org | 2005-06-23 21:53:46 |
| MIRIAM.MCKINNEY@sakilacustomer.org | 2005-06-21 17:12:08 |
| GWENDOLYN.MAY@sakilacustomer.org | 2005-06-20 02:40:27 |
| JEANETTE.GREENE@sakilacustomer.org | 2005-06-19 23:26:46 |
| HERMAN.DEVORE@sakilacustomer.org | 2005-06-19 03:20:09 |
| JEFFERY.PINSON@sakilacustomer.org | 2005-06-18 21:37:33 |
| MATTHEW.MAHAN@sakilacustomer.org | 2005-06-18 05:18:58 |
| MINNIE.ROMERO@sakilacustomer.org | 2005-06-18 01:58:34 |
| SONIA.GREGORY@sakilacustomer.org | 2005-06-17 21:44:11 |
| TERRENCE.GUNDERSON@sakilacustomer.org | 2005-06-17 05:28:35 |
| ELMER.NOE@sakilacustomer.org | 2005-06-17 02:11:13 |
| JOYCE.EDWARDS@sakilacustomer.org | 2005-06-16 21:00:26 |
| AMBER.DIXON@sakilacustomer.org | 2005-06-16 04:02:56 |
| CHARLES.KOWALSKI@sakilacustomer.org | 2005-06-16 02:26:34 |
| CATHERINE.CAMPBELL@sakilacustomer.org | 2005-06-15 20:43:03 |
+-----+
16 rows in set (0.03 sec)
```

Фильтрация

Иногда необходимо работать с каждой строкой таблицы. Вот примеры таких ситуаций:

- удаление всех данных из таблицы, используемой при подготовке заполнения данными нового хранилища;
- изменение всех строк в таблице после добавления нового столбца;
- получение всех строк из таблицы очереди сообщений.

В таких случаях в инструкциях SQL предложение `where` не обязательно, поскольку исключать из рассмотрения какие-либо строки не требуется. Однако в большинстве случаев желательно сузить фокус до некоторого подмножества строк таблицы. Поэтому все инструкции данных SQL (кроме инструкции `insert`) включают необязательное предложение `where`, содержащее одно или несколько *условий фильтрации*, используемых для ограничения количества строк, на которые действует инструкция SQL. Кроме того, инструкция `select` включает предложение `having`, в которое могут быть включены условия фильтрации, относящиеся к сгруппированным данным. В данной главе рассматриваются различные типы условий фильтрации, которые можно использовать в предложениях `where` инструкций `select`, `update` и `delete`. Использование условий фильтрации в предложении `having` инструкции `select` будет рассмотрено в главе 8, “Группировка и агрегация”.

Вычисление условий

Предложение `where` может содержать одно или несколько *условий*, разделенных операторами `and` и `or`. Если несколько условий разделяются только оператором `and`, то все эти условия должны быть истинными, чтобы строка была включена в результирующий набор. Рассмотрим следующее предложение `where`:

```
WHERE first_name = 'STEVEN' AND create_date > '2006-01-01'
```

С учетом этих двух условий в результирующий набор будут включены только строки, в которых имя — 'STEVEN', а `create_date` — после 1 января 2006 года. В этом примере используются только два условия, но независимо от того, сколько условий содержится в вашем предложении `where`, если они разделены оператором `and`, то, чтобы строка была включена в результирующий набор, *все* они должны иметь значение `true`.

Однако если все условия в предложении `where` разделены оператором `or`, то, чтобы строка была включена в результирующий набор, достаточно, чтобы только *одно* из условий было истинным. Рассмотрим следующие два условия:

```
WHERE first_name = 'STEVEN' OR create_date > '2006-01-01'
```

Строка может попасть в результирующий набор разными способами:

- если имя — 'STEVEN', а дата создания — после 1 января 2006 года;
- если имя — 'STEVEN', а дата создания — 1 января 2006 года или ранее;
- если имя — любое, но не 'STEVEN', а дата создания — после 1 января 2006 года.

В табл. 4.1 показаны возможные результаты предложения `where`, содержащего два условия, которые разделены оператором `or`.

Таблица 4.1. Вычисление двух условий с оператором or

Промежуточный результат	Окончательный результат
WHERE true OR true	true
WHERE true OR false	true
WHERE false OR true	true
WHERE false OR false	false

Использование скобок

Если ваше предложение `where` включает три или более условий, в которых используются операторы `and` и `or`, следует использовать круглые скобки, чтобы прояснить свое намерение как для сервера базы данных, так и для всех, кто читает ваш код. Вот предложение `where`, расширяющее предыдущий пример, добавляя в результирующий набор все записи с именем 'STEVEN' или фамилией 'YOUNG' и с датой создания после 1 января 2006 года:

```
WHERE (first_name = 'STEVEN' OR last_name = 'YOUNG')  
      AND create_date > '2006-01-01'
```

Теперь для строки, чтобы попасть в окончательный результатирующий набор, имеется уже три условия; либо первое, либо второе условие (или они оба) должно быть истинным; третье условие также должно быть истинным. В табл. 4.2 показаны возможные результаты вычисления этого предложения `where`.

Таблица 4.2. Вычисление трех условий с операторами `and` и `or`

Промежуточный результат	Окончательный результат
<code>WHERE (true OR true) AND true</code>	<code>true</code>
<code>WHERE (true OR false) AND true</code>	<code>true</code>
<code>WHERE (false OR true) AND true</code>	<code>true</code>
<code>WHERE (false OR false) AND true</code>	<code>false</code>
<code>WHERE (true OR true) AND false</code>	<code>false</code>
<code>WHERE (true OR false) AND false</code>	<code>false</code>
<code>WHERE (false OR true) AND false</code>	<code>false</code>
<code>WHERE (false OR false) AND false</code>	<code>false</code>

Как видите, чем больше условий в предложении `where`, тем больше комбинаций приходится вычислять серверу. В данном случае только три из восьми комбинаций дают значение окончательного результата, равное `true`.

Использование оператора `not`

Надеюсь, предыдущий пример с тремя условиями вам понятен. Рассмотрим теперь его модификацию:

```
WHERE NOT (first_name = 'STEVEN' OR last_name = 'YOUNG')  
AND create_date > '2006-01-01'
```

Вы заметили, чем отличается этот запрос от предыдущего примера? Я добавил оператор `not` перед первым набором условий. Теперь вместо того, чтобы искать людей с именем '`STEVEN`' или фамилией '`YOUNG`', запись для которых была создана после 1 января 2006 года, я получаю строки, в которых имя — не '`STEVEN`' и фамилия — не '`YOUNG`' и запись для которых была создана после 1 января 2006 года. В табл. 4.3 показаны возможные результаты для этого примера.

Таблица 4.3. Вычисление трех условий с операторами `and` и `or`

Промежуточный результат	Окончательный результат
<code>WHERE NOT (true OR true) AND true</code>	<code>false</code>
<code>WHERE NOT (true OR false) AND true</code>	<code>false</code>

Промежуточный результат	Окончательный результат
WHERE NOT (false OR true) AND true	false
WHERE NOT (false OR false) AND true	true
WHERE NOT (true OR true) AND false	false
WHERE NOT (true OR false) AND false	false
WHERE NOT (false OR true) AND false	false
WHERE NOT (false OR false) AND false	false

Хотя сервер базы данных легко справляется с такими выражениями, человеку, как правило, сложно понять предложение `where`, которое включает оператор `not`, поэтому вы будете сталкиваться с ним не очень часто. В данном случае можно переписать предложение `where` так, чтобы избежать использования оператора `not`:

```
WHERE first_name <> 'STEVEN' AND last_name <> 'YOUNG'  
AND create_date > '2006-01-01'
```

Хотя я уверен, что у сервера нет никаких предпочтений, вы, вероятно, сочтете эту запись более простой версией предложения `where`.

Построение условия

Теперь, когда вы увидели, как сервер вычисляет несколько условий, давайте сделаем шаг назад и посмотрим, что может собой представлять одно условие. Условие состоит из одного или нескольких *выражений* в сочетании с одним или несколькими *операторами*. Выражение может быть одним из следующего.

- Число
- Столбец таблицы или представления
- Строковый литерал, такой как 'Maple Street'
- Встроенная функция, такая как `concat('Learning', ' ', 'SQL')`
- Подзапрос
- Список выражений, такой как ('Boston', 'New York', 'Chicago')

Операторы, используемые в условиях, включают:

- операторы сравнения, такие как `=, !=, <, >, <>`, `like`, `in` и `between`;
- арифметические операторы, такие как `+, -, *, /`.

В следующем разделе показано, как можно комбинировать эти выражения и операторы для создания условий различных типов.

Типы условий

Есть много разных способов отфильтровать нежелательные данные. Вы можете искать конкретные значения, наборы значений или диапазоны значений для включения или исключения, можете использовать различные методы поиска по шаблону для поиска частичных совпадений при работе со строковыми данными. В следующих четырех подразделах подробно рассматривается каждый из этих типов условий.

Условия равенства

Наибольший процент условий фильтрации, которые вы будете писать или встречать у других, — это условие вида '*столбец=выражение*', как, например,

```
title = 'RIVER OUTLAW'  
fed_id = '111-11-1111'  
amount = 375.25  
film_id = (SELECT film_id FROM film WHERE title = 'RIVER OUTLAW')
```

Такие условия называются *условиями равенства*, потому что они приравнивают одно выражение к другому. В первых трех примерах столбец приравнивается к литералу (две строки и число), а в четвертом — к значению, которое возвращает подзапрос. В следующем запросе используются два условия равенства: одно — в предложении *on* (условие соединения), а второе — в предложении *where* (условие фильтрации):

```
mysql> SELECT c.email  
    ->   FROM customer c  
    ->     INNER JOIN rental r  
    ->       ON c.customer_id = r.customer_id  
    ->     WHERE date(r.rental_date) = '2005-06-14';  
+-----+  
| email           |  
+-----+  
| CATHERINE.CAMPBELL@sakilacustomer.org |  
| JOYCE.EDWARDS@sakilacustomer.org      |  
| AMBER.DIXON@sakilacustomer.org        |  
| JEANETTE.GREENE@sakilacustomer.org    |  
| MINNIE.ROMERO@sakilacustomer.org      |  
| GWENDOLYN.MAY@sakilacustomer.org      |  
| SONIA.GREGORY@sakilacustomer.org      |  
| MIRIAM.MCKINNEY@sakilacustomer.org    |  
| CHARLES.KOWALSKI@sakilacustomer.org   |
```

```
| DANIEL.CABRAL@sakilacustomer.org      |
| MATTHEW.MAHAN@sakilacustomer.org      |
| JEFFERY.PINSON@sakilacustomer.org      |
| HERMAN.DEVORE@sakilacustomer.org      |
| ELMER.NOE@sakilacustomer.org          |
| TERRANCE.ROUSH@sakilacustomer.org     |
| TERRENCE.GUNDERSON@sakilacustomer.org |
+-----+
16 rows in set (0.03 sec)
```

Этот запрос показывает адреса электронной почты каждого клиента, взявшего фильм напрокат 14 июня 2005 года.

Условия неравенства

Другой довольно распространенный тип условий — *условия неравенства*; он проверяет, что два выражения *не* равны. Вот как выглядит предыдущий запрос с условием фильтрации в предложении *where*, измененным на условие неравенства:

```
mysql> SELECT c.email
    -> FROM customer c
    -> INNER JOIN rental r
    -> ON c.customer_id = r.customer_id
    -> WHERE date(r.rental_date) <> '2005-06-14';
+-----+
| email           |
+-----+
| MARY.SMITH@sakilacustomer.org   |
| ...               |
| AUSTIN.CINTRON@sakilacustomer.org |
+-----+
16028 rows in set (0.03 sec)
```

Этот запрос возвращает все адреса электронной почты для клиентов, взявшим фильм напрокат в любую дату, кроме 14 июня 2005 года. При построении условий неравенства вы можете использовать оператор `!=` или `<>`.

Модификация данных с использованием условий равенства

Условия равенства/неравенства обычно используются при изменении данных. Например, предположим, что компания по прокату фильмов придерживается политики удаления старых строк аккаунта один раз в год. Ваша задача — удалить те строки из таблицы `rental`, в которых дата аренды соответствует 2004 году. Вот один из способов решить эту проблему:

```
DELETE FROM rental  
WHERE year(rental_date) = 2004;
```

Эта инструкция включает единственное условие равенства. А вот пример, в котором используются два условия неравенства для удаления любых строк, срок аренды которых не соответствует ни 2005, ни 2006 году:

```
DELETE FROM rental  
WHERE year(rental_date) <> 2005 AND year(rental_date) <> 2006;
```



При создании примеров инструкций `delete` и `update` я стараюсь писать каждый оператор таким образом, чтобы ни одна строка не изменялась. Поэтому, когда вы выполняете эти инструкции, ваши данные остаются без изменений, и вывод инструкций `select` всегда соответствует тому, что показано в книге.

Поскольку сеансы MySQL по умолчанию находятся в режиме автоматической фиксации (см. главу 12, “Транзакции”), вы не смогли бы откатить (отменить) какие-либо изменения, внесенные в данные примера, если бы одна из моих инструкций изменила данные. Вы, конечно, можете делать все что хотите, с примером базы данных, включая удаление данных и повторный запуск сценариев для заполнения таблиц, но я стараюсь оставлять их нетронутыми.

Условия диапазона

Наряду с проверкой того, что выражение равно (или не равно) другому выражению, вы можете создавать условия, которые проверяют, попадает ли выражение в определенный диапазон. Этот тип условия часто встречается при работе с числовыми или временными данными. Рассмотрим следующий запрос:

```
mysql> SELECT customer_id, rental_date
-> FROM rental
-> WHERE rental_date < '2005-05-25';
+-----+
| customer_id | rental_date           |
+-----+
|      130    | 2005-05-24 22:53:30 |
|      459    | 2005-05-24 22:54:33 |
|      408    | 2005-05-24 23:03:39 |
|      333    | 2005-05-24 23:04:41 |
|      222    | 2005-05-24 23:05:21 |
|      549    | 2005-05-24 23:08:07 |
|      269    | 2005-05-24 23:11:53 |
|      239    | 2005-05-24 23:31:46 |
+-----+
8 rows in set (0.00 sec)
```

Этот запрос находит все фильмы, взятые напрокат до 25 мая 2005 года. Указав верхний предел для даты аренды, можно указать более узкий диапазон:

```
mysql> SELECT customer_id, rental_date
-> FROM rental
-> WHERE rental_date <= '2005-06-16'
-> AND rental_date >= '2005-06-14';
+-----+
| customer_id | rental_date           |
+-----+
|      416    | 2005-06-14 22:53:33 |
|      516    | 2005-06-14 22:55:13 |
|      239    | 2005-06-14 23:00:34 |
|      285    | 2005-06-14 23:07:08 |
|      310    | 2005-06-14 23:09:38 |
|      592    | 2005-06-14 23:12:46 |
| ...       | ...
|      148    | 2005-06-15 23:20:26 |
|      237    | 2005-06-15 23:36:37 |
|      155    | 2005-06-15 23:55:27 |
|      341    | 2005-06-15 23:57:20 |
|      149    | 2005-06-15 23:58:53 |
+-----+
364 rows in set (0.00 sec)
```

Эта версия запроса извлекает все фильмы, взятые напрокат 14 или 15 июня 2005 года.

Оператор between

Если у вас есть как верхний, так и нижний пределы вашего диапазона, вы можете использовать единственное условие, которое использует оператор `between`, а не два отдельных условия:

```

mysql> SELECT customer_id, rental_date
-> FROM rental
-> WHERE rental_date BETWEEN '2005-06-14' AND '2005-06-16';
+-----+-----+
| customer_id | rental_date      |
+-----+-----+
|     416 | 2005-06-14 22:53:33 |
|     516 | 2005-06-14 22:55:13 |
|    239 | 2005-06-14 23:00:34 |
|    285 | 2005-06-14 23:07:08 |
|    310 | 2005-06-14 23:09:38 |
|    592 | 2005-06-14 23:12:46 |
| ...   |
|    148 | 2005-06-15 23:20:26 |
|   237 | 2005-06-15 23:36:37 |
|   155 | 2005-06-15 23:55:27 |
|   341 | 2005-06-15 23:57:20 |
|   149 | 2005-06-15 23:58:53 |
+-----+
364 rows in set (0.00 sec)

```

При использовании оператора `between` следует помнить о нескольких вещах. Всегда необходимо указывать сначала нижнюю границу диапазона (после `between`) и верхнюю границу (после `and`). Вот что произойдет, если вы ошибке укажете сначала верхнюю границу:

```

mysql> SELECT customer_id, rental_date
-> FROM rental
-> WHERE rental_date BETWEEN '2005-06-16' AND '2005-06-14';
Empty set (0.00 sec)

```

Как видите, никакие данные не возвращаются. Дело в том, что сервер, по сути, генерирует из вашего единственного условия два условия с использованием операторов `<=` и `>=`, как показано ниже:

```

SELECT customer_id, rental_date
-> FROM rental
-> WHERE rental_date >= '2005-06-16'
->   AND rental_date <= '2005-06-14'
Empty set (0.00 sec)

```

Поскольку невозможно иметь дату, одновременно большую 16 июня 2005 года и меньшую 14 июня 2005 года, запрос возвращает пустой результатирующий набор. Это подводит меня ко второй ловушке: при использовании `between` следует помнить, что ваши верхний и нижний границы указаны **включительно**, а это означает, что указанные вами значения входят в пределы диапазона. В показанном случае я хочу вернуть все фильмы, взятые напрокат 14 или 15 июня, поэтому я указываю 2005-06-14 в качестве нижней границы

диапазона и 2005-06-16 — в качестве верхней. Поскольку я не указываю компонент времени, а время по умолчанию — полночь, мой диапазон длится от 2005-06-14 00:00:00 до 2005-06-16 00:00:00 и включает все фильмы, взятые напрокат 14 или 15 июня.

Наряду с датами можно также создавать условия, указывающие диапазоны чисел. Числовые диапазоны просты в понимании. Взгляните на следующий запрос:

```
mysql> SELECT customer_id, payment_date, amount
-> FROM payment
-> WHERE amount BETWEEN 10.0 AND 11.99;
+-----+-----+
| customer_id | payment_date           | amount |
+-----+-----+
|      2 | 2005-07-30 13:47:43 | 10.99 |
|      3 | 2005-07-27 20:23:12 | 10.99 |
|     12 | 2005-08-01 06:50:26 | 10.99 |
|     13 | 2005-07-29 22:37:41 | 11.99 |
|    21 | 2005-06-21 01:04:35 | 10.99 |
|    29 | 2005-07-09 21:55:19 | 10.99 |
| ...   |
|  571 | 2005-06-20 08:15:27 | 10.99 |
|  572 | 2005-06-17 04:05:12 | 10.99 |
|  573 | 2005-07-31 12:14:19 | 10.99 |
|  591 | 2005-07-07 20:45:51 | 11.99 |
|  592 | 2005-07-06 22:58:31 | 11.99 |
|  595 | 2005-07-31 11:51:46 | 10.99 |
+-----+-----+
114 rows in set (0.01 sec)
```

Возвращаются все платежи от 10 до 11,99 долл. Не забудьте убедиться, что сначала указана меньшая сумма.

Строковые диапазоны

Легко понять диапазоны дат и чисел, но вы можете также создавать условия для поиска в диапазоне строк, хотя визуализировать их немного сложнее. Скажем, вы ищете клиентов, фамилии которых попадают в некоторый диапазон. Вот запрос, который возвращает клиентов, фамилии которых находятся между FA и FR:

```
mysql> SELECT last_name, first_name
-> FROM customer
-> WHERE last_name BETWEEN 'FA' AND 'FR';
+-----+-----+
| last_name | first_name |
+-----+-----+
| FARNSWORTH | JOHN       |
```

```

| FENNELL      | ALEXANDER   |
| FERGUSON     | BERTHA      |
| FERNANDEZ    | MELINDA     |
| FIELDS       | VICKI       |
| FISHER        | CINDY       |
| FLEMING      | MYRTLE     |
| FLETCHER     | MAE         |
| FLORES       | JULIA       |
| FORD          | CRYSTAL     |
| FORMAN        | MICHEAL     |
| FORSYTHE     | ENRIQUE     |
| FORTIER      | RAUL        |
| FORTNER      | HOWARD      |
| FOSTER        | PHYLLIS     |
| FOUST         | JACK        |
| FOWLER        | JO          |
| FOX           | HOLLY       |
+-----+

```

18 rows in set (0.00 sec)

Несмотря на то что есть пять клиентов, чьи фамилии начинаются с FR, они не включены в результаты, поскольку имя наподобие FRANKLIN находится за пределами диапазона. Однако мы можем выбрать четырех из этих пяти клиентов, расширив диапазон вправо до FRB:

```

mysql> SELECT last_name, first_name
    -> FROM customer
    -> WHERE last_name BETWEEN 'FA' AND 'FRB';
+-----+-----+
| last_name | first_name |
+-----+-----+
| FARNSWORTH | JOHN      |
| FENNELL    | ALEXANDER |
| FERGUSON    | BERTHA    |
| FERNANDEZ   | MELINDA   |
| FIELDS      | VICKI     |
| FISHER      | CINDY     |
| FLEMING     | MYRTLE    |
| FLETCHER    | MAE       |
| FLORES      | JULIA     |
| FORD         | CRYSTAL   |
| FORMAN       | MICHEAL   |
| FORSYTHE    | ENRIQUE   |
| FORTIER     | RAUL      |
| FORTNER     | HOWARD    |
| FOSTER       | PHYLLIS   |
| FOUST        | JACK      |
| FOWLER       | JO        |
| FOX          | HOLLY     |
| FRALEY       | JUAN      |
+-----+

```

```
| FRANCISCO | JOEL      |
| FRANKLIN   | BETH      |
| FRAZIER    | GLENDA   |
+-----+
22 rows in set (0.00 sec)
```

Чтобы работать с диапазонами строк, необходимо знать порядок символов в вашем наборе символов (порядок, в котором сортируются символы в наборе символов, называется *сравнением* (collation)).

Условия членства

В некоторых случаях вы не будете ограничивать выражение одним значением или диапазоном значений, а будете использовать некоторый конечный набор значений. Например, вы можете захотеть найти все фильмы с рейтингом G или PG:

```
mysql> SELECT title, rating
    -> FROM film
    -> WHERE rating = 'G' OR rating = 'PG';
+-----+-----+
| title          | rating |
+-----+-----+
| ACADEMY DINOSAUR | PG     |
| ACE GOLDFINGER  | G      |
| AFFAIR PREJUDICE | G      |
| AFRICAN EGG     | G      |
| AGENT TRUMAN    | PG     |
| ALAMO VIDEOTAPE | G      |
| ALASKA PHANTOM  | PG     |
| ALI FOREVER     | PG     |
| AMADEUS HOLY    | PG     |
| ...
| WEDDING APOLLO  | PG     |
| WEREWOLF LOLA   | G      |
| WEST LION        | G      |
| WIZARD COLDBLOODED | PG     |
| WON DARES       | PG     |
| WONDERLAND CHRISTMAS | PG     |
| WORDS HUNTER    | PG     |
| WORST BANGER    | PG     |
| YOUNG LANGUAGE   | G      |
+-----+-----+
372 rows in set (0.00 sec)
```

Хотя создать это предложение `where` (два условия, объединенных оператором `or`) было не слишком утомительно, представьте, что набор выражений содержит 10 или 20 членов. Для таких ситуаций можно использовать оператор `in`:

```
SELECT title, rating
FROM film
WHERE rating IN ('G', 'PG');
```

С помощью оператора `in` вы можете написать одно условие независимо от того, сколько выражений входит в набор.

Использование подзапросов

Помимо написания собственного набора выражений, таких как `('G', 'PG')`, вы можете создавать наборы “на лету”, используя подзапросы. Например, если можно считать, что любой фильм, название которого включает строку `'PET'`, будет безопасным для семейного просмотра, то можно выполнить подзапрос к таблице фильмов, чтобы получить все рейтинги, связанные с этими фильмами, а уже затем получить все фильмы, имеющие любой из этих рейтингов:

```
mysql> SELECT title, rating
-> FROM film
-> WHERE rating IN (SELECT rating FROM film WHERE title LIKE '%PET%');
+-----+-----+
| title          | rating |
+-----+-----+
| ACADEMY DINOSAUR | PG    |
| ACE GOLDFINGER   | G     |
| AFFAIR PREJUDICE | G     |
| AFRICAN EGG      | G     |
| AGENT TRUMAN     | PG    |
| ALAMO VIDEOTAPE  | G     |
| ALASKA PHANTOM    | PG    |
| ALI EVERLASTING    | PG    |
| AMADEUS HOLY      | PG    |
| ...               |       |
| WEDDING APOLLO    | PG    |
| WEREWOLF LOLA     | G     |
| WEST LION          | G     |
| WIZARD COLDBLOODED | PG    |
| WON DARES          | PG    |
| WONDERLAND CHRISTMAS | PG    |
| WORDS HUNTER        | PG    |
| WORST BANGER        | PG    |
| YOUNG LANGUAGE      | G     |
+-----+-----+
372 rows in set (0.00 sec)
```

Подзапрос возвращает набор `'G'` и `'PG'`, а основной запрос проверяет, находится ли значение столбца `rating` в наборе, возвращаемом подзапросом.

Использование `not in`

Иногда вам нужны данные, для которых конкретное выражение имеется в наборе выражений, но иногда нужны данные, для которых конкретное выражение в наборе выражений отсутствует. В таких случаях можно использовать оператор `not in`:

```
SELECT title, rating
FROM film
WHERE rating NOT IN ('PG-13', 'R', 'NC-17');
```

Этот запрос находит все записи, *не* помеченные рейтингом 'PG-13', 'R' или 'NC-17', и возвращает тот же набор из 372 строк, что и предыдущие запросы.

Условия соответствия

До сих пор мы встречались с условиями, которые определяют точную строку, диапазон строк или набор строк; последний из перечисленных типов условий имеет дело с частичными совпадениями строк, например так можно найти всех клиентов, чьи фамилии начинаются с Q. Можно также использовать встроенную функцию для выделения первой буквы столбца `last_name`:

```
mysql> SELECT last_name, first_name
    -> FROM customer
    -> WHERE left(last_name, 1) = 'Q';
+-----+-----+
| last_name | first_name |
+-----+-----+
| QUALLS   | STEPHEN    |
| QUINTANILLA | ROGER      |
| QUIGLEY   | TROY        |
+-----+-----+
3 rows in set (0.00 sec)
```

Хотя встроенная функция `left()` выполняет свою работу, она не обеспечивает большой гибкости. Но вы можете использовать подстановочные знаки для построения поисковых выражений, как показано в следующем подразделе.

Использование подстановочных знаков

При поиске частичных совпадений строк вас могут заинтересовать:

- строки, начинающиеся/заканчивающиеся определенным символом;
- строки, начинающиеся/заканчивающиеся некоторой подстрокой;
- строки, содержащие определенный символ в любом месте строки;

- строки, содержащие подстроку в любом месте строки;
- строки определенного формата, независимо от отдельных символов.

Вы можете создавать выражения поиска для идентификации этих и многих других строк совпадений с использованием подстановочных знаков, приведенных в табл. 4.4.

Таблица 4.4. Подстановочные символы

Подстановочный символ	Соответствие
_	В точности один символ
%	Любое количество символов (включая 0)

Символ подчеркивания заменяет один символ, а символ процента может заменять переменное количество символов. При создании условий, которые используют поисковые выражения, применяется оператор like, например:

```
mysql> SELECT last_name, first_name
-> FROM customer
-> WHERE last_name LIKE '_A_T%S';
+-----+
| last_name | first_name |
+-----+
| MATTHEWS | ERICA      |
| WALTERS  | CASSANDRA   |
| WATTS    | SHELLY      |
+-----+
3 rows in set (0.00 sec)
```

Выражение поиска в предыдущем примере определяет строки, содержащие букву A во второй позиции и T — в четвертой позиции, за которым следует любое количество символов, заканчивающееся буквой S. В табл. 4.5 показаны несколько выражений поиска и их интерпретации.

Таблица 4.5. Примеры выражений поиска

Выражение поиска	Интерпретация
F%	Строка, начинающаяся с F
%t	Строка, заканчивающаяся t
%bas%	Строка, содержащая подстроку 'bas'
t	Строка из 4 символов с t в третьей позиции
___ - ___	Строка из 11 символов с дефисами в четвертой и седьмой позициях

Подстановочные знаки отлично подходят для построения простых поисковых выражений; если же ваши потребности немного сложнее, можете

использовать несколько поисковых выражений, как показано в следующем запросе:

```
mysql> SELECT last_name, first_name
-> FROM customer
-> WHERE last_name LIKE 'Q%' OR last_name LIKE 'Y%';
+-----+-----+
| last_name | first_name |
+-----+-----+
| QUALLS    | STEPHEN    |
| QUIGLEY   | TROY       |
| QUINTANILLA | ROGER     |
| YANEZ      | LUIS        |
| YEE         | MARVIN     |
| YOUNG       | CYNTHIA    |
+-----+-----+
6 rows in set (0.00 sec)
```

Этот запрос находит всех клиентов, чьи фамилии начинаются с Q или Y.

Использование регулярных выражений

Если выяснится, что подстановочные знаки не обеспечивают достаточной гибкости, можно использовать для построения поисковых выражений регулярные выражения. Регулярное выражение — это, по сути, поисковое выражение на стероидах. Если вы новичок в SQL, но писали программы с помощью таких языков программирования, как Perl, то вы, возможно, уже хорошо знакомы с регулярными выражениями. Если же вы никогда их не использовали, возможно, вам стоит обратиться к книге Бен Форта (Ben Forta) *Изучаем регулярные выражения* (Диалектика, 2019), поскольку это слишком большая тема, чтобы охватить ее в этой книге.

Предыдущий запрос (найти всех клиентов, чьи фамилии начинаются с Q или Y) с использованием регулярных выражений MySQL будет иметь следующий вид:

```
mysql> SELECT last_name, first_name
-> FROM customer
-> WHERE last_name REGEXP '^[QY]';
+-----+-----+
| last_name | first_name |
+-----+-----+
| YOUNG     | CYNTHIA    |
| QUALLS    | STEPHEN    |
| QUINTANILLA | ROGER     |
| YANEZ      | LUIS        |
| YEE         | MARVIN     |
+-----+-----+
```

```
| QUIGLEY      | TROY      |
+-----+-----+
6 rows in set (0.16 sec)
```

Оператор `regexp` принимает регулярное выражение (в данном примере — `'^ [QY]'`) и применяет его к выражению в левой части условия (к столбцу `last_name`). Теперь запрос содержит одно условие, использующее регулярное выражение, а не два условия с использованием подстановочных знаков.

И Oracle Database, и Microsoft SQL Server также поддерживают регулярные выражения. В Oracle Database вы должны использовать функцию `regexp_like` оператора `regexp`, показанного в предыдущем примере, а в SQL Server такие выражения можно использовать с оператором `like`.

Этот таинственный `null`

Я откладывал эту тему так долго, как только мог. Но, наконец, пришло время затронуть тему, которая, как правило, вызывает страх, неуверенность и ужас: значение `null` — это отсутствие значения; например, до увольнения сотрудника его столбец `end_date` в таблице `employee` должен быть `null`. Просто не существует значения, которое могло бы быть присвоено столбцу `end_date` и иметь смысл в этой ситуации. Значение `null` на самом деле немного шире, так как есть различные случаи его применения.

Значение неприменимо

Например, столбец идентификатора сотрудника для транзакции, которая выполнена в банкомате.

Значение еще неизвестно

Например, когда федеральный идентификатор в момент создания строки клиента еще неизвестен.

Значение не определено

Например, когда создается учетная запись для товара, который еще не был добавлен в базу данных.



Некоторые теоретики утверждают, что должно быть другое выражение, которое охватывает каждую из этих (и многих других) ситуаций, но большинство практиков сходится на том, что наличие нескольких нулевых значений было бы слишком запутанным.

При работе с `null` следует помнить, что:

- выражение может быть null, но оно никогда не может быть *равным* null;
- два значения null никогда не равны одно другому.

Чтобы проверить, является ли выражение null, необходимо использовать оператор `is null`:

```
mysql> SELECT rental_id, customer_id
   -> FROM rental
   -> WHERE return_date IS NULL;
+-----+-----+
| rental_id | customer_id |
+-----+-----+
|    11496 |        155 |
|    11541 |        335 |
|    11563 |         83 |
|    11577 |        219 |
|    11593 |         99 |
| ...      |          |
|    15867 |        505 |
|    15875 |         41 |
|    15894 |        168 |
|    15966 |        374 |
+-----+-----+
183 rows in set (0.01 sec)
```

Этот запрос находит все фильмы, взятые напрокат, которые не были возвращены. А вот тот же запрос с использованием `= null` вместо `is null`:

```
mysql> SELECT rental_id, customer_id
   -> FROM rental
   -> WHERE return_date = NULL;
Empty set (0.01 sec)
```

Как видите, запрос анализируется и выполняется, но не возвращает никаких строк. Это распространенная ошибка неопытных программистов SQL; при этом сервер базы данных не предупредит вас об ошибке, поэтому будьте осторожны при построении условий, которые проверяют значение null!

Если вы хотите узнать, присвоено ли значение столбцу, можете использовать оператор `is not null`:

```
mysql> SELECT rental_id, customer_id, return_date
   -> FROM rental
   -> WHERE return_date IS NOT NULL;
+-----+-----+-----+
| rental_id | customer_id | return_date   |
+-----+-----+-----+
|       1 |        130 | 2005-05-26 22:04:30 |
|       2 |        459 | 2005-05-28 19:40:33 |
+-----+-----+-----+
```

```

|      3 |          408 | 2005-06-01 22:12:39 |
|      4 |          333 | 2005-06-03 01:43:41 |
|      5 |          222 | 2005-06-02 04:33:21 |
|      6 |          549 | 2005-05-27 01:32:07 |
|      7 |          269 | 2005-05-29 20:34:53 |
| ...
| 16043 |          526 | 2005-08-31 03:09:03 |
| 16044 |          468 | 2005-08-25 04:08:39 |
| 16045 |           14 | 2005-08-25 23:54:26 |
| 16046 |           74 | 2005-08-27 18:02:47 |
| 16047 |          114 | 2005-08-25 02:48:48 |
| 16048 |          103 | 2005-08-31 21:33:07 |
| 16049 |          393 | 2005-08-30 01:01:12 |
+-----+

```

15861 rows in set (0.02 sec)

Эта версия запроса возвращает все взятые напрокат фильмы, которые были возвращены, а это большинство строк в таблице (15861 из 16044).

Прежде чем на время отложить null, было бы полезно изучить еще одну потенциальную ловушку. Предположим, вас попросили найти все взятые напрокат фильмы, которые не вернулись с мая по август 2005 года. Вероятно, вашим первым желанием было бы написать следующий запрос:

```

mysql> SELECT rental_id, customer_id, return_date
-> FROM rental
-> WHERE return_date NOT BETWEEN '2005-05-01' AND '2005-09-01';
+-----+-----+-----+
| rental_id | customer_id | return_date       |
+-----+-----+-----+
|    15365 |          327 | 2005-09-01 03:14:17 |
|    15388 |           50 | 2005-09-01 03:50:23 |
|    15392 |          410 | 2005-09-01 01:14:15 |
|    15401 |          103 | 2005-09-01 03:44:10 |
|    15415 |          204 | 2005-09-01 02:05:56 |
| ...
|    15977 |          550 | 2005-09-01 22:12:10 |
|    15982 |          370 | 2005-09-01 21:51:31 |
|    16005 |          466 | 2005-09-02 02:35:22 |
|    16020 |          311 | 2005-09-01 18:17:33 |
|    16033 |          226 | 2005-09-01 02:36:15 |
|    16037 |           45 | 2005-09-01 02:48:04 |
|    16040 |          195 | 2005-09-02 02:19:33 |
+-----+-----+-----+

```

62 rows in set (0.01 sec)

Хотя эти 62 объекта на самом деле были возвращены за пределами окна с мая по август, внимательно посмотрев на данные, вы увидите, что все возвращенные строки имеют ненулевую дату возврата. А как же быть с 183 фильмами, которые так и не вернули? Можно сказать, что эти 183 строки также не

были возвращены в период с мая по август, поэтому они тоже должны быть включены в результирующий набор. Поэтому, чтобы правильно ответить на вопрос, необходимо учитывать возможность того, что некоторые строки могут содержать в столбце `return_date` значение `null`:

```
mysql> SELECT rental_id, customer_id, return_date
-> FROM rental
-> WHERE return_date IS NULL
-> OR return_date NOT BETWEEN '2005-05-01' AND '2005-09-01';
+-----+-----+-----+
| rental_id | customer_id | return_date |
+-----+-----+-----+
| 11496 | 155 | NULL |
| 11541 | 335 | NULL |
| 11563 | 83 | NULL |
| 11577 | 219 | NULL |
| 11593 | 99 | NULL |
| ... |
| 15939 | 382 | 2005-09-01 17:25:21 |
| 15942 | 210 | 2005-09-01 18:39:40 |
| 15966 | 374 | NULL |
| 15971 | 187 | 2005-09-02 01:28:33 |
| 15973 | 343 | 2005-09-01 20:08:41 |
| 15977 | 550 | 2005-09-01 22:12:10 |
| 15982 | 370 | 2005-09-01 21:51:31 |
| 16005 | 466 | 2005-09-02 02:35:22 |
| 16020 | 311 | 2005-09-01 18:17:33 |
| 16033 | 226 | 2005-09-01 02:36:15 |
| 16037 | 45 | 2005-09-01 02:48:04 |
| 16040 | 195 | 2005-09-02 02:19:33 |
+-----+-----+-----+
245 rows in set (0.01 sec)
```

В результирующий набор теперь входят 62 фильма, которые были возвращены вне окна с мая по август, а также 183 фильма, которые так и не были возвращены, — в общей сложности 245 строки. Хорошая идея при работе с базой данных, с которой вы не знакомы, — узнать, какие столбцы в таблице допускают нулевые значения, чтобы принять соответствующие меры при написании условий фильтрации и не пропустить нужные данные.

Проверьте свои знания

Предлагаемые здесь упражнения призваны закрепить понимание вами условий фильтрации. Ответы к упражнениям представлены в приложении Б.

В первых двух упражнениях вам потребуется следующее подмножество строк из таблицы `payment`:

payment_id	customer_id	amount	date(payment_date)
101	4	8.99	2005-08-18
102	4	1.99	2005-08-19
103	4	2.99	2005-08-20
104	4	6.99	2005-08-20
105	4	4.99	2005-08-21
106	4	2.99	2005-08-22
107	4	1.99	2005-08-23
108	5	0.99	2005-05-29
109	5	6.99	2005-05-31
110	5	1.99	2005-05-31
111	5	3.99	2005-06-15
112	5	2.99	2005-06-16
113	5	4.99	2005-06-17
114	5	2.99	2005-06-19
115	5	4.99	2005-06-20
116	5	4.99	2005-07-06
117	5	2.99	2005-07-08
118	5	4.99	2005-07-09
119	5	5.99	2005-07-09
120	5	1.99	2005-07-09

УПРАЖНЕНИЕ 4.1

Какие из идентификаторов платежей будут возвращены при следующих условиях фильтрации?

```
customer_id <> 5
AND (amount > 8 OR date(payment_date) = '2005-08-23')
```

УПРАЖНЕНИЕ 4.2

Какие из идентификаторов платежей будут возвращены при следующих условиях фильтрации?

```
customer_id = 5 AND
NOT (amount > 6 OR date(payment_date) = '2005-06-19')
```

УПРАЖНЕНИЕ 4.3

Создайте запрос, который извлекает из таблицы payments все строки, в которых сумма равна 1,98, 7,98 или 9,98.

УПРАЖНЕНИЕ 4.4

Создайте запрос, который находит всех клиентов, в фамилиях которых содержатся буква A во второй позиции и буква W — в любом месте после A.

Запросы к нескольким таблицам

Еще в главе 2, “Создание и наполнение базы данных”, я демонстрировал, как связанные концепции разбиваются на отдельные части с помощью процесса, известного как нормализация. Конечным результатом этого упражнения были две таблицы: `person` и `favorite_food`. Если, однако, вы хотите создать единый отчет с указанием имени, адреса и любимой еды человека, вам понадобится механизм, который снова соберет данные из этих двух таблиц; этот механизм известен как *соединение* (*join*), и в этой главе основное внимание уделяется простейшему и наиболее распространенному соединению — *внутреннему* (*inner join*). В главе 10, “Соединения”, демонстрируются все типы соединений.

Что такое соединение

Запросы к одной таблице — это, конечно, не редкость, но вы обнаружите, что для большинства ваших запросов потребуется две, три или даже больше таблиц. Для иллюстрации давайте рассмотрим определения таблиц `customer` и `address`, а затем определим запрос, который извлекает данные из обеих таблиц:

```
mysql> desc customer;
+-----+-----+-----+-----+
| Field      | Type           | Null | Key | Default   |
+-----+-----+-----+-----+
| customer_id | smallint(5) unsigned | NO   | PRI | NULL      |
| store_id    | tinyint(3) unsigned | NO   | MUL | NULL      |
| first_name  | varchar(45)        | NO   |     | NULL      |
| last_name   | varchar(45)        | NO   | MUL | NULL      |
| email       | varchar(50)         | YES  |     | NULL      |
| address_id  | smallint(5) unsigned | NO   | MUL | NULL      |
| active      | tinyint(1)          | NO   |     | 1          |
| create_date | datetime         | NO   |     | NULL      |
| last_update | timestamp        | YES  |     | CURRENT_TIMESTAMP |
+-----+-----+-----+-----+
```

```
mysql> desc address;
+-----+-----+-----+-----+
| Field | Type | Null | Key | Default |
+-----+-----+-----+-----+
| address_id | smallint(5) unsigned | NO | PRI | NULL |
| address | varchar(50) | NO | | NULL |
| address2 | varchar(50) | YES | | NULL |
| district | varchar(20) | NO | | NULL |
| city_id | smallint(5) unsigned | NO | MUL | NULL |
| postal_code | varchar(10) | YES | | NULL |
| phone | varchar(20) | NO | | NULL |
| location | geometry | NO | MUL | NULL |
| last_update | timestamp | NO | | CURRENT_TIMESTAMP |
+-----+-----+-----+-----+
```

Допустим, вы хотите получить имя и фамилию каждого клиента, а также его почтовый адрес. Таким образом, ваш запрос должен будет получить столбцы `customer.first_name`, `customer.last_name` и `address.address`. Но как получить данные из обеих таблиц в одном запросе? Ответ кроется в столбце `customer.address_id`, содержащем идентификатор записи клиента в таблице `address` (более формально — столбец `customer.address_id` является внешним ключом к таблице `address`). Запрос, который вы вскоре увидите, предписывает серверу использовать столбец `customer.address_id` в качестве *транспорта* между таблицами `customer` и `address`, что позволяет включать в результирующий набор запроса столбцы из обеих таблиц. Этот тип операции известен как *соединение* (join).



При желании можно создать ограничение внешнего ключа для гарантии того, что значения в одной таблице существуют в другой таблице. В предыдущем примере ограничение внешнего ключа может быть создано в таблице `customer`, чтобы гарантировать, что все значения, вставленные в столбец `customer.address_id`, можно найти в столбце `address.address_id`. Пожалуйста, обратите внимание, что ограничение внешнего ключа не является обязательным условием, чтобы соединить две таблицы.

Декартово произведение

Самый простой способ начать решать поставленную задачу — поместить таблицы `customer` и `address` в предложение `from` запроса и посмотреть, что получится. Вот запрос, который извлекает имена и фамилии клиентов вместе

с почтовым адресом, с предложением `from`, указывающим обе таблицы, разделенные ключевым словом `JOIN`:

```
mysql> SELECT c.first_name, c.last_name, a.address
-> FROM customer c JOIN address a;
+-----+-----+-----+
| first_name | last_name | address          |
+-----+-----+-----+
| MARY       | SMITH     | 47 MySakila Drive   |
| PATRICIA   | JOHNSON   | 47 MySakila Drive   |
| LINDA      | WILLIAMS  | 47 MySakila Drive   |
| BARBARA    | JONES     | 47 MySakila Drive   |
| ELIZABETH  | BROWN     | 47 MySakila Drive   |
| JENNIFER   | DAVIS     | 47 MySakila Drive   |
| MARIA      | MILLER    | 47 MySakila Drive   |
| SUSAN      | WILSON    | 47 MySakila Drive   |
| ...        |           |                   |
| SETH        | HANNON    | 1325 Fukuyama Street |
| KENT        | ARSENAULT | 1325 Fukuyama Street |
| TERRANCE   | ROUSH     | 1325 Fukuyama Street |
| RENE        | MCALISTER | 1325 Fukuyama Street |
| EDUARDO    | HIATT     | 1325 Fukuyama Street |
| TERRENCE   | GUNDERSON | 1325 Fukuyama Street |
| ENRIQUE    | FORSYTHE  | 1325 Fukuyama Street |
| FREDDIE    | DUGGAN    | 1325 Fukuyama Street |
| WADE        | DELVALLE  | 1325 Fukuyama Street |
| AUSTIN     | CINTRON   | 1325 Fukuyama Street |
+-----+-----+-----+
361197 rows in set (0.03 sec)
```

Гм... Всего имеется 599 клиентов, а в таблице `address` 603 строки, так откуда же в результирующем наборе 361 197 строк? Присмотревшись, можно увидеть, что многие клиенты имеют один и тот же адрес. Поскольку в запросе не указано, как именно должны быть соединены две таблицы, сервер базы данных генерировал *декартово произведение*, которое представляет собой *все возможные сочетания записей из двух таблиц* ($599 \text{ клиентов} \times 603 \text{ адреса} = 361\,197 \text{ сочетаний}$). Этот тип соединения известен как *перекрестное соединение* (cross join) и используется очень редко (по крайней мере, преднамеренно). Перекрестные соединения — один из типов соединений, которые мы рассмотрим в главе 10, “Соединения”.

Внутренние соединения

Чтобы изменить предыдущий запрос так, чтобы для каждого клиента возвращалась только одна строка, нужно правильно описать, как именно связаны эти две таблицы. Ранее я показал, что столбец `customer.address_id`

служит связующим звеном между двумя таблицами, поэтому необходимо добавить эту информацию в подпредложение `on` предложения `from`:

```
mysql> SELECT c.first_name, c.last_name, a.address
    -> FROM customer c JOIN address a
    -> ON c.address_id = a.address_id;
+-----+-----+-----+
| first_name | last_name | address
+-----+-----+-----+
| MARY       | SMITH     | 1913 Hanoi Way
| PATRICIA   | JOHNSON   | 1121 Loja Avenue
| LINDA      | WILLIAMS  | 692 Joliet Street
| BARBARA    | JONES     | 1566 Inegl Manor
| ELIZABETH  | BROWN     | 53 Idfu Parkway
| JENNIFER   | DAVIS      | 1795 Santiago de Compostela Way
| MARIA      | MILLER    | 900 Santiago de Compostela Parkway
| SUSAN      | WILSON    | 478 Joliet Way
| MARGARET   | MOORE     | 613 Korolev Drive
| ...
| TERRANCE   | ROUSH     | 42 Fontana Avenue
| RENE       | MCALISTER | 1895 Zhezqazghan Drive
| EDUARDO    | HIATT     | 1837 Kaduna Parkway
| TERRENCE   | GUNDERSON | 844 Bucuresti Place
| ENRIQUE    | FORSYTHE  | 1101 Bucuresti Boulevard
| FREDDIE    | DUGGAN    | 1103 Quilmes Boulevard
| WADE       | DELVALLE  | 1331 Usak Boulevard
| AUSTIN     | CINTRON   | 1325 Fukuyama Street
+-----+-----+-----+
```

599 rows in set (0.00 sec)

Теперь вместо 361 197 строк у нас имеются ожидаемые 599 строк — благодаря добавлению подпредложения `on`, которое указывает серверу необходимость соединения таблиц `customer` и `address`, используя столбец `address_id` для перехода от одной таблицы к другой. Например, строка Mary Smith в таблице `customer` содержит значение 5 в столбце `address_id` (в примере не показано). Сервер использует это значение для поиска в таблице `address` строки, имеющей значение 5 в столбце `address_id`, а затем извлекает значение '1913 Hanoi Way' из столбца `address` в этой строке.

Если значение имеется в столбце `address_id` одной таблицы, но *отсутствует* в другой, то соединение для строк, содержащих это значение, не выполняется и эти строки исключаются из результирующего набора. Этот тип соединения известен как *внутреннее соединение* и является наиболее часто используемым типом соединения. В качестве примера, если строка в таблице `customer` имеет значение 999 в столбце `address_id`, а в таблице `address` нет строки со значением 999 в столбце `address_id`, то эта строка клиента не

будет включена в результирующий набор. Если же вы хотите включить все строки из одной таблицы, независимо от того, имеется ли соответствующее значение в другой, вам необходимо указать *внешнее соединение*, но мы отложим рассмотрение этого типа соединения до главы 10, “Соединения”.

В предыдущем примере я не указывал в предложении `from`, какой тип соединения использовать. Однако, чтобы соединить две таблицы с использованием внутреннего соединения, необходимо явно указать это в своем предложении `from`; вот тот же пример с добавлением типа соединения (обратите внимание на ключевое слово `inner`):

```
SELECT c.first_name, c.last_name, a.address
FROM customer c INNER JOIN address a
ON c.address_id = a.address_id;
```

Если вы не укажете тип соединения, сервер по умолчанию выполнит внутреннее соединение. Однако, как вы увидите далее в этой книге, существует несколько типов соединений, поэтому лучше иметь привычку указывать точный тип нужного вам соединения, что будет добрым делом для других людей, которые могут использовать или поддерживать ваши запросы в будущем.

Если имена столбцов, используемых для соединения двух таблиц, идентичны (как это было в предыдущем запросе), вместо подпредложения `on` можно использовать подпредложение `using`:

```
SELECT c.first_name, c.last_name, a.address
FROM customer c INNER JOIN address a
USING (address_id);
```

Поскольку `using` — это сокращенная запись, которую можно использовать только в определенной ситуации, я предпочитаю всегда использовать подпредложение `on`, чтобы избежать путаницы.

Синтаксис соединения ANSI

Обозначения, используемые в этой книге для соединения таблиц, были введены в версии SQL92 стандарта ANSI SQL. Все основные базы данных (Oracle Database, Microsoft SQL Server, MySQL, IBM DB2 Universal Database и Sybase Adaptive Server) используют синтаксис соединения SQL92. Поскольку большинство этих серверов уже существовали во время выпуска спецификации SQL92, все они включают также более старый синтаксис соединения. Например, все эти серверы будут понимать следующий вариант предыдущего запроса:

```

mysql> SELECT c.first_name, c.last_name, a.address
-> FROM customer c, address a
-> WHERE c.address_id = a.address_id;
+-----+-----+-----+
| first_name | last_name | address
+-----+-----+-----+
| MARY       | SMITH     | 1913 Hanoi Way
| PATRICIA   | JOHNSON   | 1121 Loja Avenue
| LINDA      | WILLIAMS  | 692 Joliet Street
| BARBARA    | JONES     | 1566 Inegl Manor
| ELIZABETH  | BROWN     | 53 Idfu Parkway
| JENNIFER   | DAVIS     | 1795 Santiago de Compostela Way
| MARIA      | MILLER    | 900 Santiago de Compostela Parkway
| SUSAN      | WILSON    | 478 Joliet Way
| MARGARET   | MOORE     | 613 Korolev Drive
| ...
| TERRANCE   | ROUSH     | 42 Fontana Avenue
| RENE       | MCALISTER | 1895 Zhezqazghan Drive
| EDUARDO    | HIATT     | 1837 Kaduna Parkway
| TERRENCE   | GUNDERSON | 844 Bucuresti Place
| ENRIQUE    | FORSYTHE  | 1101 Bucuresti Boulevard
| FREDDIE    | DUGGAN    | 1103 Quilmes Boulevard
| WADE       | DELVALLE  | 1331 Usak Boulevard
| AUSTIN     | CINTRON   | 1325 Fukuyama Street
+-----+-----+-----+

```

599 rows in set (0.00 sec)

Этот старый метод определения соединений не включает подпредложение `on`; вместо него таблицы перечислены в предложении `from` через запятую, а условие соединения находится в предложении `where`. Хотя вы можете решить игнорировать синтаксис SQL92 и пользоваться более старым синтаксисом, синтаксис соединения ANSI имеет следующие преимущества.

- Условия соединения и условия фильтрации разделены на два разных предложения (подпредложение `on` и предложение `where` соответственно), что упрощает понимание запроса.
- Условия соединения для каждой пары таблиц содержатся в собственном предложении `on`, что снижает вероятность того, что некоторая часть соединения будет ошибочно пропущена.
- Запросы, использующие синтаксис соединения SQL92, переносимы между серверами баз данных, тогда как старый синтаксис на разных серверах немного отличается.

Преимущества синтаксиса соединения SQL92 проще увидеть для сложных запросов, которые включают как условия соединения, так и условия

фильтрации. Рассмотрим следующий запрос, который возвращает только тех клиентов, почтовый индекс которых — 52137:

```
mysql> SELECT c.first_name, c.last_name, a.address
-> FROM customer c, address a
-> WHERE c.address_id = a.address_id
-> AND a.postal_code = 52137;
+-----+-----+
| first_name | last_name | address
+-----+-----+
| JAMES      | GANNON    | 1635 Kuwana Boulevard |
| FREDDIE    | DUGGAN    | 1103 Quilmes Boulevard |
+-----+-----+
2 rows in set (0.01 sec)
```

На первый взгляд не так просто определить, какие условия в предложении `where` являются условиями соединения, а какие — условиями фильтрации. Также не совсем очевидно, какой тип соединения используется (чтобы определить тип соединения, нужно внимательно ознакомиться с условиями соединения в предложении `where`, чтобы увидеть, используются ли какие-либо специальные символы). Нелегко также определить, были ли какие-либо условия соединения ошибочно опущены. Вот тот же запрос с использованием синтаксиса соединения SQL92:

```
mysql> SELECT c.first_name, c.last_name, a.address
-> FROM customer c INNER JOIN address a
-> ON c.address_id = a.address_id
-> WHERE a.postal_code = 52137;
+-----+-----+
| first_name | last_name | address
+-----+-----+
| JAMES      | GANNON    | 1635 Kuwana Boulevard |
| FREDDIE    | DUGGAN    | 1103 Quilmes Boulevard |
+-----+-----+
2 rows in set (0.00 sec)
```

В этой версии запроса очевидно, какое условие используется для соединения, а какое — для фильтрации. Надеюсь, вы согласитесь, что легче понять версию, использующую синтаксис соединения SQL92.

Соединение трех и более таблиц

Соединение трех таблиц аналогично соединению двух таблиц, но с одним небольшим отличием. При соединении двух таблиц имеются две таблицы и один тип соединения в предложении `from`, а также одно подпредложение `on`, определяющее, как таблицы соединяются. При соединении трех таблиц

в предложении `from` есть три таблицы, два типа соединения и два подпредложения `on`.

Чтобы проиллюстрировать это, давайте изменим предыдущий запрос, чтобы возвращать город клиента, а не его почтовый адрес. Однако название города в таблице `address` не хранится, а доступно через внешний ключ к таблице `city`. Вот как выглядят определения таблиц:

```
mysql> desc address;
+-----+-----+-----+-----+
| Field | Type | Null | Key | Default |
+-----+-----+-----+-----+
| address_id | smallint(5) unsigned | NO | PRI | NULL |
| address | varchar(50) | NO | | NULL |
| address2 | varchar(50) | YES | | NULL |
| district | varchar(20) | NO | | NULL |
| city_id | smallint(5) unsigned | NO | MUL | NULL |
| postal_code | varchar(10) | YES | | NULL |
| phone | varchar(20) | NO | | NULL |
| location | geometry | NO | MUL | NULL |
| last_update | timestamp | NO | | CURRENT_TIMESTAMP |
+-----+-----+-----+-----+
mysql> desc city;
+-----+-----+-----+-----+
| Field | Type | Null | Key | Default |
+-----+-----+-----+-----+
| city_id | smallint(5) unsigned | NO | PRI | NULL |
| city | varchar(50) | NO | | NULL |
| country_id | smallint(5) unsigned | NO | MUL | NULL |
| last_update | timestamp | NO | | CURRENT_TIMESTAMP |
+-----+-----+-----+-----+
```

Чтобы показать город каждого клиента, необходимо перейти от таблицы `customer` к таблице `address`, используя столбец `address_id`, а затем — от таблицы `address` к таблице `city` с использованием столбца `city_id`. Запрос будет выглядеть следующим образом:

```
mysql> SELECT c.first_name, c.last_name, ct.city
   -> FROM customer c
   -> INNER JOIN address a
   -> ON c.address_id = a.address_id
   -> INNER JOIN city ct
   -> ON a.city_id = ct.city_id;
+-----+-----+-----+
| first_name | last_name | city |
+-----+-----+-----+
| JULIE      | SANCHEZ    | A Corua (La Corua) |
| PEGGY      | MYERS      | Abha                |
| TOM        | MILNER     | Abu Dhabi            |
+-----+-----+-----+
```

GLEN	TALBERT	Acua
LARRY	THRASHER	Adana
SEAN	DOUGLASS	Addis Abeba
...		
MICHELE	GRANT	Yuncheng
GARY	COY	Yuzhou
PHYLLIS	FOSTER	Zalantun
CHARLENE	ALVAREZ	Zanzibar
FRANKLIN	TROUTMAN	Zaoyang
FLOYD	GANDY	Zapopan
CONSTANCE	REID	Zaria
JACK	FOUST	Zeleznogorsk
BYRON	BOX	Zhezqazghan
GUY	BROWNLEE	Zhoushan
RONNIE	RICKETTS	Ziguinchor

599 rows in set (0.03 sec)

Для этого запроса используются три таблицы, два типа соединения и два подпредложения `on` в `from`, так что все стало выглядеть более запутанным. На первый взгляд может показаться, что порядок, в котором таблицы появляются в предложении `from`, важен, но если вы измените порядок таблиц, то получите точно такие же результаты. Все три приведенных ниже варианта запроса возвращают одни и те же результаты:

```
SELECT c.first_name, c.last_name, ct.city
FROM customer c
    INNER JOIN address a
        ON c.address_id = a.address_id
    INNER JOIN city ct
        ON a.city_id = ct.city_id;

SELECT c.first_name, c.last_name, ct.city
FROM city ct
    INNER JOIN address a
        ON a.city_id = ct.city_id
    INNER JOIN customer c
        ON c.address_id = a.address_id;

SELECT c.first_name, c.last_name, ct.city
FROM address a
    INNER JOIN city ct
        ON a.city_id = ct.city_id
    INNER JOIN customer c
        ON c.address_id = a.address_id;
```

Единственное различие, которое можно увидеть, — это порядок, в котором возвращаются строки, поскольку в запросах нет предложения `order by`, указывающего, как должны быть упорядочены результаты.

Имеет ли значение порядок соединения

Если вы не понимаете, почему все три версии запроса `customer/address/city` дают одни и те же результаты, вспомните, что SQL не является процедурным языком. Это означает, что вы описываете, что хотите получить и какие объекты базы данных должны быть вовлечены в запрос, но как лучше всего выполнить ваш запрос, должен определить сервер базы данных. Используя статистику, собранную из объектов вашей базы данных, сервер должен выбрать одну из трех таблиц в качестве отправной точки (выбранная таблица в дальнейшем называется *ведущей таблицей* (*driving table*)), а затем решить, в каком порядке соединять с ней оставшиеся таблицы. Следовательно, порядок, в котором таблицы появляются в вашем предложении `from`, значения не имеет.

Однако, если вы считаете, что таблицы в вашем запросе всегда следует соединять в определенном порядке, можете разместить таблицы в желаемом порядке, а затем указать ключевое слово `straight_join` в MySQL, запросить опцию `force order` в SQL Server или использовать подсказку оптимизатора `ordered` или `leading` в Oracle Database. Например, чтобы указать серверу MySQL использовать в качестве ведущей таблицы `city`, а затем присоединить таблицы `address` и `customer`, можно сделать следующее:

```
SELECT STRAIGHT_JOIN c.first_name, c.last_name, ct.city
FROM city ct
    INNER JOIN address a
        ON a.city_id = ct.city_id
    INNER JOIN customer c
        ON c.address_id = a.address_id
```

Использование подзапросов в качестве таблиц

Вы уже видели несколько примеров запросов, включающих несколько таблиц, но стоит упомянуть еще один вариант: что делать, если некоторые из наборов данных генерированы подзапросами? Подзапросы — это основная тема главы 9, “Подзапросы”, но я уже знакомил вас с этой концепцией в предыдущей главе. Следующий запрос соединяет таблицу `customer` с подзапросом к таблицам `address` и `city`:

```
mysql> SELECT c.first_name, c.last_name, addr.address, addr.city
-> FROM customer c
->     INNER JOIN
->         (SELECT a.address_id, a.address, ct.city
->             FROM address a
->                 INNER JOIN city ct
->                     ON a.city_id = ct.city_id
```

```

->      WHERE a.district = 'California'
->      ) addr
->  ON c.address_id = addr.address_id;
+-----+-----+-----+-----+
| first_name | last_name | address           | city
+-----+-----+-----+-----+
| PATRICIA   | JOHNSON   | 1121 Loja Avenue    | San Bernardino |
| BETTY      | WHITE      | 770 Bydgoszcz Avenue | Citrus Heights  |
| ALICE      | STEWART    | 1135 Izumisano Parkway | Fontana          |
| ROSA       | REYNOLDS   | 793 Cam Ranh Avenue | Lancaster        |
| RENEE      | LANE       | 533 al-Ayn Boulevard | Compton          |
| KRISTIN    | JOHNSTON   | 226 Brest Manor    | Sunnyvale        |
| CASSANDRA  | WALTERS    | 920 Kumbakonam Loop | Salinas          |
| JACOB      | LANCE      | 1866 al-Qatif Avenue | El Monte         |
| RENE       | MCALISTER  | 1895 Zhezqazghan Drive | Garden Grove    |
+-----+-----+-----+-----+
9 rows in set (0.00 sec)

```

Подзапрос, который начинается в строке 4 и имеет псевдоним `addr`, находит все адреса в Калифорнии. Внешний запрос соединяет результаты подзапроса с таблицей `customer`, чтобы вернуть имя, фамилию, почтовый адрес и город для всех клиентов, живущих в Калифорнии. Хотя этот запрос можно было бы написать без использования подзапроса, просто соединив три таблицы, иногда применение подзапроса может быть выгодным с точки зрения производительности и/или удобочитаемости.

Один из способов визуализировать происходящее — выполнить подзапрос сам по себе и посмотреть на полученные результаты. Вот результаты подзапроса из предыдущего примера:

```

mysql> SELECT a.address_id, a.address, ct.city
->   FROM address a
->     INNER JOIN city ct
->   ON a.city_id = ct.city_id
-> WHERE a.district = 'California';
+-----+-----+-----+
| address_id | address           | city
+-----+-----+-----+
|       6    | 1121 Loja Avenue    | San Bernardino |
|     18    | 770 Bydgoszcz Avenue | Citrus Heights  |
|     55    | 1135 Izumisano Parkway | Fontana          |
|    116    | 793 Cam Ranh Avenue | Lancaster        |
|    186    | 533 al-Ayn Boulevard | Compton          |
|    218    | 226 Brest Manor    | Sunnyvale        |
|    274    | 920 Kumbakonam Loop | Salinas          |
|    425    | 1866 al-Qatif Avenue | El Monte         |
|    599    | 1895 Zhezqazghan Drive | Garden Grove    |
+-----+-----+-----+
9 rows in set (0.00 sec)

```

Этот результирующий набор состоит из всех девяти адресов в Калифорнии. При соединении с таблицей `customer` через столбец `address_id` результирующий набор будет содержать информацию о клиентах, которым принадлежат эти адреса.

Использование одной таблицы дважды

Соединяя несколько таблиц, вы можете обнаружить, что вам нужно соединиться с одной и той же таблицей более одного раза. В рассматриваемом нами примере базы данных, например, актеры связаны с фильмами, в которых они появлялись, через таблицу `film_actor`. Если вы хотите найти все фильмы, в которых фигурируют два конкретных актера, можете написать запрос, который соединяет таблицу `film` с таблицей `film_actor` и с таблицей `actor`:

```
mysql> SELECT f.title
    -> FROM film f
    ->     INNER JOIN film_actor fa
    ->         ON f.film_id = fa.film_id
    ->     INNER JOIN actor a
    ->         ON fa.actor_id = a.actor_id
    -> WHERE ((a.first_name = 'CATE' AND a.last_name = 'MCQUEEN')
    ->         OR (a.first_name = 'CUBA' AND a.last_name = 'BIRCH'));
+-----+
| title           |
+-----+
| ATLANTIS CAUSE |
| BLOOD ARGONAUTS |
| COMMANDMENTS EXPRESS |
| DYNAMITE TARZAN |
| EDGE KISSING |
| ...             |
| TOWERS HURRICANE |
| TROJAN TOMORROW |
| VIRGIN DAISY |
| VOLCANO TEXAS |
| WATERSHIP FRONTIER |
+-----+
54 rows in set (0.00 sec)
```

Этот запрос возвращает все фильмы, в которых снимались Cate McQueen или Cuba Birch. Предположим теперь, что требуется получить только те фильмы, в которых появляются оба актера. Для этого нужно найти все строки в таблице `films`, у которых есть две строки в таблице `film_actor`, одна из которых связана с Cate McQueen, а другая — с Cuba Birch. Следовательно,

требуется включить таблицы `film_actor` и `actor` дважды, каждый раз с иным псевдонимом, чтобы сервер знал, на что именно вы ссылаетесь в различных предложениях:

```
mysql> SELECT f.title
    ->   FROM film f
    ->   INNER JOIN film_actor fa1
    ->   ON f.film_id = fa1.film_id
    ->   INNER JOIN actor a1
    ->   ON fa1.actor_id = a1.actor_id
    ->   INNER JOIN film_actor fa2
    ->   ON f.film_id = fa2.film_id
    ->   INNER JOIN actor a2
    ->   ON fa2.actor_id = a2.actor_id
    -> WHERE (a1.first_name = 'CATE' AND a1.last_name = 'MCQUEEN')
    ->   AND (a2.first_name = 'CUBA' AND a2.last_name = 'BIRCH');
+-----+
| title      |
+-----+
| BLOOD ARGONAUTS |
| TOWERS HURRICANE |
+-----+
2 rows in set (0.00 sec)
```

Эти два актера снялись в 52 разных фильмах, но есть всего два фильма, в которых снялись оба актера. Это один из примеров запроса, для которого использование псевдонимов таблиц обязательно, поскольку одни и те же таблицы используются несколько раз.

Самосоединение

Вы можете не только включать одну и ту же таблицу в один и тот же запрос более одного раза, но и соединять таблицу с самой собой. Поначалу это может показаться странным, но для этого есть веские причины. Некоторые таблицы включают в себя *самоссылающиеся внешние ключи* (self-referencing foreign key); это означает, что в таблице имеется столбец, ссылающийся на первичный ключ в той же таблице. Хотя образец базы данных такую связь не включает, давайте представим, что в таблице `film` есть столбец `prequel_film_id`, который указывает на родительский фильм (например, фильм *Потерянный скрипач 2* будет использовать этот столбец, чтобы указать на фильм *Потерянный скрипач* как на родительский). Вот как выглядела бы таблица, если бы мы добавили этот дополнительный столбец:

```
mysql> desc film;
+-----+-----+-----+-----+
| Field | Type | Null | Key | Default |
+-----+-----+-----+-----+
| film_id | smallint(5) unsigned | NO | PRI | NULL |
| title | varchar(255) | NO | MUL | NULL |
| description | text | YES | | NULL |
| release_year | year(4) | YES | | NULL |
| language_id | tinyint(3) unsigned | NO | MUL | NULL |
| original_language_id | tinyint(3) unsigned | YES | MUL | NULL |
| rental_duration | tinyint(3) unsigned | NO | | 3 |
| rental_rate | decimal(4,2) | NO | | 4.99 |
| length | smallint(5) unsigned | YES | | NULL |
| replacement_cost | decimal(5,2) | NO | | 19.99 |
| rating | enum('G','PG', | | | |
| | 'PG-13','R','NC-17') | YES | | G |
| special_features | set('Trailers',..., | | | |
| | 'Behind the Scenes') | YES | | NULL |
| last_update | timestamp | NO | | CURRENT_TIMESTAMP |
| prequel_film_id | smallint(5) unsigned | YES | MUL | NULL |
+-----+-----+-----+-----+
```

Используя *самосоединение*, можно написать запрос, в котором будут перечислены все фильмы с приквелами, включая название приквела:

```
mysql> SELECT f.title, f_prnt.title prequel
   -> FROM film f
   -> INNER JOIN film f_prnt
   -> ON f_prnt.film_id = f.prequel_film_id
   -> WHERE f.prequel_film_id IS NOT NULL;
+-----+-----+
| title | prequel |
+-----+-----+
| FIDDLER LOST II | FIDDLER LOST |
+-----+-----+
1 row in set (0.00 sec)
```

Этот запрос соединяет таблицу `film` с самой собой с помощью внешнего ключа `prequel_film_id`. Псевдонимы таблицы `f` и `f_prnt` используются для того, чтобы было понятно, какая таблица и для какой цели используется.

Проверьте свои знания

Предлагаемые здесь упражнения призваны закрепить понимание вами внутренних соединений. Ответы к упражнениям представлены в приложении Б.

УПРАЖНЕНИЕ 5.1

Заполните пропущенные места (обозначенные как <#>) в следующем запросе так, чтобы получить показанные результаты:

```
mysql> SELECT c.first_name, c.last_name, a.address, ct.city
->   FROM customer c
->     INNER JOIN address <1>
->       ON c.address_id = a.address_id
->     INNER JOIN city ct
->       ON a.city_id = <2>
->   WHERE a.district = 'California';
+-----+-----+-----+-----+
| first_name | last_name | address          | city      |
+-----+-----+-----+-----+
| PATRICIA   | JOHNSON   | 1121 Loja Avenue    | San Bernardino |
| BETTY      | WHITE      | 770 Bydgoszcz Avenue | Citrus Heights  |
| ALICE      | STEWART    | 1135 Izumisano Parkway | Fontana      |
| ROSA        | REYNOLDS   | 793 Cam Ranh Avenue  | Lancaster    |
| RENEE      | LANE        | 533 al-Ayn Boulevard | Compton      |
| KRISTIN    | JOHNSTON   | 226 Brest Manor     | Sunnyvale    |
| CASSANDRA  | WALTERS    | 920 Kumbakonam Loop  | Salinas      |
| JACOB      | LANCE       | 1866 al-Qatif Avenue | El Monte     |
| RENE        | MCALISTER   | 1895 Zhezqazghan Drive | Garden Grove |
+-----+-----+-----+-----+
9 rows in set (0.00 sec)
```

УПРАЖНЕНИЕ 5.2

Напишите запрос, который выводил бы названия всех фильмов, в которых играл актер с именем JOHN.

УПРАЖНЕНИЕ 5.3

Создайте запрос, который возвращает все адреса в одном и том же городе. Вам нужно будет соединить таблицу адресов с самой собой, и каждая строка должна включать два разных адреса.

Работа с множествами

Хотя вы можете работать с данными в базе данных по одной строке за раз, реляционные базы данных в действительности основаны на множествах. В этой главе рассказывается, как можно объединить несколько результирующих наборов с использованием различных операторов для работы с множествами¹. После краткого обзора теории множеств я продемонстрирую, как использовать операторы для работы с множествами — `union`, `intersect` и `except` — для объединения нескольких наборов данных в единое целое.

Основы теории множеств

Во многих странах азы теории множеств включены в учебные программы начального уровня по математике. Возможно, вы вспомните, что уже видели что-то похожее на рис. 6.1.

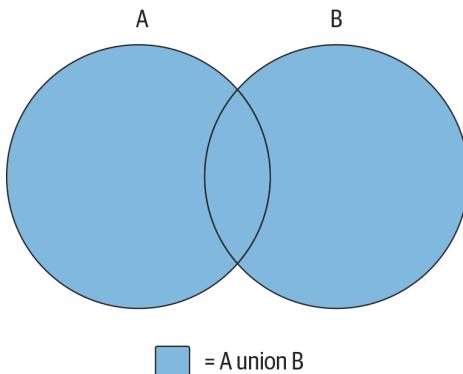


Рис. 6.1. Операция объединения

¹ Термины *множество* и *набор* в данной книге являются синонимами — просто термин “множество” применяется в первую очередь в математике, а “набор” — в базах данных. — Примеч. пер.

Заштрихованная область на рис. 6.1 представляет собой *объединение* множеств А и В, представляющее собой комбинацию двух множеств (при этом перекрывающиеся области включаются в результатирующее множество только один раз). Это начинает казаться знакомым? Если да, то вы имеете шанс применить школьные знания на практике. Если нет, не волнуйтесь, все это легко визуализируется с помощью пары диаграмм.

Используя круги для представления двух наборов данных (А и В), представим себе подмножество данных, которые являются общими для обоих наборов; эти общие данные представлены областью перекрытия кругов, показанной на рис. 6.1. Далее я использую одну и ту же диаграмму для иллюстрации всех операций над множествами. Еще одна операция над наборами данных дает только те данные, которые входят в оба набора одновременно; эта операция известна как *пересечение* и показана на рис. 6.2.

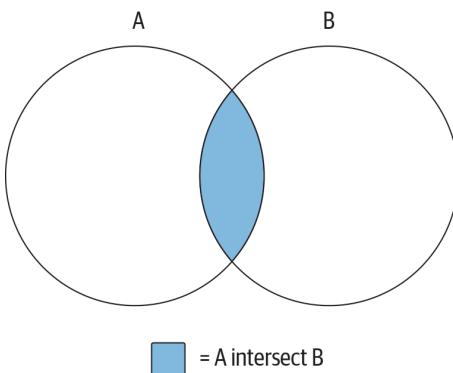


Рис. 6.2. Операция пересечения

Набор данных, сгенерированный пересечением наборов А и В, является областью перекрытия наборов — областью перекрытия кругов на рисунке. Если два набора не перекрываются, то операция пересечения дает пустой набор.

Третья, и последняя, операция над множествами, показанная на рис. 6.3, известна как операция *исключения* (except), или *разности*.

На рис. 6.3 показаны результаты выполнения операции А except В, которая представляет собой весь набор А, кроме тех данных из набора А, которые входят также в набор В (или, в геометрической интерпретации с кругами, круг множества А минус область пересечения кругов А и В). Если два набора не перекрываются, то операция А except В просто дает все множество А.

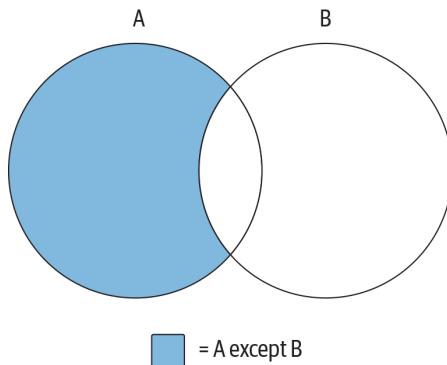


Рис. 6.3. Операция исключения

Используя эти три операции или их комбинации, можно генерировать любые требуемые результаты. Например, представим, что нужно построить набор, показанный на рис. 6.4.

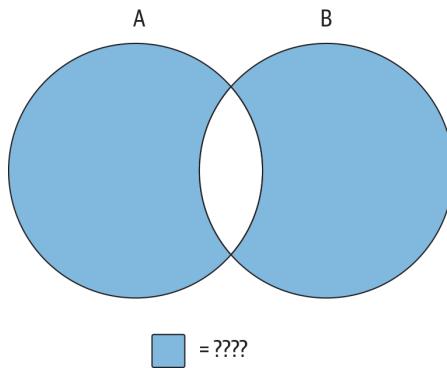


Рис. 6.4. Неизвестная операция

Набор данных, который вы ищете, включает все наборы A и B *без* области перекрытия. Достичь требуемого результата с помощью только одной из трех приведенных операций невозможно; вместо этого вам нужно сначала создать набор данных, который охватывает наборы A и B, а затем использовать вторую операцию для удаления области перекрытия. Объединенный набор описывается как A union B, а область перекрытия — как A intersect B, так что операция, необходимая для создания набора данных, представленного на рис. 6.4, выглядит следующим образом:

(A union B) except (A intersect B)

Конечно, часто есть несколько способов добиться одних и тех же результатов; результат, аналогичный показанному, можно получить с использованием следующей операции:

(A except B) union (B except A)

Эти концепции довольно легко понять с помощью диаграмм. В следующих разделах вы увидите, как эти концепции применяются к реляционной базе данных с помощью операторов SQL для работы с множествами.

Теория множеств на практике

Круги, использованные на диаграммах в предыдущем разделе для представления наборов данных, не отображают никакой информации о том, что содержат описываемые ими наборы данных. Однако при работе с фактическими данными, если они должны комбинироваться, необходимо описать состав задействованных в операциях наборов данных. Представьте себе, например, что произойдет, если вы попытаетесь сгенерировать объединение таблицы клиентов и таблицы города, определения которых следующие:

```
mysql> desc customer;
+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default |
+-----+-----+-----+-----+
| customer_id | smallint(5) unsigned | NO  | PRI | NULL    |
| store_id    | tinyint(3) unsigned | NO  | MUL | NULL    |
| first_name  | varchar(45)        | NO  |     | NULL    |
| last_name   | varchar(45)        | NO  | MUL | NULL    |
| email       | varchar(50)         | YES |     | NULL    |
| address_id  | smallint(5) unsigned | NO  | MUL | NULL    |
| active      | tinyint(1)          | NO  |     | 1       |
| create_date | datetime        | NO  |     | NULL    |
| last_update | timestamp       | YES |     | CURRENT_TIMESTAMP|
+-----+-----+-----+-----+
```

```
mysql> desc city;
+-----+-----+-----+-----+
| Field | Type          | Null | Key | Default |
+-----+-----+-----+-----+
| city_id | smallint(5) unsigned | NO  | PRI | NULL    |
| city    | varchar(50)        | NO  |     | NULL    |
| country_id | smallint(5) unsigned | NO  | MUL | NULL    |
| last_update | timestamp       | NO  |     | CURRENT_TIMESTAMP|
+-----+-----+-----+-----+
```

При объединении первый столбец в результирующем наборе будет включать столбцы `customer.customer_id` и `city.city_id`, второй столбец будет

сочетанием столбцов `customer.store_id` и `city.city` и т.д. В то время как одни пары столбцов легко комбинировать (например, два числовых столбца), не ясно, как следует комбинировать другие пары столбцов, например числовой столбец со строковым или строковый столбец со столбцом даты. Кроме того, столбцы 5–9 в объединенных таблицах будут включать данные только из столбцов 5–9 в таблице `customer`, поскольку в таблице `city` всего четыре столбца. Очевидно, что между двумя наборами данных, которые вы хотите комбинировать, должна быть какая-то общность.

Следовательно, при выполнении операций над двумя наборами данных должны применяться следующие рекомендации.

- Оба набора данных должны иметь одинаковое количество столбцов.
- Типы данных каждого столбца в двух наборах данных должны быть одинаковыми (либо сервер должен иметь возможность преобразовывать один тип в другой).

При выполнении этих правил легче представить себе, что на практике означают “перекрывающиеся данные”; каждая пара столбцов из двух объединяемых наборов должна содержать одну и ту же строку, номер или дату, чтобы строки в двух таблицах считались одинаковыми.

Вы выполняете операцию над множеством, помещая соответствующий *оператор множества* между двумя операторами `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 из отдельных запросов дает результирующий набор данных, состоящий из одной строки, состоящей из числового и строкового столбцов. Оператор множества, который в данном случае представляет собой `union`, сообщает серверу базы данных о необходимости объединить все строки из двух наборов. Таким образом, окончательный набор включает две строки по два столбца. Этот запрос известен как *составной запрос*, потому что он состоит из нескольких независимых запросов. Как вы увидите позже, составные запросы могут включать *большие* двух запросов, если для получения

окончательных результатов необходимо несколько операций над множествами.

Операторы для работы с множествами

Язык SQL включает три оператора для работы с множествами, которые позволяют выполнять операции над наборами данных, описанные в этой главе ранее. Кроме того, каждый такой оператор имеет два варианта: один включает дубликаты, а другой их удаляет (но необязательно *все*). В следующих подразделах описываются все упомянутые операторы и демонстрируется, как они используются.

Оператор union

Операторы `union` и `union all` позволяют объединять несколько наборов данных. Разница между ними заключается в том, что `union` сортирует объединенный набор и удаляет дубликаты, тогда как `union all` этого не делает. При выполнении `union all` количество строк в окончательных наборах данных всегда будет равно сумме количеств строк в объединяемых наборах данных. Это простейшая для выполнения операция (с точки зрения сервера), поскольку серверу не нужно проверять перекрывающиеся данные. Следующий пример демонстрирует, как можно использовать оператор `union all` для создания набора данных, состоящего из имен и фамилий из нескольких таблиц:

```
mysql> SELECT 'CUST' typ, c.first_name, c.last_name
   -> FROM customer c
   -> UNION ALL
   -> SELECT 'ACTR' typ, a.first_name, a.last_name
   -> FROM actor a;
+-----+-----+
| typ | first_name | last_name |
+-----+-----+
| CUST | MARY      | SMITH     |
| CUST | PATRICIA  | JOHNSON   |
| CUST | LINDA     | WILLIAMS |
| CUST | BARBARA   | JONES     |
| CUST | ELIZABETH | BROWN     |
| CUST | JENNIFER  | DAVIS     |
| CUST | MARIA     | MILLER    |
| CUST | SUSAN     | WILSON    |
| CUST | MARGARET  | MOORE     |
| CUST | DOROTHY   | TAYLOR    |
| CUST | LISA       | ANDERSON  |
| CUST | NANCY     | THOMAS    |
| CUST | KAREN     | JACKSON   |
+-----+-----+
```

```

| ...          |
| ACTR | BURT      | TEMPLE   |
| ACTR | MERYL     | ALLEN    |
| ACTR | JAYNE     | SILVERSTONE |
| ACTR | BELA       | WALKEN   |
| ACTR | REESE     | WEST     |
| ACTR | MARY       | KEITEL   |
| ACTR | JULIA     | FAWCETT  |
| ACTR | THORA     | TEMPLE   |
+-----+
799 rows in set (0.00 sec)

```

Запрос возвращает 799 имен, из которых 599 строк поступают из таблицы customer, а остальные 200 — из таблицы actor. Первый столбец имеет псевдоним typ. Он не обязательен и добавлен, чтобы показать источник каждого имени, возвращаемого запросом. Чтобы подчеркнуть, что оператор union all не удаляет дубликаты, вот другая версия предыдущего примера, с двумя идентичными запросами к таблице actor:

```

mysql> SELECT 'ACTR' typ, a.first_name, a.last_name
-> FROM actor a
-> UNION ALL
-> SELECT 'ACTR' typ, a.first_name, a.last_name
-> FROM actor a;
+-----+
| typ | first_name | last_name |
+-----+
| ACTR | PENELOPE  | GUINNESS |
| ACTR | NICK       | WAHLBERG |
| ACTR | ED         | CHASE    |
| ACTR | JENNIFER  | DAVIS    |
| ACTR | JOHNNY     | LOLLOBRIGIDA |
| ACTR | BETTE     | NICHOLSON |
| ACTR | GRACE     | MOSTEL   |
| ...
| ACTR | BURT       | TEMPLE   |
| ACTR | MERYL     | ALLEN    |
| ACTR | JAYNE     | SILVERSTONE |
| ACTR | BELA       | WALKEN   |
| ACTR | REESE     | WEST     |
| ACTR | MARY       | KEITEL   |
| ACTR | JULIA     | FAWCETT  |
| ACTR | THORA     | TEMPLE   |
+-----+
400 rows in set (0.00 sec)

```

Как видно из полученных результатов, 200 строк из таблицы actor включаются в результирующий набор дважды, так что он содержит 400 строк.

Конечно, вы вряд ли повторите в составном запросе один и тот же запрос дважды. Но вот другой составной запрос, возвращающий повторяющиеся данные:

```
mysql> SELECT c.first_name, c.last_name
   -> FROM customer c
   -> WHERE c.first_name LIKE 'J%' AND c.last_name LIKE 'D%'
   -> UNION ALL
   -> SELECT a.first_name, a.last_name
   -> FROM actor a
   -> WHERE a.first_name LIKE 'J%' AND a.last_name LIKE 'D%';
+-----+-----+
| first_name | last_name |
+-----+-----+
| JENNIFER   | DAVIS      |
| JENNIFER   | DAVIS      |
| JUDY        | DEAN       |
| JODIE       | DEGENERES |
| JULIANNE   | DENCH      |
+-----+-----+
5 rows in set (0.00 sec)
```

Оба запроса возвращают имена людей с инициалами JD. Из пяти строк в результирующем наборе одна из них является дубликатом (Jennifer Davis). Если вы хотите, чтобы в вашем результирующем наборе были *исключены* повторяющиеся строки, используйте оператор union вместо union all:

```
mysql> SELECT c.first_name, c.last_name
   -> FROM customer c
   -> WHERE c.first_name LIKE 'J%' AND c.last_name LIKE 'D%'
   -> UNION
   -> SELECT a.first_name, a.last_name
   -> FROM actor a
   -> WHERE a.first_name LIKE 'J%' AND a.last_name LIKE 'D%';
+-----+-----+
| first_name | last_name |
+-----+-----+
| JENNIFER   | DAVIS      |
| JUDY        | DEAN       |
| JODIE       | DEGENERES |
| JULIANNE   | DENCH      |
+-----+-----+
4 rows in set (0.00 sec)
```

В этой версии запроса в конечный результат включены только четыре различных имени, а не пять, возвращаемых при использовании union all.

Оператор `intersect`

Спецификация ANSI SQL включает оператор `intersect` для выполнения пересечений. К сожалению, MySQL версии 8.0 не реализует оператор `intersect`. Если вы используете Oracle Database или SQL Server 2008, то можете использовать `intersect`; поскольку для всех примеров в этой книге я использую MySQL, в результате примеры запросов в этом разделе не могут быть выполнены ни с какими версиями MySQL до версии 8.0 включительно. По той же причине я воздерживаюсь от показа подсказки `mysql>`, поскольку эти инструкции сервером MySQL не выполняются.

Если два запроса в составном запросе возвращают неперекрывающиеся наборы данных, пересечение будет пустым множеством. Рассмотрим следующий запрос:

```
SELECT c.first_name, c.last_name
FROM customer c
WHERE c.first_name LIKE 'D%' AND c.last_name LIKE 'T%'
INTERSECT
SELECT a.first_name, a.last_name
FROM actor a
WHERE a.first_name LIKE 'D%' AND a.last_name LIKE 'T%';
Empty set (0.04 sec)
```

Хотя есть и актеры, и клиенты с инициалами DT, эти наборы полностью не перекрывающиеся, поэтому пересечение этих двух наборов данных дает пустой набор. Если же вернуться к инициалам JD, то пересечение даст единственную строку:

```
SELECT c.first_name, c.last_name
FROM customer c
WHERE c.first_name LIKE 'J%' AND c.last_name LIKE 'D%'
INTERSECT
SELECT a.first_name, a.last_name
FROM actor a
WHERE a.first_name LIKE 'J%' AND a.last_name LIKE 'D%';
+-----+-----+
| first_name | last_name |
+-----+-----+
| JENNIFER   | DAVIS      |
+-----+-----+
1 row in set (0.00 sec)
```

Пересечение этих двух запросов дает Jennifer Davis — единственное имя, имеющееся в результирующих наборах обоих запросов.

Наряду с оператором `intersect`, который удаляет все повторяющиеся строки, обнаруженные в области перекрытия наборов данных, спецификация

ANSI SQL предлагает оператор `intersect all`, который дубликаты не удаляет. Единственный сервер базы данных, который в настоящее время реализует этот оператор, — это IBM DB2 Universal Server.

Оператор `except`

Спецификация ANSI SQL содержит оператор `except` для выполнения операции исключения. И вновь, к сожалению, MySQL версии 8.0 не реализует оператор `except`, поэтому для этого раздела справедливы все те же примечания, что и для предыдущего.



Если вы используете Oracle Database, вместо оператора `except` нужно использовать несовместимый со стандартом ANSI оператор `minus`.

Оператор `except` возвращает первый результирующий набор за вычетом любого перекрытия со вторым результирующим набором. Вот пример из предыдущего раздела, но с использованием `except` вместо `intersect` и с обратным порядком запросов:

```
SELECT a.first_name, a.last_name
FROM actor a
WHERE a.first_name LIKE 'J%' AND a.last_name LIKE 'D%'
EXCEPT
SELECT c.first_name, c.last_name
FROM customer c
WHERE c.first_name LIKE 'J%' AND c.last_name LIKE 'D%';
+-----+-----+
| first_name | last_name |
+-----+-----+
| JUDY       | DEAN      |
| JODIE      | DEGENERES |
| JULIANNE   | DENCH     |
+-----+-----+
3 rows in set (0.00 sec)
```

В этой версии запроса результирующий набор состоит из трех строк первого запроса минус Jennifer Davis — имя, которое находится в результирующих наборах обоих запросов. В спецификации ANSI SQL имеется также оператор `except all`, но он также реализован только в IBM DB2 Universal Server.

Оператор `except all` немного сложнее, поэтому вот пример, демонстрирующий, как обрабатываются повторяющиеся данные. Допустим, у вас есть два набора данных, которые выглядят следующим образом:

Набор А

actor_id
10
11
12
10
10

Набор В

actor_id
10
10

Операция A except B дает следующий результат:

actor_id
11
12

Если изменить операцию на A except all B, результат будет таким:

actor_id
10
11
12

Следовательно, разница между двумя операциями заключается в том, что except удаляет все вхождения повторяющихся данных из набора А, тогда как except all удаляет только одно из повторяющихся данных из набора А для каждого вхождения таких данных в набор В.

Правила применения операторов для работы с множествами

В следующих разделах описаны правила, которым рекомендуется следовать при работе с составными запросами.

Сортировка результатов составного запроса

Если результаты составного запроса должны быть отсортированы, можно добавить предложение `order by` после последнего запроса. При указании имен столбцов в предложении `order by` нужно выбрать одно из имен столбцов в первом запросе составного запроса. Часто имена столбцов для обоих запросов в составном запросе совпадают, но это необязательное условие, о чем свидетельствует следующий пример:

```
mysql> SELECT a.first_name fname, a.last_name lname
-> FROM actor a
-> WHERE a.first_name LIKE 'J%' AND a.last_name LIKE 'D%'
-> UNION ALL
-> SELECT c.first_name, c.last_name
-> FROM customer c
-> WHERE c.first_name LIKE 'J%' AND c.last_name LIKE 'D%'
-> ORDER BY lname, fname;
+-----+-----+
| fname | lname |
+-----+-----+
| JENNIFER | DAVIS |
| JENNIFER | DAVIS |
| JUDY | DEAN |
| JODIE | DEGENERES |
| JULIANNE | DENCH |
+-----+-----+
5 rows in set (0.00 sec)
```

Имена столбцов, указанные в этих двух запросах, различны. Если в предложении `order by` указать имя столбца из второго запроса, будет выведено следующее сообщение об ошибке:

```
mysql> SELECT a.first_name fname, a.last_name lname
-> FROM actor a
-> WHERE a.first_name LIKE 'J%' AND a.last_name LIKE 'D%'
-> UNION ALL
-> SELECT c.first_name, c.last_name
-> FROM customer c
-> WHERE c.first_name LIKE 'J%' AND c.last_name LIKE 'D%'
-> ORDER BY last_name, first_name;
ERROR 1054 (42S22): Unknown column 'last_name' in 'order clause'
```

Я рекомендую давать столбцам в обоих запросах одинаковые псевдонимы столбцов, чтобы избежать указанной проблемы.

Приоритеты операций над множествами

Если составной запрос содержит более двух запросов с использованием разных операторов для работы с множествами, следует подумать о порядке размещения запросов в составном запросе для достижения желаемых результатов. Рассмотрим следующий составной запрос, включающий три инструкции запроса:

```
mysql> SELECT a.first_name, a.last_name
-> FROM actor a
-> WHERE a.first_name LIKE 'J%' AND a.last_name LIKE 'D%'
-> UNION ALL
-> SELECT a.first_name, a.last_name
-> FROM actor a
-> WHERE a.first_name LIKE 'M%' AND a.last_name LIKE 'T%'
-> UNION
-> SELECT c.first_name, c.last_name
-> FROM customer c
-> WHERE c.first_name LIKE 'J%' AND c.last_name LIKE 'D%';
+-----+-----+
| first_name | last_name |
+-----+-----+
| JENNIFER   | DAVIS      |
| JUDY        | DEAN       |
| JODIE       | DEGENERES  |
| JULIANNE    | DENCH      |
| MARY        | TANDY     |
| MENA        | TEMPLE    |
+-----+-----+
6 rows in set (0.00 sec)
```

Этот составной запрос включает три запроса, которые возвращают наборы неуникальных имен; первый и второй запросы разделены оператором union all, а второй и третий запросы — оператором union. Хотя может показаться, что размещение этих операторов не имеет большого значения, на самом деле это не так. Вот тот же составной запрос, но с обратным размещением операторов:

```
mysql> SELECT a.first_name, a.last_name
-> FROM actor a
-> WHERE a.first_name LIKE 'J%' AND a.last_name LIKE 'D%'
-> UNION
-> SELECT a.first_name, a.last_name
-> FROM actor a
-> WHERE a.first_name LIKE 'M%' AND a.last_name LIKE 'T%'
-> UNION ALL
-> SELECT c.first_name, c.last_name
-> FROM customer c
```

```

-> WHERE c.first_name LIKE 'J%' AND c.last_name LIKE 'D%';
+-----+
| first_name | last_name |
+-----+
| JENNIFER   | DAVIS      |
| JUDY        | DEAN       |
| JODIE       | DEGENERES |
| JULIANNE    | DENCH      |
| MARY        | TANDY     |
| MENA        | TEMPLE    |
| JENNIFER   | DAVIS      |
+-----+
7 rows in set (0.00 sec)

```

Беглого взгляда на результаты достаточно, чтобы увидеть, что *важно*, как именно формируется составной запрос при использовании разных операторов работы с множествами. Как правило, составные запросы, содержащие три или более запросов, вычисляются в порядке сверху вниз, но со следующими предостережениями.

- Спецификация ANSI SQL требует, чтобы оператор `intersect` имел приоритет над другими операторами множества.
- Порядок, в котором выполняется объединение запросов, можно указать с помощью скобок.

MySQL пока что не позволяет использовать круглые скобки в составных запросах, но если вы используете другой сервер базы данных, то можете заключить смежные запросы в круглые скобки, чтобы переопределить стандартный порядок обработки составных запросов сверху вниз, например:

```

SELECT a.first_name, a.last_name
FROM actor a
WHERE a.first_name LIKE 'J%' AND a.last_name LIKE 'D%'
UNION
(SELECT a.first_name, a.last_name
FROM actor a
WHERE a.first_name LIKE 'M%' AND a.last_name LIKE 'T%')
UNION ALL
SELECT c.first_name, c.last_name
FROM customer c
WHERE c.first_name LIKE 'J%' AND c.last_name LIKE 'D%'
)

```

Для этого составного запроса второй и третий запросы будут объединены с использованием оператора `union all`, после чего полученные результаты будут объединены с первым запросом с помощью оператора `union`.

Проверьте свои знания

Предлагаемые здесь упражнения призваны закрепить понимание вами работы с множествами. Ответы к упражнениям представлены в приложении Б.

УПРАЖНЕНИЕ 6.1

Пусть множество A = {L,M,N,O,P}, а множество B = {P,Q,R,S,T}. Какие множества будут сгенерированы следующими операциями?

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

УПРАЖНЕНИЕ 6.2

Напишите составной запрос, который находит имена и фамилии всех актеров и клиентов, чьи фамилии начинаются с буквы L.

УПРАЖНЕНИЕ 6.3

Отсортируйте результаты выполнения упражнения 6.2 по столбцу last_name.

Генерация, обработка и преобразование данных

Как упоминалось в предисловии, назначение этой книги — научить вас универсальным методам SQL, которые могут применяться на разных серверах баз данных. Однако в данной главе речь идет о генерации, преобразовании и обработке строковых, числовых и временных данных. Язык SQL не включает команд, обеспечивающих эту функциональность. Для облегчения генерации, преобразования и обработки данных используются встроенные функции. Хотя стандарт SQL определяет некоторые функции, поставщики баз данных не всегда следуют спецификациям этих функций.

Поэтому мой подход в данной главе состоит в том, чтобы сначала показать вам некоторые распространенные способы генерации и обработки данных в инструкциях SQL, а затем — некоторые встроенные функции, реализованные в Microsoft SQL Server, Oracle Database и MySQL. При чтении этой главы я настоятельно рекомендую вам загрузить справочное руководство по функциям, реализованным на вашем сервере. Если вы работаете более чем с одним сервером баз данных, то можете поискать книги сразу по нескольким серверам баз данных.

Работа со строковыми данными

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

CHAR

Хранит строки фиксированной длины с заполнением неиспользованных знакомест пробелами. MySQL допускает значения CHAR длиной до 255 символов; Oracle Database разрешает до 2000 символов, а SQL Server — до 8000 символов.

`varchar`

Содержит строки переменной длины. MySQL позволяет использовать в столбце `varchar` до 65 535 символов. Oracle Database (с помощью типа `varchar2`) позволяет использовать до 4000 символов, а SQL Server — до 8000 символов.

`text` (*MySQL* и *SQL Server*) или `clob` (*Oracle Database*)

Содержит очень большие строки переменной длины (обычно в этом контексте именуемые документами). MySQL имеет несколько текстовых типов (`tinytext`, `text`, `mediumtext` и `longtext`) для документов размером до 4 Гбайт. SQL Server имеет единственный тип `text` для документов размером до 2 Гбайт, а Oracle Database включает данные типа `clob`, который может содержать документы размером до 128 Тбайт. SQL Server 2005 также включает тип данных `varchar(max)` и рекомендует использовать его вместо типа `text`, который будет удален в одном из будущих выпусков.

Чтобы продемонстрировать использование этих различных типов, для некоторых из примеров в этом разделе я использую следующую таблицу:

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

В следующих двух подразделах показано, как создавать строковые данные и манипулировать ими.

Генерация строк

Самый простой способ заполнить символьный столбец — заключить строку в кавычки, как показано в следующем примере:

```
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 помните, что если длина строки превышает максимальный размер для символьного столбца (установленный максимум или максимально допустимый размер для типа данных), сервер сгенерирует исключение. Хотя это поведение по умолчанию для всех трех

серверов, можно настроить MySQL и SQL Server так, что вместо генерации исключения они будут усекать строку без каких бы то ни было сообщений. Чтобы продемонстрировать, как MySQL справляется с этой ситуацией, в приведенном далее примере инструкция update пытается заменить столбец varchar_fld, максимальная длина которого определена равной 30, строкой длиной 38 символов:

```
mysql> UPDATE string_tbl  
-> SET varchar_fld = 'This is a piece of extremely long data';  
ERROR 1406 (22001): Data too long for column 'varchar_fld' at row 1
```

Начиная с MySQL версии 6.0, поведением по умолчанию является “строгое”, что означает, что при возникновении проблем генерируются исключения, в то время как в более старых версиях сервера строка была бы усечена и было бы выдано предупреждение. Если вы предпочитаете механизм усечения строк и выдачи предупреждений вместо генерации исключений, можете выбрать режим ANSI. В следующем примере показано, как проверить, в каком режиме вы работаете, а затем — как изменить режим с помощью команды set:

```
mysql> SELECT @@session.sql_mode;  
+-----+  
| @@session.sql_mode |  
+-----+  
| STRICT_TRANS_TABLES,NO_ENGINE_SUBSTITUTION |  
+-----+  
1 row in set (0.00 sec)  
  
mysql> SET sql_mode='ansi';  
Query OK, 0 rows affected (0.08 sec)  
  
mysql> SELECT @@session.sql_mode;  
+-----+  
| @@session.sql_mode |  
+-----+  
| REAL_AS_FLOAT,PIPES_AS_CONCAT,ANSI_QUOTES,IGNORE_SPACE,  
| ONLY_FULL_GROUP_BY,ANSI |  
+-----+  
1 row in set (0.00 sec)
```

Если вы повторно выполните предыдущую инструкцию update, то обнаружите, что столбец был изменен, но при этом выдается 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)
```

Выполнив выборку столбца `vchar_fld`, вы увидите, что строка действительно была усечена:

```
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 столбец `vchar_fld` попали только первые 30 символов строки. Лучший способ избежать усечения строки (или исключений в случае Oracle Database или MySQL в строгом режиме) при работе со столбцами `varchar` — установить для верхнего предела достаточно высокое значение, которого хватит для обработки самых длинных строк, помещаемых в столбец (вспомните, что сервер выделяет для хранения такой строки только необходимое место, поэтому установка высокого верхнего предела для столбцов `varchar` не является расточительной).

Включение одинарных кавычек

Поскольку строки указываются одинарными кавычками, следует обратить особое внимание на строки, содержащие одинарные кавычки или апострофы. Например, вы не сможете вставить следующую строку, потому что сервер будет рассматривать апостроф в слове `doesn't` как конец строки:

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

Чтобы сервер игнорировал апостроф, необходимо добавить в строку управляющий символ, который заставит сервер обрабатывать апостроф, как любой другой символ строки. Все три рассматриваемых нами сервера позволяют избежать проблемы с одинарной кавычкой, добавив еще одну кавычку непосредственно перед ней, например:

```
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 могут избежать проблемы с одинарной кавычкой, добавив перед ней символ обратной косой черты:

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

Если вы извлекаете такую строку для использования на экране или в поле отчета, никакая специальная обработка не требуется:

```
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 вы извлекаете строку для добавления в файл, который будет читать другая программа, то может потребоваться включение управляющего символа как части извлеченной строки. При использовании MySQL можно использовать встроенную функцию `quote()`, которая заключает всю строку в кавычки и добавляет управляющие символы к любым одинарным кавычкам или апострофам внутри строки. Вот как выглядит строка, полученная с помощью 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 получении данных для экспорта можно использовать функцию `quote()` для всех символьных столбцов, не генерируемых системой, например для столбца `customer_notes`.

Включение специальных символов

Если ваше приложение является многонациональным, в нем, вероятно, будут встречаться строки, содержащие символы, которых нет на клавиатуре. При работе с французским и немецким языками, например, вам может потребоваться включить символы с диакритическими знаками, такие как é или ö. Серверы SQL Server и MySQL включают встроенную функцию `char()`,

позволяющую создавать строки, содержащие любые из 255 символов в наборе символов ASCII (пользователи Oracle Database могут использовать функцию `chr()`). Для демонстрации в следующем примере извлекаются введенная с клавиатуры строка и ее эквивалент, построенный с помощью 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)
```

Так, 97-й символ в наборе символов ASCII — это буква *a*. В то время как показанные в предыдущем примере символы являются обычными символами английского алфавита, в следующих примерах демонстрируются различные символы с диакритическими знаками, а также иные специальные символы, такие как символы валюты:

```
mysql> SELECT CHAR(128,129,130,131,132,133,134,135,136,137);
```

```
+-----+
| CHAR(128,129,130,131,132,133,134,135,136,137) |
+-----+
| Çüéâääåçëë |
+-----+
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) |
+-----+
| èííìÃÅæÆö |
+-----+
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) |
+-----+
| öðùùýÖÜøfØ |
+-----+
1 row in set (0.00 sec)
```

```
mysql> SELECT CHAR(158,159,160,161,162,163,164,165);
```

```
+-----+
| CHAR(158,160,161,162,163,164,165) |
+-----+
| ×áíóúññ |
+-----+
1 row in set (0.01 sec)
```



Для примеров в этом разделе я использую набор символов utf8mb4. Если ваш сеанс настроен для другого набора символов, вы увидите символы, отличные от показанных здесь. Рассматриваемые концепции останутся применимыми, но вам для использования конкретных символов нужно будет ознакомиться со схемой их размещения в используемом наборе.

Посимвольное построение строк может быть довольно утомительным, особенно если символы в строке используют диакритические знаки. К счастью, вы можете использовать функцию concat () для соединения отдельных строк, одни из которых вы можете просто вводить с клавиатуры, а другие генерировать с помощью функции char (). Например, ниже показано, как построить фразу *danke schön* с использованием 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 использовать оператор конкатенации ||:

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

В SQL Server нет функции concat (); здесь вам потребуется оператор конкатенации +:

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

Если у вас есть символ и нужно найти его эквивалент в ASCII, можете воспользоваться функцией ascii (), которая возвращает номер крайнего слева символа в переданной строке :

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

Используя функции `char()`, `ascii()` и `concat()` (или операторы конкатенации), вы сможете справиться с любой латиницей, даже если используете клавиатуру без диакритических знаков и специальных символов.

Манипуляции строками

Каждый сервер базы данных включает множество встроенных функций для манипуляции строками. В этом разделе рассматриваются два типа строковых функций: те, которые возвращают числа, и те, которые возвращают строки. Однако, прежде чем начать, я сбрасываю данные в таблице `string_tbl`, заменяя их следующими:

```
mysql> DELETE FROM string_tbl;
Query OK, 1 row affected (0.02 sec)
mysql> INSERT INTO string_tbl (char_fld, varchar_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 числовые значения

Из строковых функций, возвращающих числовые значения, одной из наиболее часто используемых является `length()`, которая возвращает количество символов в строке (пользователям SQL Server потребуется функция `len()`). Следующий запрос использует функцию `length()` для каждого столбца таблицы `string_tbl`:

```
mysql> SELECT LENGTH(char_fld) char_length,
->        LENGTH(varchar_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)
```

В то время как длины столбцов `varchar` и `text` соответствуют нашим ожиданиям, вы, возможно, ожидаете, что длина столбца `char` будет равна 30, поскольку я сказал вам, что строки в столбцах `char` хранятся с заполнением пробелами справа. Но сервер MySQL удаляет завершающие пробелы из данных типа `char` при их извлечении, так что вы увидите одни и те же

результаты для всех строковых функций независимо от типа столбца, в котором хранятся строки.

Наряду с определением длины строки можно найти местоположение подстроки внутри строки. Например, чтобы найти позицию, в которой в столбце `vchar_fld` появляется строка 'characters', можно использовать функцию `position()`, как показано в следующем примере:

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

Если подстрока не может быть найдена, функция `position()` возвращает значение 0.



Тем из вас, кто программирует на таком языке, как С или C++, в которых отсчет элементов массива начинается с позиции 0, при работе с базами данных следует помнить, что первый символ строки находится в позиции 1. Возвращаемое значение 0 указывает, что подстрока не может быть найдена, а не то, что подстрока является началом строки, в которой выполняется поиск.

Если вы хотите начать поиск с какого-то иного, отличного от первого, символа строки, используйте функцию `locate()`, которая подобна функции `position()`, но допускает необязательный третий параметр, используемый для указания начальной позиции поиска. Функция `locate()` не является стандартной, в то время как функция `position()` является частью стандарта SQL:2003. Вот пример запроса позиции строки `is`, начиная с пятого символа в столбце `vchar_fld`:

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



В Oracle Database нет функций `position()` и `locate()`, но есть функция `instr()`, которая имитирует функцию `position()` при наличии двух аргументов и функцию `locate()` — при наличии трех. SQL Server также не включает функции `position()` или `locate()`, но в нем есть функция `charindex()`, которая принимает два или три аргумента, как и функция Oracle `instr()`.

Еще одна функция, которая принимает в качестве аргументов строки и возвращает числовое значение, — это функция сравнения строк `strcmp()`. Она 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 предшествует второй в порядке сортировки;
- 0, если строки идентичны;
- 1, если первая строка в порядке сортировки идет после второй.

Чтобы проиллюстрировать, как работает эта функция, я сначала покажу порядок сортировки пяти строк, используя запрос, а затем — как строки сравниваются одна с другой, используя `strcmp()`. Вот эти пять строк, которые я вставляю 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'),
->           ('xyz'),
->           ('QRSTUV'),
->           ('qrstuv'),
->           ('12345');
```

```
Query OK, 5 rows affected (0.05 sec)
Records: 5 Duplicates: 0 Warnings: 0
```

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 выполняет шесть сравнений между пятью разными строками:

```
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)
```

Первое сравнение дает 0, чего и следовало ожидать, поскольку я сравнивал строку саму с собой. Четвертое сравнение также дает 0, что немного удивительно, поскольку, хотя строки и состоят из одних и тех же букв, одна строка состоит из прописных букв, а другая — из строчных. Причина такого результата в том, что функция MySQL `strcmp()` нечувствительна к регистру, о чем следует помнить при ее использовании. Остальные четыре сравнения дают -1 или 1, в зависимости от того, предшествует ли первая строка второй или идет после нее в порядке сортировки. Например, `strcmp('abcd', 'xyz')` дает -1, поскольку строка 'abcd' в порядке сортировки находится перед строкой 'xyz'.

Наряду с функцией `strcmp()` MySQL также позволяет использовать для сравнения строк в предложении `select` операторы `like` и `regexp`. Такие сравнения будут давать значение 1 (истина) или 0 (ложь). Таким образом, эти операторы позволяют создавать выражения, возвращающие число, действуя очень похоже на описанные в этом разделе функции, например:

```
mysql> SELECT name, name LIKE '%y' ends_in_y
->   FROM category;
+-----+-----+
| name      | ends_in_y |
+-----+-----+
| Action    |      0 |
| Animation |      0 |
| Children  |      0 |
| Classics  |      0 |
| Comedy    |      1 |
```

```
| Documentary |      1 |
| Drama       |      0 |
| Family      |      1 |
| Foreign     |      0 |
| Games       |      0 |
| Horror      |      0 |
| Music       |      0 |
| New         |      0 |
| Sci-Fi      |      0 |
| Sports      |      0 |
| Travel      |      0 |
+-----+
16 rows in set (0.00 sec)
```

В этом примере извлекаются все имена категорий вместе с выражением, возвращающим 1, если имя заканчивается буквой 'у', или 0 — в противном случае. Если требуется проверка на соответствие более сложному шаблону, можно использовать оператор regexp, как показано ниже:

```
mysql> SELECT name, name REGEXP 'y$' ends_in_y
-> FROM category;
+-----+-----+
| name      | ends_in_y |
+-----+-----+
| Action    |      0 |
| Animation |      0 |
| Children  |      0 |
| Classics  |      0 |
| Comedy    |      1 |
| Documentary |      1 |
| Drama    |      0 |
| Family   |      1 |
| Foreign  |      0 |
| Games    |      0 |
| Horror   |      0 |
| Music    |      0 |
| New      |      0 |
| Sci-Fi   |      0 |
| Sports   |      0 |
| Travel   |      0 |
+-----+
16 rows in set (0.00 sec)
```

Второй столбец этого запроса возвращает 1, если значение, хранящееся в столбце name, соответствует заданному регулярному выражению.



Пользователи Microsoft SQL Server и Oracle Database могут добиться аналогичных результатов путем построения case-выражений, которые я подробно описываю в главе 11, “Условная логика”.

Строковые функции, возвращающие строки

В некоторых случаях требуется изменять существующие строки, извлекая часть строки или добавляя к ней дополнительный текст. Каждый сервер базы данных включает несколько функций, помогающих в решении этих задач. Перед тем как начать, я еще раз сбрасываю данные в таблице `string_tbl`:

```
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 я продемонстрировал использование функции `concat()` для создания слов, содержащих символы с диакритическими знаками. Функция `concat()` полезна и во многих других ситуациях, в том числе когда нужно добавить к хранимой строке дополнительные символы. Например, в следующем примере к строке, хранящейся в столбце `text_fld`, в конце добавляется дополнительная фраза:

```
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 содержимое столбца `text_fld` выглядит следующим образом:

```
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, можно использовать `concat()` для замены данных, хранящихся в символьном столбце, как и прочие функции, возвращающие строку.

Еще одно распространенное использование функции `concat()` — построение строки из отдельных фрагментов данных. Например, следующий запрос генерирует повествовательную строку для каждого клиента:

```
mysql> SELECT concat(first_name, ' ', last_name,
-> ' has been a customer since ',
-> date(create_date)) cust_narrative
-> FROM customer;
+-----+
```

```
+-----+
| cust_narrative
+-----+
| MARY SMITH has been a customer since 2006-02-14
| PATRICIA JOHNSON has been a customer since 2006-02-14
| LINDA WILLIAMS has been a customer since 2006-02-14
| BARBARA JONES has been a customer since 2006-02-14
| ELIZABETH BROWN has been a customer since 2006-02-14
| JENNIFER DAVIS has been a customer since 2006-02-14
| MARIA MILLER has been a customer since 2006-02-14
| SUSAN WILSON has been a customer since 2006-02-14
| MARGARET MOORE has been a customer since 2006-02-14
| DOROTHY TAYLOR has been a customer since 2006-02-14
| ...
| RENE MCALISTER has been a customer since 2006-02-14
| EDUARDO HIATT has been a customer since 2006-02-14
| TERRENCE GUNDERSON has been a customer since 2006-02-14
| ENRIQUE FORSYTHE has been a customer since 2006-02-14
| FREDDIE DUGGAN has been a customer since 2006-02-14
| WADE DELVALLE has been a customer since 2006-02-14
| AUSTIN CINTRON has been a customer since 2006-02-14
+-----+
```

599 rows in set (0.00 sec)

Функция concat() может обрабатывать любое выражение, возвращающее строку, и даже преобразовывать числа и даты в строковый формат, о чем свидетельствует столбец даты (create_date), использованный в качестве аргумента. Хотя Oracle Database имеет функцию concat(), она принимает только два строковых аргумента, поэтому предыдущий запрос в Oracle работать не будет. Вместо этого нужно использовать оператор конкатенации ||, а не вызов функции:

```
SELECT first_name || ' ' || last_name ||
  ' has been a customer since ' || date(create_date)) cust_narrative
FROM customer;
```

SQL Server не включает функцию concat(), поэтому вам нужно будет использовать тот же подход, что и в предыдущем запросе, с использованием оператора конкатенации SQL Server + вместо ||.

Хотя функция concat() полезна для добавления символов в начало или конец строки, может также потребоваться добавить или заменить символы в средине строки. Все три сервера баз данных предоставляют функции для этой цели, но все они разные, поэтому я демонстрирую функцию MySQL, а затем показываю функции двух других серверов.

MySQL включает функцию insert(), которая принимает четыре аргумента: исходную строку, позицию, с которой следует начать вставку, количество

вставляемых символов и вставляемую строку. В зависимости от значения третьего аргумента функция может использоваться как для вставки, так и для замены символов в строке. При значении третьего аргумента 0 строка вставляется (любые завершающие символы сдвигаются вправо):

```
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 позиции 9, сдвигаются вправо и в строку вставляется подстрока 'cruel'. Если же третий аргумент больше нуля, то он указывает количество символов, которые заменяются вставляемой строкой:

```
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 заменяются строкой 'hello'. Oracle Database не предоставляет ни одной функции, обладающей гибкостью функции MySQL `insert()`. В Oracle Database имеется функция `replace()`, предназначенная для замены одной подстроки другой. Вот предыдущий пример, переработанный для применения функции `replace()`:

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

Все экземпляры строки 'goodbye' будут заменены строкой 'hello', в результате чего получится строка 'hello world'. Функция `replace()` заменяет *каждый* экземпляр искомой подстроки заменяемой строкой, поэтому нужно быть осторожным, чтобы не получить больше замен, чем ожидалось.

SQL Server также включает функцию `replace()`, с той же функциональностью, что и Oracle, но в SQL Server есть еще одна функция с именем `stuff()`, с функциональностью, аналогичной таковой у функции MySQL `insert()`:

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

При выполнении удаляются пять символов, начиная с позиции 1, а затем в начальную позицию вставляется строка 'Goodbye cruel', в результате чего образуется строка 'Goodbye cruel world'.

Помимо вставки символов в строку, может потребоваться извлечение подстроки из строки. Для этой цели все три сервера включают функцию `substring()` (хотя в Oracle Database она называется `substr()`), которая извлекает указанное количество 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)
```

Помимо функций, продемонстрированных здесь, все три сервера включают еще много встроенных функций для управления строковыми данными. Хотя многие из них предназначены для очень специфических целей, например для создания строкового эквивалента восьмеричного или шестнадцатеричного числа, имеется много других функций общего назначения, например для удаления или добавления конечных пробелов. Для получения дополнительной информации обратитесь к документации на вашу СУБД или к более подробно рассматривающим этот вопрос книгам, например к книге *SQL. Полное руководство*¹.

Работа с числовыми данными

В отличие от строковых (и временных, как вы вскоре увидите) данных, генерация числовых данных довольно проста. Вы можете ввести число, получить его из другого столбца или сгенерировать его путем вычислений. При выполнении вычислений доступны все обычные арифметические операторы (+, -, *, /), а для изменения приоритетов вычислений можно использовать скобки:

```
mysql> SELECT (37 * 59) / (78 - (8 * 6));
+-----+
| (37 * 59) / (78 - (8 * 6)) |
+-----+
```

¹ Джеймс Грофф, Пол Вайнберг, Эндрю Оппель. *SQL. Полное руководство*. — СПб.: ООО “Диалектика”, 2020.

```
+-----+  
|      72.77 |  
+-----+  
1 row in set (0.00 sec)
```

Как я упоминал в главе 2, “Создание и наполнение базы данных”, основная проблема при хранении числовых данных заключается в том, что числа могут быть округлены, если они больше размера, указанного для числового столбца. Например, число 9,96 будет округлено до 10,0, если оно сохранено в столбце, определенном как float(3,1).

Выполнение математических функций

Большинство встроенных числовых функций используются для определенных вычислений, таких как вычисление квадратного корня из числа. В табл. 7.1 перечислены некоторые из распространенных числовых функций, которые принимают единственный числовой аргумент и возвращают числовое значение.

Таблица 7.1. Числовые функции от одного аргумента

Имя функции	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 x
tan(x)	Вычисляет тангенс x

Эти функции выполняют очень специфические задачи, так что я воздержусь от примеров их применения (если вы не узнаёте функцию по имени или описанию, то вряд ли она вам потребуется). Однако другие числовые функции, используемые для вычислений, немного более гибкие и заслуживают некоторого объяснения.

Например, оператор вычисления остатка от деления одного числа на другое реализован в MySQL и Oracle Database в виде функции 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)
```

В то время как функция `mod()` обычно работает с целочисленными аргументами, в MySQL можно также использовать действительные числа, например:

```
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 используется оператор `%`. Таким образом, выражение $10\%4$ дает значение 2.

Еще одна числовая функция, которая принимает два числовых аргумента, — это функция `pow()` (или `power()`, если вы используете Oracle Database или SQL Server), которая возвращает одно число, возведенное в степень, равную значению второго числа, например:

```
mysql> SELECT POW(2,8);
+-----+
| POW(2,8) |
+-----+
|      256  |
+-----+
1 row in set (0.03 sec)
```

Таким образом, `pow(2, 8)` является MySQL-эквивалентом для 2^8 . Поскольку память компьютера выделяется блоками по 2^x байт, функция `pow()` может быть удобным средством определения точного количества байтов в определенном объеме памяти:

```
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 легче запомнить, что гигабайт — это 2^{30} байт, чем запомнить число 1073 741 824.

Управление точностью чисел

При работе с числами с плавающей точкой не всегда требуется работа или отображение числа с полной точностью. Например, можно хранить денежные данные транзакции с точностью до шести знаков после запятой, но для их отображения понадобится округление до ближайшей сотой. При ограничении точности чисел с плавающей точкой используются следующие четыре функции: `ceil()`, `floor()`, `round()` и `truncate()`. Они предоставляют все три сервера; правда, Oracle Database использует `trunc()` вместо `truncate()`, а SQL Server — `ceiling()` вместо `ceil()`.

Функции `ceil()` и `floor()` используются для округления в большую или меньшую сторону к ближайшему целому числу, 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)
```

Таким образом, для любого числа между 72 и 73 функция `ceil()` вернет значение 73, а функция `floor()` — 72. Помните, что `ceil()` будет округляться в большую сторону, даже если десятичная часть числа очень мала, а `floor()` — в меньшую, даже если десятичная часть очень 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 для вашего приложения это важно, можете использовать функцию `round()` для округления к ближайшему целому:

```
mysql> SELECT ROUND(72.49999), ROUND(72.5), ROUND(72.50001);
+-----+-----+-----+
| ROUND(72.49999) | ROUND(72.5) | ROUND(72.50001) |
+-----+-----+-----+
```

```
+-----+-----+-----+
|      72 |      73 |      73 |
+-----+-----+-----+
1 row in set (0.00 sec)
```

При использовании `round()` любое число, десятичная часть которого равна 0,5 или больше, будет округлено в большую сторону, тогда как, если она меньше 0,5, то округление будет выполнено в меньшую сторону.

В большинстве случаев требуется сохранять хотя бы некоторую часть десятичных знаков после запятой, а не округлять значение до целого числа. Для этого функция `round()` допускает необязательный второй аргумент, который указывает, какое количество цифр справа от запятой следует оставить при округлении. В следующем примере показано, как можно использовать второй аргумент для округления значения 72,0909 с точностью до одного, двух и трех десятичных знаков:

```
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, указывающий количество цифр справа от десятичной запятой, но `truncate()` просто отбрасывает ненужные цифры без округления. В следующем примере показано, как число 72,0909 усекается 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 функции `truncate()` нет. Вместо этого функция `round()` имеет необязательный третий аргумент, при наличии которого, не равного нулю, число усекается, а не округляется.

И `truncate()`, и `round()` допускают *отрицательное* значение второго аргумента. Это означает, что усекаются или округляются цифры слева от

десятичной точки. Это поначалу может показаться странным, но для таких значений аргумента имеются вполне допустимые приложения. Например, вы можете продавать продукт, который можно покупать только по 10 единиц. Если покупателем заказаны 17 единиц, можете выбрать один из следующих способов изменения количества товара в заказе клиента:

```
mysql> SELECT ROUND(17, -1), TRUNCATE(17, -1);
+-----+-----+
| ROUND(17, -1) | TRUNCATE(17, -1) |
+-----+-----+
|      20 |          10 |
+-----+-----+
1 row in set (0.00 sec)
```

Если речь идет о канцелярских кнопках, то прибыль не сильно будет зависеть от того, сколько канцелярских кнопок вы продадите покупателю, — 10 или 20, если он запросил их 17. Но если вы продаете часы Rolex, ваш бизнес при неверном выборе может пострадать.

Работа со знаковыми данными

Если вы работаете с числовыми столбцами, в которых допускаются отрицательные значения (в главе 2, “Создание и наполнение базы данных”, я показывал, как указать, что числовой столбец может содержать только неотрицательные значения), вам могут быть полезны некоторые специализированные числовые функции. Пусть, например, вас просят создать отчет, показывающий текущее состояние набора банковских счетов, используя следующие данные из таблицы account:

```
+-----+-----+-----+
| account_id | acct_type   | balance |
+-----+-----+-----+
|      123 | MONEY MARKET |  785.22 |
|      456 | SAVINGS     |    0.00 |
|      789 | CHECKING    | -324.22 |
+-----+-----+-----+
```

Показанный далее запрос возвращает три столбца, полезных при создании отчета:

```
mysql> SELECT account_id, SIGN(balance), ABS(balance)
-> FROM account;
+-----+-----+-----+
| account_id | SIGN(balance) | ABS(balance) |
+-----+-----+-----+
|      123 |           1 |      785.22 |
|      456 |           0 |       0.00 |
+-----+-----+-----+
```

```
|      789 |          -1 |      324.22 |
+-----+-----+
3 rows in set (0.00 sec)
```

Во втором столбце функция `sign()` возвращает значение `-1`, если баланс счета отрицательный, `0`, если баланс нулевой, и `1`, если баланс положительный. Третий столбец получает абсолютное значение баланса счета с помощью функции `abs()`.

Работа с временными данными

Из трех типов данных, обсуждаемых в этой главе (символьные, числовые и временные), когда дело доходит до создания и обработки данных, временные данные являются наиболее сложными. Некоторая сложность временных данных вызвана множеством способов, которыми можно описать единственную дату и время. Например, дату, когда я пишу этот абзац, можно описать следующими способами (и это далеко не полный список).

- Wednesday, June 5, 2019
- 6/05/2019 2:14:56 P.M. EST
- 6/05/2019 19:14:56 GMT
- 1562019 (Юлианская дата)
- Звездная дата [-4] 97026.79 14:14:56 (формат *Звездного пути*)

Хотя некоторые из этих различий связаны исключительно с форматированием, большая часть сложности связана с системой отсчета, которую мы исследуем в следующем разделе.

Часовые пояса

Поскольку люди во всем мире предпочитают, чтобы полдень примерно соответствовал максимальному подъему Солнца в их местности, никогда не было никаких серьезных попыток заставить всех в мире использовать некоторое универсальное время. Вместо этого мир был разделен на 24 воображаемые полосы, именуемые *часовыми поясами*; в пределах определенного часового пояса используют одно и то же текущее время, которое в разных часовых поясах различается. В то время как это выглядит достаточно просто, надо учесть, что некоторые страны сдвигают свое время на один час два раза в год (реализуя так называемое *летнее время*), а некоторые нет, поэтому разница во времени между двумя точками на Земле может составлять в первую

половину года четыре часа, а во вторую — пять. Даже в пределах одного часового пояса разные регионы могут придерживаться, или не придерживаться, летнего времени, в результате чего разные часы в одном часовом поясе согласованы в течение первой половины года, но отличаются на один час во время второй половины.

Хотя компьютерный век усугубил эту проблему, фактически люди сталкиваются с различиями в часовых поясах с первых дней морских путешествий. Чтобы обеспечить общую точку отсчета для хронометражка, мореплаватели еще в XV веке установили свои часы на время дня в Гринвиче, Англия. Это время стало известно как *среднее время по Гринвичу* (Greenwich Mean Time, или GMT). Все остальные часовые пояса можно описать количеством часов сдвига от GMT; например, часовой пояс для восточной части США, известный как восточное стандартное время, может быть описан как GMT -5:00 или на пять часов раньше, чем GMT.

Сегодня мы используем вариант GMT, именуемый *Всемирным координированным временем* (Coordinated Universal Time — UTC), который основан на атомных часах (или, если быть более точным, на среднем времени 200 атомных часов в 50 местах по всему миру, которое известно как *всемирное время* (Universal Time)). И SQL Server, и MySQL предоставляют функции, которые будут возвращать текущую метку времени UTC (`getutcdate()` — для SQL Server и `utc_timestamp()` — для MySQL).

Большинство серверов баз данных по умолчанию используют настройки часового пояса, в котором находится сервер, и предоставляют инструменты для изменения часового пояса, если это необходимо. Например, база данных, используемая для хранения биржевых транзакций со всего мира, обычно настраивается на использование времени UTC, тогда как база данных, используемая для хранения транзакций в конкретном розничном заведении, может использовать часовой пояс сервера.

MySQL хранит два разных часовых пояса: глобальный часовой пояс и часовой пояс сеанса, которые могут быть разными для каждого пользователя, вошедшего в базу данных. Вы можете увидеть обе настройки с помощью следующего запроса:

```
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` говорит о том, что сервер использует настройку часового пояса, в котором находится база данных.

Если вы сидите за компьютером в Цюрихе (Швейцария) и открываете сеанс по сети на сервере MySQL, расположенном в Нью-Йорке, то можете изменить настройку часового пояса для своего сеанса с помощью следующей команды:

```
mysql> SET time_zone = 'Europe/Zurich';
Query OK, 0 rows affected (0.18 sec)
```

Если вы еще раз проверите настройки часового пояса, то увидите 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 вашем сеансе, теперь будут соответствовать времени Цюриха.



Пользователи Oracle Database могут изменить настройку часового пояса для сеанса с помощью следующей команды:

```
ALTER SESSION TIMEZONE = 'Europe/Zurich'
```

Генерация временных данных

Вы можете генерировать временные данные любым из следующих способов:

- копирование данных из существующего столбца `date`, `datetime` или `time`;
- выполнение встроенной функции, которая возвращает `date`, `datetime` или `time`;
- построение строкового представления временных данных для вычисления сервером.

Чтобы использовать последний метод, вам необходимо знать о различных компонентах, используемых при форматировании дат.

Строковые представления временных данных

В табл. 2.5 главы 2, “Создание и наполнение базы данных”, представлены наиболее популярные компоненты дат; чтобы освежить вашу память, в табл. 7.2 показаны эти же компоненты.

Таблица 7.2. Компоненты формата даты

Компонент	Определение	Диапазон
YYYY	Год, включая век	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	Минуты	От 00 до 59
SS	Секунды	От 00 до 59

Чтобы создать строку, которую сервер может интерпретировать как date, datetime или time, необходимо собрать вместе различные компоненты в порядке, указанном в табл. 7.3.

Таблица 7.3. Компоненты дат и времени

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:SSS

Таким образом, чтобы заполнить столбец datetime значением 15:30 17 сентября 2019 года, нужно построить следующую строку:

```
'2019-09-17 15:30:00'
```

Если сервер ожидает значение datetime, например при обновлении столбца datetime или при вызове встроенной функции, которая принимает аргумент datetime, можно предоставить корректно отформатированную строку с необходимыми компонентами даты, и сервер выполнит преобразование вместо вас. Например, вот инструкция, используемая для изменения даты возврата взятого напрокат фильма:

```
UPDATE rental
SET return_date = '2019-09-17 15:30:00'
WHERE rental_id = 99999;
```

Сервер определяет, что строка, указанная в предложении `set`, должна быть значением `datetime`, так как строка используется для заполнения столбца `datetime`. Поэтому сервер попытается преобразовать строку, разделяя ее на шесть компонентов (год, месяц, день, час, минута, секунда), включаемых в формат `datetime` по умолчанию.

Преобразование строки в дату

Если сервер *не* ожидает значения `datetime` или если вы хотите представить `datetime` в формате, отличном от формата по умолчанию, вам нужно указать серверу на необходимость преобразования строки в дату и время. Например, вот простой запрос, который возвращает значение `datetime` с помощью функции `cast()`:

```
mysql> SELECT CAST('2019-09-17 15:30:00' AS DATETIME);
+-----+
| CAST('2019-09-17 15:30:00' AS DATETIME) |
+-----+
| 2019-09-17 15:30:00                     |
+-----+
1 row in set (0.00 sec)
```

Мы рассмотрим функцию `cast()` в конце этой главы. Хотя этот пример демонстрирует, как создавать значения `datetime`, та же логика применяется и к типам `date` и `time`. В следующем запросе функция `cast()` используется для генерации значений `date` и `time`:

```
mysql> SELECT CAST('2019-09-17' AS DATE) date_field,
   -> CAST('108:17:57' AS TIME) time_field;
+-----+-----+
| date_field | time_field |
+-----+-----+
| 2019-09-17 | 108:17:57 |
+-----+-----+
1 row in set (0.00 sec)
```

Конечно, вместо того, чтобы позволить серверу выполнять неявные преобразования, можно выполнять их явно, даже если сервер ожидает значение `date`, `datetime` или `time`.

Когда строки преобразуются во временные значения — явно или неявно — вы должны предоставить все компоненты даты в необходимом порядке. Хотя некоторые серверы довольно строги в отношении формата даты, сервер MySQL весьма снисходительно относится к разделителям, находящимся между компонентами. Например, MySQL примет все следующие строки как корректное представление даты 15:30 17 сентября 2019 года:

```
'2019-09-17 15:30:00'  
'2019/09/17 15:30:00'  
'2019,09,17,15,30,00'  
'20190917153000'
```

Хотя это дает вам немного большую гибкость, вы можете попытаться создать временное значение без компонентов даты по умолчанию; в следующем разделе показана встроенная функция, которая намного более гибкая, чем функция `cast()`.

Функции генерации дат

Если вам нужно сгенерировать временные данные из строки, а строка находится в неверном виде для использования функции `cast()`, можно использовать встроенную функцию, которая принимает вместе со строкой даты строку формата. В MySQL эта функция имеет имя `str_to_date()`. Пусть, например, вы извлекаете из файла строку 'September 17, 2019' и вам нужно использовать ее для обновления столбца `date`. Поскольку строка не соответствует требуемому формату `YYYY-MM-DD`, можете использовать `str_to_date()` вместо того, чтобы переформатировать строку для использования функции `cast()`:

```
UPDATE rental  
SET return_date = STR_TO_DATE('September 17, 2019', '%M %d, %Y')  
WHERE rental_id = 99999;
```

Второй аргумент в вызове `str_to_date()` определяет формат строки даты. В данном случае строка содержит название месяца (%M), числовое значение дня (%d) и четырехзначное числовое значение года (%Y). Хотя имеется более 30 распознаваемых компонентов формата, в табл. 7.4 показаны полтора десятка наиболее часто используемых из них.

Таблица 7.4. Компоненты формата даты

Компонент формата	Описание
%M	Полное имя месяца (January..December)
%m	Числовое значение месяца
%d	Числовое значение дня месяца (00..31)
%j	День года (001..366)
%W	Полное имя дня недели (Sunday..Saturday)
%Y	Значение года (четыре цифры)
%y	Значение года (две цифры)
%H	Час дня в 24-часовом формате (00..23)
%h	Час дня в 12-часовом формате (01..12)

Компонент формата	Описание
%i	Минуты в часе (00..59)
%s	Число секунд (00..59)
%f	Число микросекунд (000000..999999)
%p	AM или PM
%a	Краткое имя дня недели — Sun, Mon, ...
%b	Краткое имя месяца — Jan, Feb, ...

Функция `str_to_date()` возвращает значение `datetime`, `date` или `time` в зависимости от содержимого строки формата. Например, если строка формата включает только `%H`, `%i` и `%s`, будет возвращено значение `time`.



Пользователи Oracle Database могут использовать функцию `to_date()` так же, как функцию MySQL `str_to_date()`. SQL Server включает функцию `convert()`, которая не такая гибкая, как в MySQL и Oracle Database: в ней вместо указания настраиваемой строки формата строка даты должна соответствовать одному из 21 предопределенного формата.

Если вы пытаетесь сгенерировать *текущую* дату/время, вам не нужно создавать строку, потому что имеются следующие встроенные функции, которые обращаются к системным часам и возвращают текущую дату и/или время в виде строки:

```
mysql> SELECT CURRENT_DATE(), CURRENT_TIME(), CURRENT_TIMESTAMP();
+-----+-----+-----+
| CURRENT_DATE() | CURRENT_TIME() | CURRENT_TIMESTAMP() |
+-----+-----+-----+
| 2019-06-05     | 16:54:36      | 2019-06-05 16:54:36 |
+-----+-----+-----+
1 row in set (0.12 sec)
```

Значения, возвращаемые этими функциями, имеют формат по умолчанию для возвращаемого временного типа. Oracle Database включает функции `current_date()` и `current_timestamp()`, но не `current_time()`, а Microsoft SQL Server включает одну только функцию `current_timestamp()`.

Манипуляции временными данными

В этом разделе исследуются встроенные функции, которые принимают аргументы даты и возвращают даты, строки или числа.

Функции, возвращающие даты

Многие встроенные временные функции принимают в качестве аргумента одну дату и возвращают другую. Например, функция MySQL `date_add()` позволяет добавлять любые интервалы (например, дни, месяцы, годы) к указанной дате для создания другой даты. 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) |
+-----+
| 2019-06-10                                |
+-----+
1 row in set (0.06 sec)
```

Второй аргумент состоит из трех элементов: ключевого слова `interval`, желаемого количества и типа интервала. В табл. 7.5 показаны некоторые из наиболее распространенных используемых типов интервалов.

Таблица 7.5. Распространенные типы интервалов

Имя интервала	Описание
second	Количество секунд
minute	Количество минут
hour	Количество часов
day	Количество дней
month	Количество месяцев
year	Количество лет
minute_second	Количество минут и секунд, разделенных ":"
hour_second	Количество часов, минут и секунд, разделенных ":"
year_month	Количество лет и месяцев, разделенных "-"

Хотя первые шесть типов, перечисленных в табл. 7.5, довольно просты, последние три типа требуют пояснения, поскольку имеют по несколько элементов. Например, если вам сказали, что фильм вернули на 3 часа 27 минут 11 секунд позже, чем было указано изначально, вы можете исправить это следующим образом:

```
UPDATE rental
SET return_date = DATE_ADD(return_date, INTERVAL '3:27:11' HOUR_SECOND)
WHERE rental_id = 99999;
```

В этом примере функция `date_add()` принимает значение из столбца `return_date` и добавляет к нему 3 часа 27 минут и 11 секунд. Затем это значение используется для изменения значения столбца `return_date`.

Или, например, если вы работаете в отделе кадров и выяснили, что сотрудник с идентификатором 4789 в базе данных старше, чем на самом деле, можете, например, добавить к дате его рождения 9 лет и 11 месяцев:

```
UPDATE employee  
SET birth_date = DATE_ADD(birth_date, INTERVAL '9-11' YEAR_MONTH)  
WHERE emp_id = 4789;
```



Пользователи SQL Server могут выполнить предыдущий пример, используя функцию `dateadd()`:

```
UPDATE employee  
SET birth_date =  
    DATEADD(MONTH, 119, birth_date)  
WHERE emp_id = 4789
```

В SQL Server нет комбинированных интервалов (например, `year_month`), поэтому я преобразовал 9 лет и 11 месяцев в 119 месяцев.

Пользователи Oracle Database могут использовать для этого функцию `add_months()`:

```
UPDATE employee  
SET birth_date = ADD_MONTHS(birth_date, 119)  
WHERE emp_id = 4789;
```

Предположим, вы хотите добавить интервал к дате и знаете, какая конечная дата вам нужна, но не знаете, сколько дней надо добавить. Например, клиент банка входит в систему онлайн-банкинга и планирует перевод на конец месяца. Вместо того чтобы писать код, который вычисляет текущий месяц, а затем ищет количество дней в этом месяце, можно вызвать функцию `last_day()`, которая сама выполнит всю работу (как MySQL, так и Oracle Database включают функцию `last_day()`; SQL Server подобной функции не имеет). Если клиент запрашивает перевод 17 сентября 2019 года, можно узнать последний день сентября следующим образом:

```
mysql> SELECT LAST_DAY('2019-09-17');  
+-----+  
| LAST_DAY('2019-09-17') |  
+-----+  
| 2019-09-30           |  
+-----+  
1 row in set (0.10 sec)
```

Независимо от того, указываете ли вы значение date или datetime, функция last_day() всегда возвращает date. Хотя применение этой функции может показаться не слишком большой экономией времени, лежащая в ее основе логика может оказаться сложной, если, например, вы пытаетесь найти последний день февраля и необходимо выяснить, является ли текущий год високосным.

Функции, возвращающие строки

Большинство временных функций, возвращающих строковые значения, используются для извлечения части даты или времени. Например, в MySQL имеется функция dayname() для определения того, на какой день недели выпадает определенная дата, например:

```
mysql> SELECT DAYNAME('2019-09-18');
+-----+
| DAYNAME('2019-09-18') |
+-----+
| Wednesday           |
+-----+
1 row in set (0.00 sec)
```

В MySQL включены многие такие функции для извлечения информации из значений даты, но я рекомендую вместо них использовать функцию extract(), так как проще запомнить несколько вариантов одной функции, чем десяток различных функций. Кроме того, функция extract() является частью стандарта SQL:2003 и реализована и в Oracle Database, и в MySQL.

Функция extract() использует те же типы интервалов, что и функция date_add() (см. табл. 7.5), чтобы определить, какой элемент даты вас интересует. Например, чтобы извлечь из значения datetime только часть года, можно сделать следующее:

```
mysql> SELECT EXTRACT(YEAR FROM '2019-09-18 22:19:05');
+-----+
| EXTRACT(YEAR FROM '2019-09-18 22:19:05') |
+-----+
|                               2019 |
+-----+
1 row in set (0.00 sec)
```



SQL Server не реализует extract(), но включает функцию datepart(). Вот как извлекается год из значения datetime с помощью этой функции:

```
SELECT DATEPART(YEAR, GETDATE())
```

Функции, возвращающие числовые значения

Ранее в этой главе была показана функция, которая используется для добавления заданного интервала к значению даты и, таким образом, генерирует другое значение даты. Еще одна распространенная задача при работе с датами — определение количества интервалов (дней, недель, лет) между двумя датами. Для этого MySQL включает функцию `datediff()`, которая возвращает количество полных дней между двумя датами. Например, чтобы узнать, сколько дней мои дети не будут ходить в школу этим летом, можно делать следующее:

```
mysql> SELECT DATEDIFF('2019-09-03', '2019-06-21');
+-----+
| DATEDIFF('2019-09-03', '2019-06-21') |
+-----+
|                               74 |
+-----+
1 row in set (0.00 sec)
```

Итак, мне предстоит пережить 74 дня ядовитого плюща, укусов комаров и царапин на коленях, прежде чем дети благополучно вернутся в школу. Функция `datediff()` игнорирует время дня в своих аргументах. Даже если я включу время дня, установив его для первой даты за секунду до полуночи, а для второй — через секунду после полуночи, это никак не повлияет на вычисления:

```
mysql> SELECT DATEDIFF('2019-09-03 23:59:59', '2019-06-21 00:00:01');
+-----+
| DATEDIFF('2019-09-03 23:59:59', '2019-06-21 00:00:01') |
+-----+
|                               74 |
+-----+
1 row in set (0.00 sec)
```

Если я поменяю местами аргументы и сначала укажу более раннюю дату, `dateiff()` вернет отрицательное значение:

```
mysql> SELECT DATEDIFF('2019-06-21', '2019-09-03');
+-----+
| DATEDIFF('2019-06-21', '2019-09-03') |
+-----+
| -74 |
+-----+
1 row in set (0.00 sec)
```



SQL Server также включает функцию `dateiff()`, но ее реализация более гибкая, чем в MySQL, — в ней вы можете указать тип интервала (т.е. год, месяц, день, час) вместо подсчета только количества дней между двумя датами. Вот как предыдущий пример выглядит в SQL Server:

```
SELECT DATEDIFF(DAY, '2019-06-21', '2019-09-03')
```

Oracle Database позволяет определять количество дней между двумя датами, просто вычитая одну дату из другой.

Функции преобразования

Ранее в этой главе было показано, как использовать функцию `cast()` для преобразования строки в значение `datetime`. Хотя каждый сервер базы данных включает в себя ряд собственных, присущих только ему, функций для преобразования данных из одного типа в другой, я рекомендую использовать функцию `cast()`, которая включена в стандарт SQL:2003 и реализована в серверах MySQL, Oracle Database и Microsoft SQL Server.

Чтобы использовать `cast()`, вы предоставляете значение или выражение, ключевое слово `as` и тип, в который вы хотите преобразовать это значение. Вот пример преобразования строки в целое число:

```
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()` пытается преобразовать всю строку слева направо; если в строке обнаруживается знак, которого не может быть в числе, преобразование останавливается без сообщения об ошибке. Рассмотрим следующий пример:

```
mysql> SELECT CAST('999ABC111' AS UNSIGNED INTEGER);
+-----+
| CAST('999ABC111' AS UNSIGNED INTEGER) |
+-----+
|           999 |
+-----+
1 row in set, 1 warning (0.08 sec)
```

```
mysql> show warnings;
+-----+-----+
| Level | Code | Message |
+-----+-----+
| Warning | 1292 | Truncated incorrect INTEGER value: '999ABC111' |
+-----+
1 row in set (0.07 sec)
```

В этом случае преобразуются первые три цифры строки, в то время как остальные отбрасываются, что приводит к значению 999. Однако сервер выдает предупреждение, чтобы вы знали, что не вся строка была преобразована в число.

Если вы конвертируете строку в значение date, datetime или time, следует придерживаться форматов по умолчанию для каждого типа, поскольку предоставить функции cast() строку формата нельзя. Если ваша строка даты представлена не в формате по умолчанию (т.е. YYYY-MM-DD HH:MI:SS для типа datetime), то нужно прибегнуть к другой функции, такой как функция MySQL str_to_date(), описанная ранее в данной главе.

Проверьте свои знания

Предлагаемые здесь упражнения призваны закрепить понимание вами работы со встроенными функциями, описанными в данной главе. Ответы к упражнениям представлены в приложении Б.

УПРАЖНЕНИЕ 7.1

Напишите запрос, который возвращает символы строки 'Please find the substring in this string' с 17-го по 25-й.

УПРАЖНЕНИЕ 7.2

Напишите запрос, который возвращает абсолютное значение и знак (-1, 0 или 1) числа -25,76823. Верните также число, округленное до ближайших двух знаков после запятой.

УПРАЖНЕНИЕ 7.3

Напишите запрос, возвращающий для текущей даты только часть, соответствующую месяцу.

Группировка и агрегация

Данные обычно хранятся на самом низком уровне детализации, необходимом пользователям любой базы данных; если Вере из бухгалтерии нужно просмотреть отдельные транзакции клиентов, в базе данных должна быть таблица, в которой хранятся отдельные транзакции. Это, однако, не означает, что все пользователи должны иметь дело с данными в том виде, в котором они хранятся в базе данных. В этой главе основное внимание уделяется тому, как данные могут быть сгруппированы и агрегированы, чтобы позволить пользователям взаимодействовать с ними на более высоком уровне детализации, чем используемый для хранения в базе данных.

Концепции группировки

Иногда может понадобиться обнаружить определенные тенденции в данных. Для этого потребуется небольшая обработка данных сервером, чтобы вы могли сгенерировать искомые результаты. Например, предположим, что вы отвечаете за отправку купонов на бесплатную аренду по адресам ваших лучших клиентов. Вы можете выполнить простой запрос, чтобы просмотреть необработанные данные:

```
mysql> SELECT customer_id FROM rental;
+-----+
| customer_id |
+-----+
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|           1 |
|           ... |
|           599 |
|           599 |
+-----+
```

```
|      599 |
|      599 |
|      599 |
|      599 |
+-----+
16044 rows in set (0.01 sec)
```

При наличии 599 клиентов, имеющих более 16 000 записей об аренде, просмотрев необработанные данные, невозможно определить, какие клиенты взяли напрокат больше всего фильмов. Вместо этого вы можете попросить сервер базы данных сгруппировать данные с помощью предложения `group by`. Вот тот же запрос, но с использованием предложения `group by` для группировки данных о прокате по идентификатору клиента:

```
mysql> SELECT customer_id
-> FROM rental
-> GROUP BY customer_id;
+-----+
| customer_id |
+-----+
|      1 |
|      2 |
|      3 |
|      4 |
|      5 |
|      6 |
| ...      |
|     594 |
|     595 |
|     596 |
|     597 |
|     598 |
|     599 |
+-----+
599 rows in set (0.00 sec)
```

Результирующий набор содержит по одной строке для каждого отдельного значения в столбце `customer_id`, в результате получается 599 строк вместо общего количества в 16044 строк. Причина меньшего результирующего набора в том, что некоторые заказчики брали напрокат более одного фильма. Чтобы увидеть, сколько фильмов было взято напрокат каждым клиентом, можно использовать *агрегатную функцию* в предложении `select` для подсчета количества строк в каждой группе:

```
mysql> SELECT customer_id, count(*)
-> FROM rental
-> GROUP BY customer_id;
+-----+-----+
```

```

| customer_id | count(*) |
+-----+-----+
|      1 |     32 |
|      2 |     27 |
|      3 |     26 |
|      4 |     22 |
|      5 |     38 |
|      6 |     28 |
| ...   |
| 594 |     27 |
| 595 |     30 |
| 596 |     28 |
| 597 |     25 |
| 598 |     22 |
| 599 |     19 |
+-----+-----+
599 rows in set (0.01 sec)

```

Агрегатная функция `count()` подсчитывает количество строк в каждой группе, а звездочка сообщает серверу, что нужно считать все, что есть в группе. Используя комбинацию `group by` и агрегатную функцию `count()`, вы можете генерировать точные данные, необходимые для ответа на имеющийся бизнес-вопрос.

Глядя на результаты, можно увидеть, что 32 фильма были взяты напрокат клиентом с идентификатором 1, а 25 фильмов — клиентом с идентификатором 597. Чтобы определить, какие клиенты взяли напрокат больше всего фильмов, просто добавим предложение `order by`:

```

mysql> SELECT customer_id, count(*)
    -> FROM rental
    -> GROUP BY customer_id
    -> ORDER BY 2 DESC;
+-----+-----+
| customer_id | count(*) |
+-----+-----+
|      148 |     46 |
|      526 |     45 |
|      236 |     42 |
|      144 |     42 |
|       75 |     41 |
| ...   |
|     248 |     15 |
|     110 |     14 |
|     281 |     14 |
|      61 |     14 |
|     318 |     12 |
+-----+-----+
599 rows in set (0.01 sec)

```

Теперь, когда результаты отсортированы, легко увидеть, что клиент с идентификатором 148 брал напрокат наибольшее количество фильмов (46), в то время как клиент с идентификатором 318 брал наименьшее количество фильмов (12).

При группировке данных вам может потребоваться отфильтровать нежелательные данные из набора результатов на основе групп данных, а не небоработанных данных. Поскольку предложение `group by` выполняется после того, как вычислено предложение `where`, добавить условия фильтрации к предложению `where` для этой цели нельзя. Например, вот к чему приводит попытка отфильтровать клиентов, бравших напрокат менее 40 фильмов:

```
mysql> SELECT customer_id, count(*)
-> FROM rental
-> WHERE count(*) >= 40
-> GROUP BY customer_id;
ERROR 1111 (HY000): Invalid use of group function1
```

Вы не можете обратиться к агрегатной функции `count(*)` в предложении `where`, потому что во время вычисления предложения `where` группы еще не были сгенерированы. Вместо этого вы должны поместить условия группового фильтра в предложение `having`. Вот как выглядит правильный запрос:

```
mysql> SELECT customer_id, count(*)
-> FROM rental
-> GROUP BY customer_id
-> HAVING count(*) >= 40;
+-----+-----+
| customer_id | count(*) |
+-----+-----+
| 75          | 41      |
| 144         | 42      |
| 148         | 46      |
| 197         | 40      |
| 236         | 42      |
| 469         | 40      |
| 526         | 45      |
+-----+-----+
7 rows in set (0.01 sec)
```

Поскольку группы, содержащие менее 40 членов, отфильтрованы с помощью предложения `having`, результирующий набор теперь содержит только тех клиентов, которые брали напрокат 40 и более фильмов.

¹ Неверное применение групповой функции.

Агрегатные функции

Агрегатные функции выполняют определенные операции над всеми строками в группе. Несмотря на то что каждый сервер базы данных имеет собственный набор специализированных агрегатных функций, имеются распространенные агрегатные функции, реализованные всеми основными серверами, в частности следующие.

`max()`

Возвращает максимальное значение в наборе.

`min()`

Возвращает минимальное значение в наборе.

`avg()`

Возвращает усредненное значение набора.

`sum()`

Возвращает сумму значений набора.

`count()`

Возвращает количество значений в наборе.

Вот как выглядит запрос, в котором используются все распространенные агрегатные функции для анализа данных по прокату фильмов:

```
mysql> SELECT MAX(amount) max_amt,
   ->   MIN(amount) min_amt,
   ->   AVG(amount) avg_amt,
   ->   SUM(amount) tot_amt,
   ->   COUNT(*) num_payments
   -> FROM payment;
+-----+-----+-----+-----+
| max_amt | min_amt | avg_amt | tot_amt | num_payments |
+-----+-----+-----+-----+
|    11.99 |      0.00 |  4.200667 | 67416.51 |          16049 |
+-----+-----+-----+-----+
1 row in set (0.09 sec)
```

Результаты этого запроса говорят о том, что из 16 049 строк в таблице платежей максимальная выплаченная за прокат фильма сумма составила 11,99 доллара, минимальная — 0 долларов, средний платеж равен 4,20 доллара, а общая сумма всех платежей — 67 416,51 доллара. Надеюсь, этот пример даст вам представление о роли агрегатных функций. В следующих подразделах более подробно разъясняется, как эти функции можно использовать.

Неявная и явная группировка

В предыдущем примере каждое значение, возвращаемое запросом, генерируется агрегатной функцией. Поскольку предложения `group by` в запросе нет, существует единственная неявная группа (все строки в таблице `payment`).

Однако в большинстве случаев требуется получить дополнительные столбцы вместе со столбцами, генерируемыми агрегатными функциями. Например, можно расширить предыдущий запрос и выполнить те же пять агрегатных функций, но не для всех клиентов одновременно, а для каждого клиента. Для такого запроса нужно вместе с пятью агрегатными функциями выполнить выборку `customer_id`:

```
SELECT customer_id,
       MAX(amount) max_amt,
       MIN(amount) min_amt,
       AVG(amount) avg_amt,
       SUM(amount) tot_amt,
       COUNT(*) num_payments
  FROM payment;
```

Однако, если вы попытаетесь выполнить такой запрос, то получите сообщение об ошибке:

```
ERROR 1140 (42000): In aggregated query without GROUP BY,
expression #1 of SELECT list contains nonaggregated column2
```

Хотя для вас может быть очевидно, что вы хотите применения агрегатных функций к каждому найденному в таблице `payment` клиенту, этот запрос не выполняется, потому что в нем не указано явно, как должны быть сгруппированы данные. Следовательно, в запрос нужно добавить предложение `group by`, чтобы указать, к какой группе строк следует применять агрегатные функции:

```
mysql> SELECT customer_id,
   ->    MAX(amount) max_amt,
   ->    MIN(amount) min_amt,
   ->    AVG(amount) avg_amt,
   ->    SUM(amount) tot_amt,
   ->    COUNT(*) num_payments
   ->   FROM payment
   ->  GROUP BY customer_id;
+-----+-----+-----+-----+-----+
|customer_id |max_amt |min_amt | avg_amt |tot_amt |num_payments |
+-----+-----+-----+-----+-----+
```

² В агрегированном запросе без `GROUP BY` выражение №1 списка `SELECT` содержит неагрегированный столбец.

```

| 1 | 9.99 | 0.99 | 3.708750 | 118.68 | 32 |
| 2 | 10.99 | 0.99 | 4.767778 | 128.73 | 27 |
| 3 | 10.99 | 0.99 | 5.220769 | 135.74 | 26 |
| 4 | 8.99 | 0.99 | 3.717273 | 81.78 | 22 |
| 5 | 9.99 | 0.99 | 3.805789 | 144.62 | 38 |
| 6 | 7.99 | 0.99 | 3.347143 | 93.72 | 28 |
| ... |
| 594 | 8.99 | 0.99 | 4.841852 | 130.73 | 27 |
| 595 | 10.99 | 0.99 | 3.923333 | 117.70 | 30 |
| 596 | 6.99 | 0.99 | 3.454286 | 96.72 | 28 |
| 597 | 8.99 | 0.99 | 3.990000 | 99.75 | 25 |
| 598 | 7.99 | 0.99 | 3.808182 | 83.78 | 22 |
| 599 | 9.99 | 0.99 | 4.411053 | 83.81 | 19 |
+-----+

```

599 rows in set (0.04 sec)

При включении предложения `group by` сервер понимает, что надо сначала сгруппировать строки, имеющие одинаковое значение в столбце `customer_id`, а затем применить к каждой из 599 групп пять агрегатных функций.

Подсчет различных значений

При использовании функции `count()` для определения количества членов в каждой группе у вас есть выбор: подсчитать *все* элементы группы или подсчитать только *различные* значения столбца среди всех элементов группы.

Рассмотрим, например, следующий запрос, в котором функция `count()` и столбец `customer_id` используются двумя разными способами:

```

mysql> SELECT COUNT(customer_id) num_rows,
      -> COUNT(DISTINCT customer_id) num_customers
      -> FROM payment;
+-----+-----+
| num_rows | num_customers |
+-----+-----+
| 16049 | 599 |
+-----+
1 row in set (0.01 sec)

```

В первом столбце запроса просто подсчитывается количество строк в таблице `payment`, в то время как во втором столбце исследуются значения в столбце `customer_id` и подсчитывается только количество уникальных значений. Таким образом, при указании ключевого слова `distinct` функция `count()` проверяет значения столбца для каждого члена группы, находя и удаляя дубликаты, а не просто подсчитывает количество значений в группе.

Использование выражений

Наряду с использованием столбцов в качестве аргументов агрегатных функций можно использовать и выражения. Например, можно найти максимальное количество дней между моментом, когда фильм был взят напрокат, и последующим его возвратом. Эту информацию можно получить с помощью следующего запроса:

```
mysql> SELECT MAX(datediff(return_date, rental_date))
   -> FROM rental;
+-----+
| MAX(datediff(return_date, rental_date)) |
+-----+
|                               33 |
+-----+
1 row in set (0.01 sec)
```

Функция `datediff` используется для вычисления количества дней между датой возврата фильма и датой взятия его напрокат для каждой аренды фильма, а функция `max` возвращает максимальное найденное значение, которое в данном случае составляет 33 дня.

Хотя в этом примере используется довольно простое выражение, на практике выражения, используемые в качестве аргументов агрегатных функций, могут быть любой необходимой степени сложности, лишь бы они возвращали числа, строки или даты. В главе 11, “Условная логика”, показано, как для того, чтобы определить, должна ли конкретная строка быть включена в агрегацию, можно использовать выражения `case` с агрегатными функциями.

Обработка значений `null`

При выполнении агрегации (на самом деле — любого вида числовых вычислений) всегда следует учитывать, как на результат вычислений могут повлиять значения `null`. Для иллюстрации я построю простую таблицу для хранения числовых данных и заполню ее множеством {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, который выполняет пять агрегатных функций для указанного множества 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 |   3.0000 |
+-----+-----+-----+-----+
1 row in set (0.08 sec)
```

Результаты оказываются такими, как и следовало ожидать: и `count(*)`, и `count(val)` возвращают значение 3, `sum(val)` возвращает значение 9, `max(val)` возвращает 5, а `avg(val)` возвращает 3. Теперь я добавляю в таблицу `number_tbl` значение `null` и снова выполняю тот же запрос:

```
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 |   3.0000 |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Даже при добавлении в таблицу значения `null` функции `sum()`, `max()` и `avg()` возвращают те же значения; такие результаты указывают на то, что эти функции игнорируют любые встречающиеся значения `null`. Функция `count(*)` теперь возвращает значение 4, которое является корректным, поскольку таблица `number_tbl` сейчас содержит четыре строки, а функция `count(val)` по-прежнему возвращает значение 3. Дело в том, что `count(*)` подсчитывает количество строк, тогда как `count(val)` подсчитывает

количество значений, содержащихся в столбце `val`, и игнорирует любые обнаруженные в нем значения `null`.

Генерация групп

Людей редко интересуют необработанные данные. Обычно те, кто заинтересован в анализе данных, манипулируют необработанными данными так, чтобы они удовлетворяли их потребностям. Распространенными примерами манипуляции являются:

- генерация итогов для географического региона, например общий объем продаж в Европе;
- выявление отклоняющихся значений, таких как лучший продавец 2020 года;
- определение частотных показателей, например количества фильмов, взятых напрокат за месяц.

Чтобы ответить на эти типы запросов, нужно запросить сервер базы данных сгруппировать строки вместе по одному или нескольким столбцам или выражениям. Как вы уже видели в приводимых примерах, механизм для группировки данных в запросе — это предложение `group by`. Из этого раздела вы узнаете, как группировать данные по одному или нескольким столбцам, как группировать данные с помощью выражений и как создавать сводки внутри групп.

Группировка по одному столбцу

Группы из одного столбца — это самый простой и наиболее часто используемый тип группировки. Если, например, вы хотите найти количество фильмов, связанных с каждым актером, нужна группировка по единственному столбцу `film_actor.actor_id`:

```
mysql> SELECT actor_id, count(*)
   -> FROM film_actor
   -> GROUP BY actor_id;
+-----+-----+
| actor_id | count(*) |
+-----+-----+
|      1   |      19 |
|      2   |      25 |
|      3   |      22 |
|      4   |      22 |
| ...     |        |

```

```

|      197 |      33 |
|      198 |      40 |
|      199 |      15 |
|     200 |      20 |
+-----+
200 rows in set (0.11 sec)

```

Этот запрос генерирует 200 групп, по одной для каждого актера, а затем суммирует количество фильмов для каждого участника группы.

Многостолбцевая группировка

В некоторых случаях вам может потребоваться создавать группы, охватывающие более одного столбца. Расширяя предыдущий пример, представьте, что для каждого актера вы хотите найти общее число фильмов с разными рейтингами (G, PG, ...). В следующем примере показано, как этого добиться:

```

mysql> SELECT fa.actor_id, f.rating, count(*)
    -> FROM film_actor fa
    -> INNER JOIN film f
    -> ON fa.film_id = f.film_id
    -> GROUP BY fa.actor_id, f.rating
    -> ORDER BY 1,2;
+-----+-----+-----+
| actor_id | rating | count(*) |
+-----+-----+-----+
|      1 | G      |      4 |
|      1 | PG     |      6 |
|      1 | PG-13  |      1 |
|      1 | R      |      3 |
|      1 | NC-17  |      5 |
|      2 | G      |      7 |
|      2 | PG     |      6 |
|      2 | PG-13  |      2 |
|      2 | R      |      2 |
|      2 | NC-17  |      8 |
| ...
| 199 | G      |      3 |
| 199 | PG     |      4 |
| 199 | PG-13  |      4 |
| 199 | R      |      2 |
| 199 | NC-17  |      2 |
| 200 | G      |      5 |
| 200 | PG     |      3 |
| 200 | PG-13  |      2 |
| 200 | R      |      6 |
| 200 | NC-17  |      4 |
+-----+-----+-----+
996 rows in set (0.01 sec)

```

Эта версия запроса генерирует 996 групп, по одной для каждой комбинации “актер/рейтинг фильма”, полученной путем соединения таблицы `film_actor` с таблицей `film`. Столбец `rating` я добавил и в предложение `select`, и в предложение `group by`, поскольку значение `rating` извлекается из таблицы, а не генерируется с помощью агрегатной функции, такой как `max` или `count`.

Группировка с помощью выражений

Для группировки можно использовать не только данные столбцов, но и значения, генерируемые выражениями. Рассмотрим следующий запрос, который группирует прокат по годам:

```
mysql> SELECT extract(YEAR FROM rental_date) year,
->     COUNT(*) how_many
->   FROM rental
->  GROUP BY extract(YEAR FROM rental_date);
+-----+
| year | how_many |
+-----+
| 2005 |      15862 |
| 2006 |        182 |
+-----+
2 rows in set (0.01 sec)
```

В этом запросе применено довольно простое выражение, которое использует функцию `extract()` для того, чтобы вернуть из даты только год для соответствующей группировки строк в таблице `rental`.

Генерация итоговых данных

В разделе “Многостолбцевая группировка” приведен пример, в котором подсчитывается количество фильмов для каждого актера и рейтинга фильмов. Пусть, однако, вместе с общим количеством для каждой комбинации “актер/рейтинг”, нужно получить и общее количество для каждого отдельного актера. Можно выполнить дополнительный запрос и объединить результаты, можно загрузить результаты запроса в электронную таблицу или создать сценарий Python, программу Java или какой-либо иной механизм для получения этих данных и выполнения дополнительных вычислений. Но еще лучше использовать конструкцию `with rollup`, чтобы сервер базы данных сделал эту работу вместо вас. Вот измененный запрос, использующий `with rollup` в предложении `group by`:

```

mysql> SELECT fa.actor_id, f.rating, count(*)
->   FROM film_actor fa
->     INNER JOIN film f
->       ON fa.film_id = f.film_id
-> GROUP BY fa.actor_id, f.rating WITH ROLLUP
-> ORDER BY 1,2;
+-----+-----+-----+
| actor_id | rating | count(*) |
+-----+-----+-----+
| NULL     | NULL    |      5462 |
| 1         | NULL    |        19 |
| 1         | G       |        4  |
| 1         | PG      |        6  |
| 1         | PG-13   |        1  |
| 1         | R       |        3  |
| 1         | NC-17   |        5  |
| 2         | NULL    |       25  |
| 2         | G       |        7  |
| 2         | PG      |        6  |
| 2         | PG-13   |        2  |
| 2         | R       |        2  |
| 2         | NC-17   |        8  |
| ...      | ...     |      ... |
| 199      | NULL    |       15 |
| 199      | G       |        3  |
| 199      | PG      |        4  |
| 199      | PG-13   |        4  |
| 199      | R       |        2  |
| 199      | NC-17   |        2  |
| 200      | NULL    |       20 |
| 200      | G       |        5  |
| 200      | PG      |        3  |
| 200      | PG-13   |        2  |
| 200      | R       |        6  |
| 200      | NC-17   |        4  |
+-----+-----+-----+
1197 rows in set (0.07 sec)

```

Теперь в результирующем наборе имеется 201 дополнительная строка, по одной для каждого из 200 различных актеров и одна общая (для всех актеров вместе). В столбце `rating` для итоговых значений для 200 актеров предоставляется значение `null`, поскольку выполняется накопление по всем рейтингам. Взглянув, например, на первую строку для `actor_id`, равного 200, вы увидите, что всего с этим актером связано 20 фильмов; это значение равно сумме итогов по каждому рейтингу ($4 \text{ NC-17} + 6 \text{ R} + 2 \text{ PG-13} + 3 \text{ PG} + 5 \text{ G}$). Для строки общего итога (первая строка вывода) значение `null` предоставлено как для столбца `actor_id`, так и для столбца `rating`; сумма в первой строке вывода равна 5 462, что соответствует числу строк в таблице `film_actor`.



В Oracle Database нужно использовать немного другой синтаксис, указывающий, что вы хотите выполнить сведение итоговых данных. Предложение group by из предыдущего запроса при использовании Oracle выглядело бы следующим образом:

```
GROUP BY ROLLUP (fa.actor_id, f.rating)
```

Преимущество этого синтаксиса в том, что он позволяет выполнять сведение для подмножества столбцов в предложении group by. Если группировка выполняется по столбцам a, b и c, например, можно указать, что сервер должен выполнять сведение только для столбцов b и c с помощью следующей конструкции:

```
GROUP BY a, ROLLUP (b, c)
```

Если, помимо итогов по актерам, вы хотите подсчитать итоги по рейтингу, можете использовать конструкцию with cube, которая будет генерировать итоговые строки для *всех* возможных комбинаций столбцов группировки. К сожалению, конструкция with cube недоступна в MySQL версии 8.0, но доступна в SQL Server и Oracle Database.

Условия группового фильтра

В главе 4, “Фильтрация”, вы познакомились с типами условий фильтрации и узнали, как их использовать в предложении where. При группировке данных также можно применить фильтрующее условие к данным *после* того, как были сгенерированы группы. Эти типы условий фильтрации должны быть размещены в предложении having. Рассмотрим следующий пример:

```
mysql> SELECT fa.actor_id, f.rating, count(*)
    ->   FROM film_actor fa
    ->     INNER JOIN film f
    ->       ON fa.film_id = f.film_id
    -> WHERE f.rating IN ('G', 'PG')
    -> GROUP BY fa.actor_id, f.rating
    -> HAVING count(*) > 9;
+-----+-----+-----+
| actor_id | rating | count(*) |
+-----+-----+-----+
|      137 | PG     |      10 |
|       37  | PG     |      12 |
|      180  | PG     |      12 |
|        7  | G      |      10 |
|      83   | G      |      14 |
|     129   | G      |      12 |
```

```

| 111 | PG      | 15 |
| 44  | PG      | 12 |
| 26  | PG      | 11 |
| 92  | PG      | 12 |
| 17  | G       | 12 |
| 158 | PG      | 10 |
| 147 | PG      | 10 |
| 14  | G       | 10 |
| 102 | PG      | 11 |
| 133 | PG      | 10 |
+-----+-----+
16 rows in set (0.01 sec)

```

Этот запрос имеет два условия фильтрации: одно — в предложении `where`, которое отфильтровывает любые фильмы с рейтингом, отличным от `G` или `PG`, и еще одно — в предложении `having`, которое отфильтровывает всех актеров, снявшихся менее чем в 10 фильмах. Таким образом, один из фильтров действует на данные *до* их группировки, а другой — *после* того, как группы были созданы. Если вы по ошибке поместите оба фильтра в предложение `where`, то получите сообщение об ошибке:

```

mysql> SELECT fa.actor_id, f.rating, count(*)
   -> FROM film_actor fa
   -> INNER JOIN film f
   -> ON fa.film_id = f.film_id
   -> WHERE f.rating IN ('G','PG')
   -> AND count(*) > 9
   -> GROUP BY fa.actor_id, f.rating;
ERROR 1111 (HY000): Invalid use of group function3

```

Этот запрос не работает, потому что нельзя включать агрегатную функцию в предложение `where` запроса. Это связано с тем, что фильтры в предложении `where` вычисляются до группировки, поэтому сервер еще не в состоянии выполнять какие-либо функции для групп.



При добавлении фильтров в запрос, который включает предложение `group by`, хорошо подумайте, действует ли фильтр на необработанные данные, — в этом случае он должен принадлежать предложению `where`. Если же фильтр относится к сгруппированным данным, он должен принадлежать предложению `having`.

³ Неверное применение групповой функции.

Проверьте свои знания

Предлагаемые здесь упражнения призваны закрепить понимание вами возможностей группировки и агрегации SQL. Ответы к упражнениям вы найдете в приложении Б.

УПРАЖНЕНИЕ 8.1

Создайте запрос, который подсчитывает количество строк в таблице payment.

УПРАЖНЕНИЕ 8.2

Измените запрос из упражнения 8.1 так, чтобы подсчитать количество платежей, произведенных каждым клиентом. Выведите идентификатор клиента и общую уплаченную сумму для каждого клиента.

УПРАЖНЕНИЕ 8.3

Измените запрос из упражнения 8.2, включив в него только тех клиентов, у которых имеется не менее 40 выплат.

Подзапросы

Подзапросы являются мощным инструментом, который можно использовать во всех четырех инструкциях данных SQL. В этой главе показано, как использовать подзапросы для фильтрации данных, генерации значений и создания временных наборов данных. После изучения этой главы, я думаю, вы согласитесь, что подзапросы — одна из самых мощных возможностей языка SQL.

Что такое подзапрос

Подзапрос — это запрос, содержащийся в другой инструкции SQL (которую в остальной части этой главы я называю *содержащей инструкцией*). Подзапрос всегда заключен в круглые скобки и обычно выполняется перед содержащей инструкцией. Подобно любому запросу, подзапрос возвращает результирующий набор, который может состоять из:

- одной строки с одним столбцом;
- нескольких строк с одним столбцом;
- нескольких строк с несколькими столбцами.

Тип результирующего набора, возвращаемого подзапросом, определяет, как можно его использовать и какие операторы может использовать содержащая инструкция для взаимодействия с данными, возвращаемыми подзапросом. Когда содержащая инструкция завершает выполнение, данные, возвращенные любыми подзапросами, уничтожаются, что делает подзапрос действующим, как временная таблица с *областью видимости инструкции* (это означает, что сервер освобождает всю память, выделенную для результатов подзапроса, после выполнения инструкции SQL).

Вы уже видели несколько примеров подзапросов в предыдущих главах, но вот еще один простой пример для начала:

```
mysql> SELECT customer_id, first_name, last_name
-> FROM customer
-> WHERE customer_id = (SELECT MAX(customer_id) FROM customer);
+-----+-----+
| customer_id | first_name | last_name |
+-----+-----+
|      599    | AUSTIN     | CINTRON   |
+-----+-----+
1 row in set (0.27 sec)
```

В этом примере подзапрос возвращает максимальное значение, найденное в столбце `customer_id` в таблице `customer`, а затем содержащая инструкция возвращает данные об этом клиенте. Если вы сталкиваетесь с ситуацией, когда не понимаете, что делает подзапрос, можете запустить подзапрос сам по себе (без скобок), чтобы увидеть, что он возвращает. Вот результат работы подзапроса из предыдущего примера:

```
mysql> SELECT MAX(customer_id) FROM customer;
+-----+
| MAX(customer_id) |
+-----+
|      599        |
+-----+
1 row in set (0.00 sec)
```

Этот подзапрос возвращает одну строку с одним столбцом, что позволяет использовать ее как одно из выражений в условии равенства (если бы подзапрос вернул две или более строк, его можно было бы сравнивать с чем-то, но он не мог бы быть *равным* чему-либо; подробнее об этом речь пойдет ниже). В этом случае вы можете взять значение, возвращенное подзапросом, и заменить им правое выражение условия фильтрации в содержащем запросе:

```
mysql> SELECT customer_id, first_name, last_name
-> FROM customer
-> WHERE customer_id = 599;
+-----+-----+
| customer_id | first_name | last_name |
+-----+-----+
|      599    | AUSTIN     | CINTRON   |
+-----+-----+
1 row in set (0.00 sec)
```

В этом случае подзапрос полезен, поскольку позволяет получить информацию о клиенте с наибольшим идентификатором в одном запросе вместо максимального значения `customer_id` с помощью одного запроса, а затем написания второго запроса для получения требуемых данных из таблицы `customer`. Как вы увидите, подзапросы полезны и во многих других

ситуациях и могут стать одним из самых мощных ваших инструментов при работе с SQL.

Типы подзапросов

Наряду с отмеченными ранее различиями в отношении типа результирующего набора, возвращаемого подзапросом (одна строка/столбец, одна строка/несколько столбцов или несколько строк/столбцов), можно использовать другую характеристику для классификации подзапросов; одни подзапросы полностью автономные (именуются *некоррелированными подзапросами*), в то время как другие ссылаются на столбцы из содержащей инструкции (именуются *коррелированными подзапросами*). В нескольких следующих разделах рассматриваются эти два типа подзапросов и демонстрируются различные операторы, которые можно использовать для взаимодействия с ними.

Некоррелированные подзапросы

Пример, показанный выше в этой главе, — некоррелированный подзапрос, который может быть выполнен отдельно и не ссылается ни на что из содержащей инструкции. Большинство подзапросов, с которыми вы столкнетесь, будут принадлежать этому типу, если только вы не будете писать инструкции update или delete, которые часто используют коррелированные подзапросы (подробнее об этом речь пойдет ниже). Помимо того что это некоррелированный подзапрос, данный пример также возвращает результирующий набор, содержащий только одну строку и один столбец. Этот тип подзапроса известен как *скалярный подзапрос* и может появляться на любой стороне условия, использующего обычные операторы сравнения (`=`, `<`, `>`, `<=`, `>=`). В следующем примере показано, как можно использовать скалярный подзапрос в условии неравенства:

```
mysql> SELECT city_id, city
    ->   FROM city
    -> WHERE country_id <>
    ->   (SELECT country_id FROM country WHERE country = 'India');
+-----+-----+
| city_id | city           |
+-----+-----+
|      1 | A Corua (La Corua) |
|      2 | Abha             |
|      3 | Abu Dhabi        |
|      4 | Acua              |
|      5 | Adana            |
```

```

|   6 | Addis Abeba
| ...
| 595 | Zapopan
| 596 | Zaria
| 597 | Zeleznogorsk
| 598 | Zhezqazghan
| 599 | Zhoushan
| 600 | Ziguinchor
+-----+
540 rows in set (0.02 sec)

```

Этот запрос возвращает все города, находящиеся не в Индии. Подзапрос, который находится в последней строке инструкции, возвращает идентификатор страны для Индии, а содержащий его запрос возвращает все города, идентификатор страны которых не равен полученному. Хотя подзапрос в этом примере довольно прост, фактически подзапросы могут быть настолько сложными, насколько это нужно, и могут использовать любые доступные предложения запроса (`select`, `from`, `where`, `group by`, `having` и `order by`).

Если подзапрос используется в условии равенства, но возвращает более одной строки, то будет получено сообщение об ошибке. Например, если вы измените предыдущий запрос так, чтобы подзапрос возвращал *все* страны, за исключением Индии, вы получите следующее сообщение об ошибке:

```

mysql> SELECT city_id, city
    -> FROM city
    -> WHERE country_id <>
    -> (SELECT country_id FROM country WHERE country <> 'India');
ERROR 1242 (21000): Subquery returns more than 1 row1

```

Если вы выполните подзапрос отдельно, то увидите следующие результаты:

```

mysql> SELECT country_id FROM country WHERE country <> 'India';
+-----+
| country_id |
+-----+
|      1 |
|      2 |
|      3 |
|      4 |
| ...
|    106 |
|    107 |
|    108 |
|    109 |
+-----+
108 rows in set (0.00 sec)

```

¹ Подзапрос вернул более одной строки.

Содержащий запрос не выполняется, потому что выражение (`country_id`) не может быть приравнено к набору выражений (1, 2, 3, ..., 109). Другими словами, нельзя приравнять одну вещь и набор вещей. В следующем разделе вы увидите, как решить проблему с помощью другого оператора.

Подзапросы с несколькими строками и одним столбцом

Если подзапрос возвращает более одной строки, его нельзя использовать в условии равенства, как показано в предыдущем примере. Однако есть четыре дополнительных оператора, которые можно использовать для создания условий с этими типами подзапросов.

Операторы `in` и `not in`

Хотя проверить на равенство одно значение с набором значений нельзя, можно проверить, входит ли конкретное значение в набор. Следующий пример (пусть и не использующий подзапрос) демонстрирует, как создать условие, которое использует оператор `in` для поиска для значения в наборе значений:

```
mysql> SELECT country_id
-> FROM country
-> WHERE country IN ('Canada', 'Mexico');
+-----+
| country_id |
+-----+
|      20 |
|      60 |
+-----+
2 rows in set (0.00 sec)
```

Выражение в левой части условия — столбец `country`, а в правой — набор строк. Оператор `in` проверяет, входит ли строка из столбца `country` в этот набор; если да, то условие выполняется и строка добавляется к результирующему набору. Тот же результат можно получить, используя два условия равенства, например:

```
mysql> SELECT country_id
-> FROM country
-> WHERE country = 'Canada' OR country = 'Mexico';
+-----+
| country_id |
+-----+
|      20 |
|      60 |
+-----+
2 rows in set (0.00 sec)
```

Хотя такой подход кажется разумным, когда набор содержит только два выражения, легко понять, почему одно условие с использованием оператора `in` предпочтительнее, когда множество содержит десятки (или сотни, тысячи и т.д.) значений.

Хотя иногда набор строк, дат или чисел для использования с одной стороны условия создается вручную, куда чаще такой набор создается с помощью подзапроса, который возвращает одну или несколько строк. В следующем запросе оператор `in` используется с подзапросом в правой части условия фильтра, чтобы получить все города, которые находятся в Канаде или Мексике:

```
mysql> SELECT city_id, city
->   FROM city
-> WHERE country_id IN
->   (SELECT country_id
->     FROM country
-> WHERE country IN ('Canada', 'Mexico'));
+-----+-----+
| city_id | city
+-----+-----+
|    179 | Gatineau
|    196 | Halifax
|    300 | Lethbridge
|    313 | London
|    383 | Oshawa
|    430 | Richmond Hill
|    565 | Vancouver
| ...
|    452 | San Juan Bautista Tuxtepec
|    541 | Torren
|    556 | Uruapan
|    563 | Valle de Santiago
|    595 | Zapopan
+-----+
37 rows in set (0.00 sec)
```

Помимо проверки, имеется ли некоторое значение в наборе, вы можете проверить обратное, используя оператор `not in`. Вот еще одна версия предыдущего запроса, использующая `not in` вместо `in`:

```
mysql> SELECT city_id, city
->   FROM city
-> WHERE country_id NOT IN
->   (SELECT country_id
->     FROM country
-> WHERE country IN ('Canada', 'Mexico'));
+-----+-----+
| city_id | city
+-----+-----+
```

```

|   1 | A Corua (La Corua)      |
|   2 | Abha                     |
|   3 | Abu Dhabi                 |
|   5 | Adana                     |
|   6 | Addis Abeba               |
| ...
| 596 | Zaria                   |
| 597 | Zeleznogorsk            |
| 598 | Zhezqazghan              |
| 599 | Zhoushan                 |
| 600 | Ziguinchor                |
+-----+
563 rows in set (0.00 sec)

```

Этот запрос находит все города, расположенные не в Канаде или Мексике.

Оператор all

В то время как оператор `in` используется, чтобы выяснить, имеется ли выражение в наборе выражений, оператор `all` позволяет сравнивать отдельное значение и каждое значение в наборе. Чтобы создать такое условие, нужно использовать один из операторов сравнения (`=`, `<>`, `<`, `>` и т.д.) в сочетании с оператором `all`. Например, следующий запрос находит всех клиентов, которые никогда не получали фильмы напрокат бесплатно:

```

mysql> SELECT first_name, last_name
    -> FROM customer
    -> WHERE customer_id <> ALL
    -> (SELECT customer_id
    ->   FROM payment
    ->   WHERE amount = 0);
+-----+-----+
| first_name | last_name  |
+-----+-----+
| MARY       | SMITH      |
| PATRICIA   | JOHNSON    |
| LINDA      | WILLIAMS   |
| BARBARA    | JONES      |
| ...
| EDUARDO    | HIATT      |
| TERRENCE   | GUNDERSON |
| ENRIQUE    | FORSYTHE   |
| FREDDIE    | DUGGAN     |
| WADE       | DELVALLE   |
| AUSTIN     | CINTRON    |
+-----+-----+
576 rows in set (0.01 sec)

```

Подзапрос возвращает набор идентификаторов клиентов, которые заплатили 0 долларов за прокат фильма, а содержащий запрос возвращает имена

всех клиентов, чьи идентификаторы не указаны в результирующем наборе, возвращенном подзапросом. Если этот подход кажется вам несколько неуклюжим, вы в хорошей компании: большинство людей предпочли бы сформулировать запрос иначе и избежать использования оператора `all`. Для иллюстрации — предыдущий запрос дает те же результаты, что и следующий пример, в котором используется оператор `not in`:

```
SELECT first_name, last_name
FROM customer
WHERE customer_id NOT IN
  (SELECT customer_id
   FROM payment
   WHERE amount = 0)
```

Выбор того или иного запроса — вопрос вкуса, но я думаю, что большинство людей сочтут версию, которая использует `not in`, более легкой для понимания.



При использовании `not in` или `<> all` для сравнения значения с набором значений нужно быть осторожным и убедиться, что набор значений не содержит значения `null`, потому что сервер выполняет приравнивание значения с левой стороны выражения к каждому члену набора, а любая попытка приравнять значение к `null` дает `unknown`. Таким образом, следующий запрос возвращает пустой результирующий набор:

```
mysql> SELECT first_name, last_name
    -> FROM customer
    -> WHERE customer_id NOT IN (122, 452, NULL);
Empty set (0.00 sec)
```

Вот еще один пример с использованием оператора `all`, но на этот раз подзапрос находится вложении `having`:

```
mysql> SELECT customer_id, count(*)
    -> FROM rental
    -> GROUP BY customer_id
    -> HAVING count(*) > ALL
    ->  (SELECT count(*)
    ->  FROM rental r
    ->      INNER JOIN customer c
    ->      ON r.customer_id = c.customer_id
    ->      INNER JOIN address a
    ->      ON c.address_id = a.address_id
    ->      INNER JOIN city ct
    ->      ON a.city_id = ct.city_id
    ->      INNER JOIN country co
```

```

->      ON ct.country_id = co.country_id
-> WHERE co.country IN ('United States', 'Mexico', 'Canada')
-> GROUP BY r.customer_id
-> );
+-----+-----+
| customer_id | count(*) |
+-----+-----+
|       148 |      46 |
+-----+-----+
1 row in set (0.01 sec)

```

Подзапрос в этом примере возвращает общее количество прокатов фильмов для каждого клиента в Северной Америке, а содержащий запрос возвращает всех клиентов, общее количество прокатов фильмов у которых превышает значение у любого из североамериканских клиентов.

Оператор any

Как и оператор all, оператор any позволяет сравнивать значение с членами набора значений; в отличие от all, однако, условие, использующее оператор any, вычисляется как истинное, как только найдется хотя бы одно выполняющееся сравнение. В приведенном далее примере выполняется поиск всех клиентов, чьи суммарные платежи за прокат фильмов превышают суммарные платежи всех клиентов в Боливии, Парагвае или Чили:

```

mysql> SELECT customer_id, sum(amount)
-> FROM payment
-> GROUP BY customer_id
-> HAVING sum(amount) > ANY
->   (SELECT sum(p.amount)
->     FROM payment p
->       INNER JOIN customer c
->         ON p.customer_id = c.customer_id
->       INNER JOIN address a
->         ON c.address_id = a.address_id
->       INNER JOIN city ct
->         ON a.city_id = ct.city_id
->       INNER JOIN country co
->         ON ct.country_id = co.country_id
-> WHERE co.country IN ('Bolivia', 'Paraguay', 'Chile')
-> GROUP BY co.country
-> );
+-----+-----+
| customer_id | sum(amount) |
+-----+-----+
|       137 |    194.61 |
|       144 |    195.58 |
|       148 |    216.54 |
|       178 |    194.61 |
+-----+-----+

```

```
|      459 |      186.62 |
|      526 |      221.55 |
+-----+
6 rows in set (0.03 sec)
```

Подзапрос возвращает общую стоимость проката фильмов для всех клиентов в Боливии, Парагвае и Чили, а содержащий запрос возвращает всех клиентов, которые израсходовали сумму, превышающую расходы клиентов хотя бы одной из этих трех стран.



Хотя большинство людей предпочитают использовать `in`, применение `= any` эквивалентно применению оператора `in`.

Многостолбцовые подзапросы

До сих пор примеры подзапросов в этой главе возвращали один столбец и одну или несколько строк. Однако в определенных ситуациях можно использовать подзапросы, возвращающие два или более столбцов. Помочь показать полезность многостолбцовых подзапросов может приведенный далее пример, в котором используется несколько подзапросов с одним столбцом:

```
mysql> SELECT fa.actor_id, fa.film_id
   -> FROM film_actor fa
   -> WHERE fa.actor_id IN
   ->   (SELECT actor_id FROM actor WHERE last_name = 'MONROE')
   ->   AND fa.film_id IN
   ->   (SELECT film_id FROM film WHERE rating = 'PG');
+-----+-----+
| actor_id | film_id |
+-----+-----+
|      120 |      63 |
|      120 |     144 |
|      120 |     414 |
|      120 |     590 |
|      120 |     715 |
|      120 |     894 |
|      178 |     164 |
|      178 |     194 |
|      178 |     273 |
|      178 |     311 |
|      178 |     983 |
+-----+-----+
11 rows in set (0.00 sec)
```

В этом запросе для идентификации всех участников с фамилией MONROE и всех фильмов с рейтингом PG используются два подзапроса, а затем содержа-

щий запрос использует эту информацию для извлечения всех случаев, когда актер с этой фамилией появляется в фильме с рейтингом PG. Однако можно объединить два подзапроса с одним столбцом в один подзапрос с несколькими столбцами и сравнивать результаты с двумя столбцами таблицы `film_actor`. Для этого условие фильтра должно указывать два столбца из таблицы `film_actor` в круглых скобках и в том же порядке, что и в подзапросе:

```
mysql> SELECT actor_id, film_id
-> FROM film_actor
-> WHERE (actor_id, film_id) IN
->  (SELECT a.actor_id, f.film_id
->   FROM actor a
->     CROSS JOIN film f
->   WHERE a.last_name = 'MONROE'
->   AND f.rating = 'PG');
+-----+-----+
| actor_id | film_id |
+-----+-----+
|      120 |      63 |
|      120 |     144 |
|      120 |     414 |
|      120 |     590 |
|      120 |     715 |
|      120 |     894 |
|      178 |     164 |
|      178 |     194 |
|      178 |     273 |
|      178 |     311 |
|      178 |     983 |
+-----+-----+
11 rows in set (0.00 sec)
```

Эта версия запроса выполняет ту же функцию, что и в предыдущем примере, но с использованием одного подзапроса, который возвращает два столбца вместо двух подзапросов, каждый из которых возвращает один столбец. Подзапрос в этой версии использует тип соединения, именуемого *перекрестным соединением* (которое будет рассмотрено в следующей главе). Основная идея — вернуть все комбинации актеров по имени Monroe (2) и всех фильмов с рейтингом PG (194), всего — 388 строк, 11 из которых могут быть найдены в таблице `film_actor`.

Коррелированные подзапросы

Все подзапросы, показанные до сих пор, не зависели от содержащихся в них операторов. Это означает, что вы можете выполнить их автономно

и проверить возвращаемые ими результаты. Коррелированный же подзапрос зависит от содержащей его инструкции, ссылаясь на один или несколько ее столбцов. В отличие от некоррелированного подзапроса, коррелированный подзапрос не выполняется один раз перед выполнением содержащей его инструкции; вместо этого коррелированный подзапрос выполняется по одному разу для каждой строки-кандидата (строки, которая может быть включена в окончательный результат). Например, в следующем запросе используется коррелированный подзапрос для подсчета количества прокатов фильмов для каждого клиента, а содержащий запрос затем извлекает тех клиентов, которые взяли напрокат ровно 20 фильмов:

```
mysql> SELECT c.first_name, c.last_name
    -> FROM customer c
    -> WHERE 20 =
    -> (SELECT count(*) FROM rental r
    -> WHERE r.customer_id = c.customer_id);
+-----+-----+
| first_name | last_name |
+-----+-----+
| LAUREN     | HUDSON    |
| JEANETTE   | GREENE    |
| TARA        | RYAN      |
| WILMA       | RICHARDS  |
| JO          | FOWLER    |
| KAY         | CALDWELL  |
| DANIEL     | CABRAL    |
| ANTHONY    | SCHWAB   |
| TERRY       | GRISSOM   |
| LUIS        | YANEZ     |
| HERBERT    | KRUGER    |
| OSCAR       | AQUINO    |
| RAUL        | FORTIER   |
| NELSON     | CHRISTENSON |
| ALFREDO    | MCADAMS   |
+-----+
15 rows in set (0.01 sec)
```

Ссылка на `c.customer_id` в самом конце подзапроса делает этот подзапрос коррелированным; содержащий запрос должен предоставлять значения для `c.customer_id`, чтобы подзапрос мог быть выполнен. В данном случае содержащий запрос извлекает все 599 строк из таблицы `customer` и выполняет подзапрос по одному разу для каждого клиента, передавая при каждом выполнении соответствующий идентификатор клиента. Если подзапрос возвращает значение 20, условие фильтра выполняется и строка добавляется к результирующему набору.



Предупреждение: поскольку коррелированный подзапрос будет выполняться по одному разу для каждой строки содержащего запроса, использование коррелированных подзапросов может привести к проблемам с производительностью, особенно если содержащий запрос возвращает большое количество строк.

Помимо условий равенства, можно использовать коррелированные подзапросы в условиях других типов, таких, например, как условие диапазона, показанное далее:

```
mysql> SELECT c.first_name, c.last_name
->   FROM customer c
-> WHERE
->   (SELECT sum(p.amount) FROM payment p
->    WHERE p.customer_id = c.customer_id)
->   BETWEEN 180 AND 240;
+-----+
| first_name | last_name |
+-----+
| RHONDA    | KENNEDY   |
| CLARA     | SHAW       |
| ELEANOR   | HUNT       |
| MARION    | SNYDER    |
| TOMMY     | COLLAZO   |
| KARL      | SEAL       |
+-----+
6 rows in set (0.03 sec)
```

Этот вариант предыдущего запроса находит всех клиентов, чьи общие платежи за все прокаты фильмов составляют от 180 до 240 долларов. И вновь коррелированный подзапрос здесь используется 599 раз (по одному разу для каждой строки клиента); каждое выполнение подзапроса возвращает общий баланс для данного клиента.



Еще одно тонкое отличие показанного запроса заключается в том, что этот подзапрос находится в левой части условия (что может выглядеть немного странно, но совершенно корректно).

Оператор exists

Хотя вы часто будете встречаться с коррелированными подзапросами, используемыми в условиях равенства и диапазона, наиболее распространенный оператор, используемый для создания условий с коррелированными подзапросами — это оператор `exists`. Оператор `exists` используется, когда

нужно определить существование связи безотносительно к количеству; например, следующий запрос находит всех клиентов, которые взяли напрокат хотя бы один фильм до 25 мая 2005 года, без учета того, сколько фильмов было взято:

```
mysql> SELECT c.first_name, c.last_name
-> FROM customer c
-> WHERE EXISTS
-> (SELECT 1 FROM rental r
-> WHERE r.customer_id = c.customer_id
-> AND date(r.rental_date) < '2005-05-25');
+-----+
| first_name | last_name |
+-----+
| CHARLOTTE | HUNTER   |
| DELORES   | HANSEN   |
| MINNIE    | ROMERO   |
| CASSANDRA | WALTERS  |
| ANDREW    | PURDY    |
| MANUEL    | MURRELL  |
| TOMMY     | COLLAZO  |
| NELSON    | CHRISTENSON |
+-----+
8 rows in set (0.03 sec)
```

Используя оператор `exists`, ваш подзапрос может возвращать нуль, одну или несколько строк, и условие просто проверяет, вернул ли подзапрос хотя бы одну строку. Если посмотреть на предложение `select` подзапроса, то видно, что он состоит из единственного литерала (1), так как условию в содержащем запросе нужно только знать, сколько строк было возвращено, фактические данные, возвращенные подзапросом, значения не имеют. Ваш подзапрос может возвращать все что угодно, как показано ниже:

```
mysql> SELECT c.first_name, c.last_name
-> FROM customer c
-> WHERE EXISTS
-> (SELECT r.rental_date, r.customer_id, 'ABCD' str, 2*3/7 nmbr
-> FROM rental r
-> WHERE r.customer_id = c.customer_id
-> AND date(r.rental_date) < '2005-05-25');
+-----+
| first_name | last_name |
+-----+
| CHARLOTTE | HUNTER   |
| DELORES   | HANSEN   |
| MINNIE    | ROMERO   |
| CASSANDRA | WALTERS  |
| ANDREW    | PURDY    |
+-----+
```

```
| MANUEL      | MURRELL      |
| TOMMY       | COLLAZO      |
| NELSON      | CHRISTENSON |
+-----+
8 rows in set (0.03 sec)
```

Однако при использовании `exists` принято указывать либо `select 1`, либо `select *`.

Вы также можете использовать `not exists` для отбора подзапросов, которые не возвращают строк, как показано ниже:

```
mysql> SELECT a.first_name, a.last_name
->   FROM actor a
-> WHERE NOT EXISTS
->   (SELECT 1
->     FROM film_actor fa
->       INNER JOIN film f ON f.film_id = fa.film_id
->      WHERE fa.actor_id = a.actor_id
->        AND f.rating = 'R');
+-----+-----+
| first_name | last_name |
+-----+-----+
| JANE       | JACKMAN    |
+-----+-----+
1 row in set (0.00 sec)
```

Этот запрос находит всех актеров, которые никогда не снимались в фильмах с рейтингом R.

Работа с данными с помощью коррелированных подзапросов

Все приведенные до сих пор в главе примеры были инструкциями `select`, но не думайте, что это означает, что подзапросы бесполезны в других инструкциях SQL. Подзапросы также широко используются в инструкциях `update`, `delete` и `insert`, причем особенно часто коррелированные подзапросы появляются в инструкциях `update` и `delete`. Вот пример коррелированного подзапроса, используемого для изменения столбца `last_update` в таблице `customer`:

```
UPDATE customer c
SET c.last_update =
(SELECT max(r.rental_date) FROM rental r
 WHERE r.customer_id = c.customer_id);
```

Эта инструкция изменяет каждую строку в таблице клиентов (поскольку в ней нет предложения `where`), находя последнюю дату проката для каждого

клиента в таблице rental. Хотя кажется разумным ожидать, что у каждого клиента будет хотя бы один прокат фильма, все же лучше всего проверить это, прежде чем пытаться обновить столбец last_update; в противном случае для столбца будет установлено значение NULL, поскольку подзапрос не вернет никакой строки. Вот еще одна версия инструкции update, на этот раз использующая предложение where со вторым коррелированным подзапросом:

```
UPDATE customer c
SET c.last_update =
  (SELECT max(r.rental_date) FROM rental r
   WHERE r.customer_id = c.customer_id)
WHERE EXISTS
  (SELECT 1 FROM rental r
   WHERE r.customer_id = c.customer_id);
```

Эти два коррелированных подзапроса идентичны, за исключением предложений select. Однако подзапрос в предложении set выполняется, только если условие в предложении where инструкции update истинно (т.е. если для клиента был найден хотя бы один прокат), тем самым защищая данные в столбце last_update от перезаписывания значением null.

Коррелированные подзапросы также распространены в инструкциях delete. Например, вы можете запускать сценарий обслуживания данных в конце каждого месяца, который удаляет ненужные данные. Сценарий может включать следующую инструкцию, которая удаляет те строки из таблицы customer, для которых в прошлом году не было проката фильмов:

```
DELETE FROM customer
WHERE 365 < ALL
  (SELECT datediff(now(), r.rental_date) days_since_last_rental
   FROM rental r
   WHERE r.customer_id = customer.customer_id);
```

При использовании коррелированных подзапросов с инструкциями delete в MySQL имейте в виду, что по какой-то причине псевдонимы таблиц при использовании delete не разрешены — вот почему мне пришлось использовать полное имя таблицы в подзапросе. В большинстве других серверов баз данных можно указать псевдоним для таблицы customer, например:

```
DELETE FROM customer c
WHERE 365 < ALL
  (SELECT datediff(now(), r.rental_date) days_since_last_rental
   FROM rental r
   WHERE r.customer_id = c.customer_id);
```

Применение подзапросов

Теперь, когда вы узнали о различных типах подзапросов и различных операторах, которые можно использовать для взаимодействия с данными, возвращаемыми подзапросами, пришло время изучить множество способов использования подзапросов для создания мощных инструкций SQL. В следующих трех разделах показано, как можно использовать подзапросы для создания пользовательских таблиц, построения условий и генерации значений столбцов в результирующих наборах.

Подзапросы как источники данных

Еще в главе 3, “Запросы”, было указано, что предложение `from` инструкции `select` содержит *таблицы*, которые будут использоваться запросом. Поскольку подзапрос генерирует результирующий набор, содержащий строки и столбцы данных, вполне допустимо включать подзапросы в предложение `from` вместе с таблицами. Хотя на первый взгляд это может показаться интересной возможностью без особой практической ценности, использование подзапросов вместе с таблицами является одним из самых мощных инструментов, доступных при написании запросов. Вот простой пример:

```
mysql> SELECT c.first_name, c.last_name,
->   pymnt.num_rentals, pymnt.tot_payments
-> FROM customer c
->   INNER JOIN
->     (SELECT customer_id,
->           count(*) num_rentals, sum(amount) tot_payments
->      FROM payment
->      GROUP BY customer_id
->    ) pymnt
->   ON c.customer_id = pymnt.customer_id;
+-----+-----+-----+-----+
| first_name | last_name | num_rentals | tot_payments |
+-----+-----+-----+-----+
| MARY       | SMITH    |      32 |      118.68 |
| PATRICIA  | JOHNSON  |      27 |      128.73 |
| LINDA     | WILLIAMS |      26 |      135.74 |
| BARBARA   | JONES    |      22 |      81.78  |
| ELIZABETH | BROWN    |      38 |      144.62 |
| ...        |          |          |          |
| TERENCE   | GUNDERSON |      30 |      117.70 |
| ENRIQUE   | FORSYTHE  |      28 |      96.72  |
| FREDDIE   | DUGGAN   |      25 |      99.75  |
| WADE      | DELVALLE |      22 |      83.78  |
+-----+-----+-----+-----+
```

```
| AUSTIN      | CINTRON      |      19 |      83.81 |
+-----+-----+-----+
599 rows in set (0.03 sec)
```

В этом примере подзапрос генерирует список идентификаторов клиентов вместе с количеством прокатов фильмов и общими платежами. Вот как выглядит результирующий набор, сгенерированный подзапросом:

```
mysql> SELECT customer_id, count(*) num_rentals,
       sum(amount) tot_payments
   -> FROM payment
   -> GROUP BY customer_id;
+-----+-----+-----+
| customer_id | num_rentals | tot_payments |
+-----+-----+-----+
|      1 |      32 |     118.68 |
|      2 |      27 |     128.73 |
|      3 |      26 |     135.74 |
|      4 |      22 |      81.78 |
| ...          |           |           |
|    596 |      28 |      96.72 |
|    597 |      25 |      99.75 |
|    598 |      22 |      83.78 |
|    599 |      19 |      83.81 |
+-----+-----+-----+
599 rows in set (0.03 sec)
```

Подзапрос получает имя `rentmt` и соединяется с таблицей `customer` через столбец `customer_id`. Затем содержащий запрос извлекает имя клиента из таблицы `customer` вместе со сводными столбцами из подзапроса `rentmt`.

Подзапросы, используемые в предложении `from`, должны быть некоррелированными²; они выполняются первыми, и их данные хранятся в памяти до тех пор, пока не завершится выполнение содержащего запроса. При написании запросов подзапросы предлагают огромную гибкость, потому что вы можете выйти далеко за рамки имеющегося множества доступных таблиц для создания практически любого требуемого представления данных, с последующим соединением результатов с другими таблицами или подзапросами. При написании отчетов или генерации потоков данных во внешние системы можно с помощью одного запроса решать задачи, которые иначе требовали бы выполнения нескольких запросов или применения процедурного языка программирования.

² Фактически в зависимости от используемого сервера базы данных можно включать в предложение `from` коррелированные подзапросы с помощью конструкций `cross apply` или `external apply`, но эти возможности выходят за рамки данной книги.

Создание данных

Наряду с использованием подзапросов для подытоживания существующих данных можно использовать подзапросы для генерации данных, которых в базе данных нет ни в какой форме. Например, можно сгруппировать клиентов по сумме денег, потраченной на прокат фильмов, но при этом вы хотите использовать определения групп, которых нет в вашей базе данных. Например, допустим, что вы хотите распределить клиентов по группам, показанным в табл. 9.1.

Таблица 9.1. Группировка клиентов по платежам

Группа	Нижняя граница, долл.	Верхняя граница, долл.
Small Fry	0	74.99
Average Joes	75	149.99
Heavy Hitters	150	9 999 999.99

Чтобы сгенерировать эти группы в рамках одного запроса, требуется способ определить эти три группы. Первым шагом является создание запроса, который генерирует определения групп:

```
mysql> SELECT 'Small Fry' name, 0 low_limit, 74.99 high_limit
-> UNION ALL
-> SELECT 'Average Joes' name, 75 low_limit, 149.99 high_limit
-> UNION ALL
-> SELECT 'Heavy Hitters' name, 150 low_limit,
->                               9999999.99 high_limit;
+-----+-----+-----+
| name      | low_limit | high_limit |
+-----+-----+-----+
| Small Fry |      0 |    74.99 |
| Average Joes | 75 | 149.99 |
| Heavy Hitters | 150 | 9999999.99 |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

Я использовал оператор union all, чтобы объединить результаты трех отдельных запросов в единый результирующий набор. Каждый запрос извлекает три литерала, а результаты трех запросов объединяются для создания результирующего набора с тремя строками и тремя столбцами. Теперь, когда у вас есть запрос для создания требуемых групп, его можно поместить в предложение from другого запроса для генерации групп клиентов:

```
mysql> SELECT pymnt_grps.name, count(*) num_customers
-> FROM
-> (SELECT customer_id,
->       count(*) num_rentals, sum(amount) tot_payments
```

```

->   FROM payment
->   GROUP BY customer_id
-> ) pymnt
-> INNER JOIN
->   (SELECT 'Small Fry' name, 0 low_limit, 74.99 high_limit
-> UNION ALL
->   SELECT 'Average Joes' name, 75 low_limit, 149.99 high_limit
-> UNION ALL
->   SELECT 'Heavy Hitters' name, 150 low_limit,
->                      9999999.99 high_limit
-> ) pymnt_grps
->   ON pymnt.tot_payments
->   BETWEEN pymnt_grps.low_limit AND pymnt_grps.high_limit
-> GROUP BY pymnt_grps.name;
+-----+-----+
| name          | num_customers |
+-----+-----+
| Average Joes |          515 |
| Heavy Hitters |           46 |
| Small Fry     |           38 |
+-----+-----+
3 rows in set (0.03 sec)

```

Предложение `from` содержит два подзапроса; первый подзапрос, с именем `pymnt`, возвращает общее количество прокатов фильмов и общие платежи для каждого клиента, в то время как второй подзапрос, с именем `pymnt_grps`, генерирует три группы клиентов. Два подзапроса объединяются путем определения, к какой из трех групп принадлежит каждый покупатель, а затем строки группируются по имени группы для подсчета количества клиентов в каждой группе.

Конечно, вы можете просто создать постоянную (или временную) таблицу для хранения определений групп вместо использования подзапроса. Используя такой подход, вы обнаружите, что через некоторое время ваша база данных будет завалена небольшими таблицами специального назначения, и вы не сможете вспомнить причину, по которой было создано большинство из них. Однако, используя подзапросы, вы сможете придерживаться политики, согласно которой таблицы добавляются в базу данных только тогда, когда существует явная бизнес-потребность в хранении новых данных.

Подзапросы, ориентированные на задачу

Допустим, вы хотите создать отчет, в котором будут указаны имя каждого клиента, а также его город, общее количество прокатов и общая сумма платежа. Это можно сделать, соединив таблицы `payment`, `customer`, `address` и `city`, а затем сгруппировав их по имени и фамилии клиента:

```

mysql> SELECT c.first_name, c.last_name, ct.city,
->     sum(p.amount) tot_payments, count(*) tot_rentals
->   FROM payment p
->   INNER JOIN customer c
->     ON p.customer_id = c.customer_id
->   INNER JOIN address a
->     ON c.address_id = a.address_id
->   INNER JOIN city ct
->     ON a.city_id = ct.city_id
-> GROUP BY c.first_name, c.last_name, ct.city;
+-----+-----+-----+-----+
| first_name | last_name | city      | tot_payments | tot_rentals |
+-----+-----+-----+-----+
| MARY       | SMITH     | Sasebo    | 118.68      | 32         |
| PATRICIA  | JOHNSON   | San Bernardo | 128.73      | 27         |
| LINDA      | WILLIAMS  | Athenai   | 135.74      | 26         |
| BARBARA   | JONES     | Myingyan  | 81.78       | 22         |
| ...        |           |           |             |             |
| TERENCE   | GUNDERSON | Jinzhou  | 117.70      | 30         |
| ENRIQUE   | FORSYTHE  | Patras    | 96.72       | 28         |
| FREDDIE   | DUGGAN    | Sullana   | 99.75       | 25         |
| WADE       | DELVALLE  | Lausanne  | 83.78       | 22         |
| AUSTIN    | CINTRON   | Tieli     | 83.81       | 19         |
+-----+-----+-----+-----+
599 rows in set (0.06 sec)

```

Этот запрос возвращает желаемые данные, но если вы внимательно на него посмотрите, то увидите, что таблицы customer, address и city нужны только для отображения и что в таблице payment есть все необходимое для создания группировок (customer_id и amount). Таким образом, вы можете выделить задачу создания групп в подзапросе, а затем присоединить остальные три таблицы к таблице, сгенерированной подзапросом, для достижения желаемого конечного результата. Вот какой вид имеет подзапрос группировки:

```

mysql> SELECT customer_id,
->     count(*) tot_rentals, sum(amount) tot_payments
->   FROM payment
-> GROUP BY customer_id;
+-----+-----+-----+
| customer_id | tot_rentals | tot_payments |
+-----+-----+-----+
| 1           | 32         | 118.68      |
| 2           | 27         | 128.73      |
| 3           | 26         | 135.74      |
| 4           | 22         | 81.78       |
| ...         |            |             |
| 595         | 30         | 117.70      |
| 596         | 28         | 96.72       |
+-----+-----+-----+

```

```

|      597 |        25 |     99.75 |
|      598 |        22 |     83.78 |
|      599 |        19 |     83.81 |
+-----+

```

599 rows in set (0.03 sec)

Это центральная часть запроса; прочие таблицы нужны только для предоставления значимых строк вместо значения `customer_id`. Следующий запрос соединяет предыдущий набор данных с тремя другими таблицами:

```

mysql> SELECT c.first_name, c.last_name,
->   ct.city,
->   pymnt.tot_payments, pymnt.tot_rentals
->   FROM
->   (SELECT customer_id,
->         count(*) tot_rentals, sum(amount) tot_payments
->   FROM payment
->   GROUP BY customer_id
->   ) pymnt
->   INNER JOIN customer c
->   ON pymnt.customer_id = c.customer_id
->   INNER JOIN address a
->   ON c.address_id = a.address_id
->   INNER JOIN city ct
->   ON a.city_id = ct.city_id;
+-----+-----+-----+-----+
|first_name |last_name |city          |tot_payments |tot_rentals |
+-----+-----+-----+-----+

```

first_name	last_name	city	tot_payments	tot_rentals
MARY	SMITH	Sasebo	118.68	32
PATRICIA	JOHNSON	San Bernardino	128.73	27
LINDA	WILLIAMS	Athenai	135.74	26
BARBARA	JONES	Myingyan	81.78	22
...				
TERRENCE	GUNDERSON	Jinzhou	117.70	30
ENRIQUE	FORSYTHE	Patras	96.72	28
FREDDIE	DUGGAN	Sullana	99.75	25
WADE	DELVALLE	Lausanne	83.78	22
AUSTIN	CINTRON	Tieli	83.81	19

599 rows in set (0.06 sec)

Я понимаю, что “красота — в глазах смотрящего”, но я считаю, что эта версия запроса гораздо более привлекательна, чем большая “плоская” версия. Этот запрос, кроме того, может выполняться быстрее, поскольку группировка выполняется по одному числовому столбцу (`customer_id`), а не по нескольким столбцам с длинными строками (`customer.first_name`, `customer.last_name`, `city.city`).

Обобщенные табличные выражения

Обобщенные табличные выражения (Common table expressions, CTE), которые появились в MySQL в версии 8.0, уже были доступны на других серверах баз данных в течение некоторого времени. СТЕ — это именованный подзапрос, который появляется в верхней части запроса в предложении `with`, которое может содержать несколько СТЕ через запятую. Помимо того что запросы при этом становятся более понятными, это также позволяет каждому СТЕ обращаться к любому другому СТЕ, определенному над ним в том же предложении `with`. Следующий пример включает три обобщенных табличных выражения, причем второе ссылается на первое, а третье — на второе:

```
mysql> WITH actors_s AS
->   (SELECT actor_id, first_name, last_name
->     FROM actor
->    WHERE last_name LIKE 'S%'
->  ),
->   actors_s_pg AS
->   (SELECT s.actor_id, s.first_name, s.last_name,
->          f.film_id, f.title
->     FROM actors_s s
->       INNER JOIN film_actor fa
->         ON s.actor_id = fa.actor_id
->       INNER JOIN film f
->         ON f.film_id = fa.film_id
->    WHERE f.rating = 'PG'
->  ),
->   actors_s_pg_revenue AS
->   (SELECT spg.first_name, spg.last_name, p.amount
->     FROM actors_s_pg spg
->       INNER JOIN inventory i
->         ON i.film_id = spg.film_id
->       INNER JOIN rental r
->         ON i.inventory_id = r.inventory_id
->       INNER JOIN payment p
->         ON r.rental_id = p.rental_id
->  ) - конец предложения WITH
->   SELECT spg_rev.first_name, spg_rev.last_name,
->          sum(spg_rev.amount) tot_revenue
->   FROM actors_s_pg_revenue spg_rev
->   GROUP BY spg_rev.first_name, spg_rev.last_name
->   ORDER BY 3 desc;
+-----+-----+-----+
| first_name | last_name | tot_revenue |
+-----+-----+-----+
| NICK       | STALLONE   |      692.21 |
| JEFF        | SILVERSTONE |      652.35 |
| DAN         | STREEP      |      509.02 |
```

```

| GROUCHO      | SINATRA        |      457.97 |
| SISSY        | SOBIESKI       |      379.03 |
| JAYNE        | SILVERSTONE    |      372.18 |
| CAMERON      | STREEP         |      361.00 |
| JOHN          | SUVARI         |      296.36 |
| JOE           | SWANK          |      177.52 |
+-----+
9 rows in set (0.18 sec)

```

Этот запрос вычисляет общий доход от проката тех фильмов с рейтингом PG, актерский состав которых включает актера, фамилия которого начинается с S. Первый подзапрос (`actors_s`) находит всех актеров, чьи фамилии начинаются с S, второй подзапрос (`actors_s_pg`) соединяет этот набор данных с таблицей `film` и фильтрует фильмы с рейтингом PG, а третий подзапрос (`actors_s_pg_revenue`) соединяет этот набор данных с таблицей `payment`, чтобы узнать суммы, уплаченные за аренду любого из этих фильмов. Последний запрос просто группирует данные из `actors_s_pg_revenue` по имени/фамилии и суммирует доходы.



Все, кто склонны использовать временные таблицы для хранения результатов запросов для их использования в последующих запросах, могут счесть СТЕ привлекательной альтернативой.

Подзапросы как генераторы выражений

В этом разделе главы я заканчиваю то, с чего начал: одностолбцовые, однострочные скалярные подзапросы. Скалярные подзапросы могут использоваться не только в условиях фильтрации, но и везде, где может появиться выражение, включая предложения `select` и `order by` запроса и предложение `value` инструкции `insert`.

В разделе “Подзапросы, ориентированные на задачу” было показано, как использовать подзапросы для отделения механизма группировки от остальной части запроса. Вот еще одна версия того же запроса, который использует подзапросы для той же цели, но иным способом:

```

mysql> SELECT
->   (SELECT c.first_name FROM customer c
->     WHERE c.customer_id = p.customer_id
->   ) first_name,
->   (SELECT c.last_name FROM customer c
->     WHERE c.customer_id = p.customer_id
->   ) last_name,
->   (SELECT ct.city
->     FROM customer c

```

```

->      INNER JOIN address a
->          ON c.address_id = a.address_id
->      INNER JOIN city ct
->          ON a.city_id = ct.city_id
-> WHERE c.customer_id = p.customer_id
-> ) city,
-> sum(p.amount) tot_payments,
-> count(*) tot_rentals
-> FROM payment p
-> GROUP BY p.customer_id;
+-----+-----+-----+-----+-----+
| first_name | last_name | city           | tot_payments | tot_rentals |
+-----+-----+-----+-----+-----+
| MARY       | SMITH      | Sasebo        |    118.68   |     32      |
| PATRICIA   | JOHNSON    | San Bernardino |    128.73   |     27      |
| LINDA      | WILLIAMS   | Athenai       |    135.74   |     26      |
| BARBARA    | JONES      | Myingyan     |     81.78   |     22      |
| ...        |            |               |             |             |
| TERENCE    | GUNDERSON | Jinzhou      |    117.70   |     30      |
| ENRIQUE    | FORSYTHE   | Patras        |     96.72   |     28      |
| FREDDIE    | DUGGAN     | Sullana       |     99.75   |     25      |
| WADE       | DELVALLE   | Lausanne      |     83.78   |     22      |
| AUSTIN     | CINTRON    | Tieli         |     83.81   |     19      |
+-----+-----+-----+-----+-----+
599 rows in set (0.06 sec)

```

Между этим запросом и более ранней его версией, использующей подзапрос в предложении `from`, имеется два основных различия.

- Вместо того чтобы соединять таблицы `customer`, `address` и `city` с данными о платежах, коррелированные скалярные подзапросы используются в предложении `select` для поиска имени/фамилии и города клиента.
- К таблице `customer` имеется три обращения (по одному в каждом из трех подзапросов), а не одно.

К таблице `customer` выполняется три обращения, потому что скалярные подзапросы могут возвращать только один столбец и строку, поэтому, если нам нужны три столбца, связанные с клиентом, необходимо использовать три разных подзапроса.

Как отмечалось ранее, скалярные подзапросы могут появляться и в предложении `order by`. Следующий запрос извлекает имена и фамилии актеров и сортирует их по количеству фильмов, в которых снялся актер:

```
mysql> SELECT a.actor_id, a.first_name, a.last_name
-> FROM actor a
-> ORDER BY
```

```

-> (SELECT count(*) FROM film_actor fa
-> WHERE fa.actor_id = a.actor_id) DESC;
+-----+-----+-----+
| actor_id | first_name | last_name |
+-----+-----+-----+
| 107 | GINA | DEGENERES |
| 102 | WALTER | TORN |
| 198 | MARY | KEITEL |
| 181 | MATTHEW | CARREY |
| ... |
| 71 | ADAM | GRANT |
| 186 | JULIA | ZELLWEGER |
| 35 | JUDY | DEAN |
| 199 | JULIA | FAWCETT |
| 148 | EMILY | DEE |
+-----+-----+-----+
200 rows in set (0.01 sec)

```

Этот запрос использует коррелированный скалярный подзапрос в предложении `order by` только для возврата количества фильмов, и это значение используется исключительно для сортировки.

Наряду с коррелированными скалярными подзапросами в инструкциях `select` можно использовать некоррелированные скалярные подзапросы для генерации значений для инструкции `insert`. Пусть, например, вы собираетесь создать новую строку в таблице `film_actor` и у вас имеются следующие данные:

- имя и фамилия актера;
- название фильма.

Есть два варианта, как поступить: выполнить два запроса, чтобы получить значения первичных ключей из таблиц `film` и `actor` и поместить эти значения в инструкцию `insert`; или использовать для получения двух значений ключей в инструкции `insert` подзапросы. Вот пример последнего подхода:

```

INSERT INTO film_actor (actor_id, film_id, last_update)
VALUES (
  (SELECT actor_id FROM actor
   WHERE first_name = 'JENNIFER' AND last_name = 'DAVIS'),
  (SELECT film_id FROM film
   WHERE title = 'ACE GOLDFINGER'),
  now()
);

```

В заключение

В этой главе затрагивалось много вопросов, поэтому было бы неплохо бегло повторить основные тезисы. Примеры в этой главе демонстрируют подзапросы, которые:

- возвращают один столбец и одну строку, один столбец с несколькими строками и несколько столбцов и строк;
- не зависят от содержащей инструкции (некоррелированные подзапросы);
- ссылаются на один или несколько столбцов из содержащей инструкции (коррелированные подзапросы);
- используются в условиях, в которых используются операторы сравнения, а также специальные операторы `in`, `not in`, `exists` и `not exists`;
- могут применяться в инструкциях `select`, `update`, `delete` и `insert`;
- могут создавать результирующие наборы, которые можно соединять с другими таблицами (или подзапросами) в запросе;
- могут использоваться для генерации значений для заполнения таблицы или столбцов в результирующем наборе запроса;
- используются в предложениях `select`, `from`, `where`, `having` и `order by` запросов.

Очевидно, что подзапросы являются очень мощным универсальным инструментом, поэтому не расстраивайтесь, если вы не смогли усвоить все эти концепции с первого прочтения главы. Продолжайте экспериментировать с различными вариантами использования подзапросов, и вскоре вы поймете себя на том, что каждый раз, когда вы пишете нетривиальную инструкцию SQL, вы думаете о том, как бы использовать в ней подзапрос.

Проверьте свои знания

Предлагаемые здесь упражнения призваны закрепить понимание вами подзапросов. Ответы к упражнениям вы найдете в приложении Б.

УПРАЖНЕНИЕ 9.1

Создайте запрос к таблице `film`, который использует условие фильтрации с некоррелированным подзапросом к таблице `category`, чтобы найти все боевики (`category.name = 'Action'`).

УПРАЖНЕНИЕ 9.2

Переработайте запрос из упражнения 9.1, используя *коррелированный подзапрос* к таблицам category и film_category для получения тех же результатов.

УПРАЖНЕНИЕ 9.3

Соедините следующий запрос с подзапросом к таблице film_actor, чтобы показать уровень мастерства каждого актера:

```
SELECT 'Hollywood Star' level, 30 min_roles, 99999 max_roles  
UNION ALL  
SELECT 'Prolific Actor' level, 20 min_roles, 29 max_roles  
UNION ALL  
SELECT 'Newcomer' level, 1 min_roles, 19 max_roles
```

Подзапрос к таблице film_actor должен подсчитывать количество строк для каждого актера с использованием group by actor_id, и результат подсчета должен сравниваться со столбцами min_roles/max_roles, чтобы определить, какой уровень мастерства имеет каждый актер.

Соединения

К настоящему времени вы уже должны быть хорошо знакомы с концепцией внутреннего соединения, которая была представлена в главе 5, “Запросы к нескольким таблицам”. В этой главе рассматриваются другие способы соединения таблиц, включая внешнее соединение и перекрестное соединение.

Внешние соединения

До сих пор во всех примерах, включающих несколько таблиц, нас не волновало, что условия соединения могут не найти совпадений для всех строк таблицы. Например, таблица `inventory` содержит строку для каждого фильма, доступного для проката, но из 1000 строк в таблице `film` только 958 имеют одну или несколько строк в таблице `inventory`. Остальные 42 фильма для проката недоступны (возможно, это новинки, которые должны прибыть в пункты проката со дня на день), поэтому идентификаторы этих фильмов в таблице `inventory` отсутствуют. Следующий запрос подсчитывает количество доступных копий каждого фильма с помощью соединения этих двух таблиц:

```
mysql> SELECT f.film_id, f.title, count(*) num_copies
-> FROM film f
->     INNER JOIN inventory i
->         ON f.film_id = i.film_id
-> GROUP BY f.film_id, f.title;
+-----+-----+-----+
| film_id | title           | num_copies |
+-----+-----+-----+
|      1 | ACADEMY DINOSAUR |      8 |
|      2 | ACE GOLDFINGER   |      3 |
|      3 | ADAPTATION HOLES |      4 |
|      4 | AFFAIR PREJUDICE |      7 |
| ...    |                   |          |
|     13 | ALI FOREVER      |      4 |
|     15 | ALIEN CENTER     |      6 |
```

```

| ...          |
| 997 | YOUTH KICK      |           2 |
| 998 | ZHIVAGO CORE    |           2 |
| 999 | ZOOLANDER FICTION|           5 |
| 1000 | ZORRO ARK        |           8 |
+-----+
958 rows in set (0.02 sec)

```

Хотя можно было бы ожидать, что будет возвращено 1000 строк (по одной для каждого фильма), запрос возвращает только 958 строк. Дело в том, что запрос использует внутреннее соединение, которое возвращает только строки, удовлетворяющие условию соединения. Например, фильм *Alice Fantasia* (`film_id` равен 14) не отображается в результатах, потому что для него нет строк в таблице `inventory`.

Если вы хотите, чтобы запрос возвращал все 1000 фильмов, независимо от того, имеются ли соответствующие строки в таблице `inventory`, можете использовать внешнее соединение, которое, по сути, делает условие соединения необязательным:

```

mysql> SELECT f.film_id, f.title, count(i.inventory_id) num_copies
   -> FROM film f
   -> LEFT OUTER JOIN inventory i
   ->     ON f.film_id = i.film_id
   -> GROUP BY f.film_id, f.title;
+-----+-----+-----+
| film_id | title            | num_copies |
+-----+-----+-----+
| 1       | ACADEMY DINOSAUR | 8          |
| 2       | ACE GOLDFINGER   | 3          |
| 3       | ADAPTATION HOLES | 4          |
| 4       | AFFAIR PREJUDICE | 7          |
| ...     |                   |           |
| 13      | ALI FOREVER      | 4          |
| 14      | ALICE FANTASIA   | 0          |
| 15      | ALIEN CENTER     | 6          |
| ...     |                   |           |
| 997     | YOUTH KICK      | 2          |
| 998     | ZHIVAGO CORE    | 2          |
| 999     | ZOOLANDER FICTION| 5          |
| 1000    | ZORRO ARK        | 8          |
+-----+-----+-----+
1000 rows in set (0.01 sec)

```

Как видите, запрос теперь возвращает все 1000 строк из таблицы `film`, при этом 42 строки из общего количества строк (включая фильм *Alice Fantasia*) имеют значение 0 в столбце `num_copies`, что указывает на отсутствие доступных для проката копий.

Вот описание изменений по сравнению с предыдущей версией запроса.

- Определение соединения было изменено с `inner` на `left outer`, что указывает серверу на необходимость включения всех строк из таблицы в левой части соединения (в данном случае `film`) и включения столбцов из таблицы с правой стороны соединения (`inventory`), если соединение прошло успешно.
- Определение столбца `num_copies` было изменено с `count(*)` на `count(i.inventory_id)`; это выражение подсчитывает количество значений столбца `inventory.inventory_id`, не равных `null`.

Теперь давайте удалим предложение `group by` и отфильтруем большинство строк, чтобы ясно видеть различия между внутренними и внешними соединениями. Вот запрос с использованием внутреннего соединения и условия фильтрации для возврата всего лишь нескольких фильмов:

```
mysql> SELECT f.film_id, f.title, i.inventory_id
->   FROM film f
->     INNER JOIN inventory i
->       ON f.film_id = i.film_id
-> WHERE f.film_id BETWEEN 13 AND 15;
+-----+-----+-----+
| film_id | title      | inventory_id |
+-----+-----+-----+
|    13 | ALI FOREVER |        67 |
|    13 | ALI FOREVER |        68 |
|    13 | ALI FOREVER |        69 |
|    13 | ALI FOREVER |        70 |
|    15 | ALIEN CENTER |       71 |
|    15 | ALIEN CENTER |       72 |
|    15 | ALIEN CENTER |       73 |
|    15 | ALIEN CENTER |       74 |
|    15 | ALIEN CENTER |       75 |
|    15 | ALIEN CENTER |       76 |
+-----+-----+-----+
10 rows in set (0.00 sec)
```

Результаты показывают, что в прокате имеется четыре копии *Ali Forever* и шесть копий *Alien Center*. Вот как выглядит тот же запрос, но с использованием внешнего соединения:

```
mysql> SELECT f.film_id, f.title, i.inventory_id
->   FROM film f
->     LEFT OUTER JOIN inventory i
->       ON f.film_id = i.film_id
-> WHERE f.film_id BETWEEN 13 AND 15;
+-----+-----+-----+
```

```

| film_id | title           | inventory_id |
+-----+-----+-----+
|    13 | ALI FOREVER     |      67 |
|    13 | ALI FOREVER     |      68 |
|    13 | ALI FOREVER     |      69 |
|    13 | ALI FOREVER     |      70 |
|   14 | ALICE FANTASIA |      NULL |
|    15 | ALIEN CENTER    |      71 |
|    15 | ALIEN CENTER    |      72 |
|    15 | ALIEN CENTER    |      73 |
|    15 | ALIEN CENTER    |      74 |
|    15 | ALIEN CENTER    |      75 |
|    15 | ALIEN CENTER    |      76 |
+-----+-----+-----+
11 rows in set (0.00 sec)

```

Результаты для *Ali Forever* и *Alien Center* остаются прежними, но появляется одна новая строка для *Alice Fantasia*, со значением null в столбце `inventory.inventory_id`. В этом примере показано, как внешнее соединение добавляет значения столбцов без ограничения на количество строк, возвращаемых запросом. Если условие соединения не выполняется (как в случае *Alice Fantasia*), то любые столбцы, извлеченные из внешне соединенной таблицы, будут иметь значения null.

Левое и правое внешние соединения

В примерах внешнего соединения в предыдущем разделе было указано `left outer join`. Ключевое слово `left` указывает, что таблица в левой части соединения отвечает за определения количества строк в результирующем наборе, тогда как таблица в правой части используется для предоставления значений столбца всякий раз, когда найдено совпадение. Однако можно указать и правое внешнее соединение — в этом случае таблица с правой стороны от `right outer join` отвечает за определение количества строк в результирующем наборе, тогда как таблица с левой стороны используется для предоставления значений столбцов.

Вот последний запрос из предыдущего раздела, в котором используется правое внешнее соединение вместо левого:

```

mysql> SELECT f.film_id, f.title, i.inventory_id
-> FROM inventory i
-> RIGHT OUTER JOIN film f
->     ON f.film_id = i.film_id
-> WHERE f.film_id BETWEEN 13 AND 15;
+-----+-----+-----+
| film_id | title           | inventory_id |
+-----+-----+-----+

```

13	ALI FOREVER		67	
13	ALI FOREVER		68	
13	ALI FOREVER		69	
13	ALI FOREVER		70	
14	ALICE FANTASIA		NULL	
15	ALIEN CENTER		71	
15	ALIEN CENTER		72	
15	ALIEN CENTER		73	
15	ALIEN CENTER		74	
15	ALIEN CENTER		75	
15	ALIEN CENTER		76	

11 rows in set (0.00 sec)

Заметим, что обе версии запроса выполняют внешнее соединение; ключевые слова `left` и `right` служат только для того, чтобы сообщить серверу, в какой таблице разрешено иметь пробелы в данных. Если вы хотите внешне соединить таблицы A и B и хотите, чтобы в результирующем наборе присутствовали все строки из A (с дополнительными столбцами из B всякий раз, когда есть соответствующие данные), то можете указать либо A `left outer join` B, либо B `right outer join` A.



Поскольку правые внешние соединения встречаются крайне редко (и не все серверы баз данных их поддерживают), лучше всегда использовать левые внешние соединения. Ключевое слово `outer` необязательное, так что можно просто писать A `left join` B, но я рекомендую включать слово `outer` для ясности.

Трехсторонние внешние соединения

В некоторых случаях вам может потребоваться внешнее соединение одной таблицы с двумя другими. Например, запрос из предыдущего раздела можно расширить, включив в него данные из таблицы `rental`:

```
mysql> SELECT f.film_id, f.title, i.inventory_id, r.rental_date
-> FROM film f
-> LEFT OUTER JOIN inventory i
-> ON f.film_id = i.film_id
-> LEFT OUTER JOIN rental r
-> ON i.inventory_id = r.inventory_id
-> WHERE f.film_id BETWEEN 13 AND 15;
+-----+-----+-----+
| film_id | title      | inventory_id | rental_date      |
+-----+-----+-----+
|     13  | ALI FOREVER |           67 | 2005-07-31 18:11:17 |
```

13	ALI FOREVER		67	2005-08-22 21:59:29	
13	ALI FOREVER		68	2005-07-28 15:26:20	
13	ALI FOREVER		68	2005-08-23 05:02:31	
13	ALI FOREVER		69	2005-08-01 23:36:10	
13	ALI FOREVER		69	2005-08-22 02:12:44	
13	ALI FOREVER		70	2005-07-12 10:51:09	
13	ALI FOREVER		70	2005-07-29 01:29:51	
13	ALI FOREVER		70	2006-02-14 15:16:03	
14	ALICE FANTASIA	NULL	NULL		
15	ALIEN CENTER		71	2005-05-28 02:06:37	
15	ALIEN CENTER		71	2005-06-17 16:40:03	
15	ALIEN CENTER		71	2005-07-11 05:47:08	
15	ALIEN CENTER		71	2005-08-02 13:58:55	
15	ALIEN CENTER		71	2005-08-23 05:13:09	
15	ALIEN CENTER		72	2005-05-27 22:49:27	
15	ALIEN CENTER		72	2005-06-19 13:29:28	
15	ALIEN CENTER		72	2005-07-07 23:05:53	
15	ALIEN CENTER		72	2005-08-01 05:55:13	
15	ALIEN CENTER		72	2005-08-20 15:11:48	
15	ALIEN CENTER		73	2005-07-06 15:51:58	
15	ALIEN CENTER		73	2005-07-30 14:48:24	
15	ALIEN CENTER		73	2005-08-20 22:32:11	
15	ALIEN CENTER		74	2005-07-27 00:15:18	
15	ALIEN CENTER		74	2005-08-23 19:21:22	
15	ALIEN CENTER		75	2005-07-09 02:58:41	
15	ALIEN CENTER		75	2005-07-29 23:52:01	
15	ALIEN CENTER		75	2005-08-18 21:55:01	
15	ALIEN CENTER		76	2005-06-15 08:01:29	
15	ALIEN CENTER		76	2005-07-07 18:31:50	
15	ALIEN CENTER		76	2005-08-01 01:49:36	
15	ALIEN CENTER		76	2005-08-17 07:26:47	

32 rows in set (0.01 sec)

Результаты включают в себя все фильмы, имеющиеся в наличии, но у фильма *Alice Fantasia* в столбцах из обеих таблиц, соединенных внешним соединением, находятся значения null.

Перекрестные соединения

Еще в главе 5, “Запросы к нескольким таблицам”, была представлена концепция декартова произведения, которая, по сути, представляет собой результат соединения нескольких таблиц без указания каких-либо условий соединения. Декартово произведение довольно часто используется случайно (например, если вы забыли добавить условие соединения в предложение `from`), но преднамеренное его применение не так уж распространено. Если

все же вы намереваетесь генерировать декартово произведение двух таблиц, то должны указать *перекрестное соединение*, как в следующем примере:

```
mysql> SELECT c.name category_name, l.name language_name
    -> FROM category c
    -> CROSS JOIN language l;
+-----+-----+
| category_name | language_name |
+-----+-----+
| Action        | English          |
| Action        | Italian           |
| Action        | Japanese          |
| Action        | Mandarin          |
| Action        | French            |
| Action        | German             |
| Animation     | English           |
| Animation     | Italian           |
| Animation     | Japanese          |
| Animation     | Mandarin          |
| Animation     | French            |
| Animation     | German             |
| ...           |                  |
| Sports         | English           |
| Sports         | Italian           |
| Sports         | Japanese          |
| Sports         | Mandarin          |
| Sports         | French            |
| Sports         | German             |
| Travel         | English           |
| Travel         | Italian           |
| Travel         | Japanese          |
| Travel         | Mandarin          |
| Travel         | French            |
| Travel         | German             |
+-----+-----+
96 rows in set (0.00 sec)
```

Этот запрос генерирует декартово произведение таблиц category и language, в результате чего получается результирующий набор из 96 строк (16 строк category \times 6 строк language). Теперь, когда вы знаете, что такое перекрестное соединение и как его указать, возникает вопрос “А для чего оно используется?” В большинстве книг по SQL после описания, что такое перекрестное соединение, просто говорится, что оно редко бывает полезным. Но я хотел бы поделиться с вами ситуацией, в которой я считаю перекрестное соединение весьма полезным.

В главе 9, “Подзапросы”, рассматривалось, как использовать подзапросы для создания таблиц и как построить таблицу из трех строк, которую можно

объединить с другими таблицами. Вот собранная таким образом таблица из примера:

```
mysql> SELECT 'Small Fry' name, 0 low_limit, 74.99 high_limit
-> UNION ALL
-> SELECT 'Average Joes' name, 75 low_limit, 149.99 high_limit
-> UNION ALL
-> SELECT 'Heavy Hitters' name, 150 low_limit,
      9999999.99 high_limit;
+-----+-----+-----+
| name      | low_limit | high_limit |
+-----+-----+-----+
| Small Fry |          0 |       74.99 |
| Average Joes |      75 |     149.99 |
| Heavy Hitters |    150 | 9999999.99 |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

Хотя эта таблица — именно то, что нужно для разделения клиентов на три группы на основе суммы их платежей, такая стратегия объединения однострочных таблиц с использованием оператора union all — все же не самый лучший вариант в случае больших таблиц.

Пусть, например, вы хотите создать запрос, который генерирует по одной строке на каждый день 2020 года, но в базе данных нет таблицы, которая содержит по строке для каждого дня. Используя стратегию из упомянутого примера, вы можете сделать что-то вроде следующего:

```
SELECT '2020-01-01' dt
UNION ALL
SELECT '2020-01-02' dt
UNION ALL
SELECT '2020-01-03' dt
UNION ALL
...
...
...
SELECT '2020-12-29' dt
UNION ALL
SELECT '2020-12-30' dt
UNION ALL
SELECT '2020-12-31' dt
```

Создание запроса, объединяющего результаты 366 запросов, несколько утомительно, поэтому, пожалуй, нужна другая стратегия. Что если вы создадите таблицу с 366 строками (2020 — високосный год) с одним столбцом, содержащим числа от 0 до 366, а затем добавите это количество дней к 1 января 2020 года? Вот один из возможных способов создания такой 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 вы возьмете декартово произведение трех множеств, $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, $\{0, 10, 20, 30, 40, 50, 60, 70, 80, 90\}$ и $\{0, 100, 200, 300\}$, и сложите значения в трех столбцах, то получите набор результатов из 400 строк, содержащий все числа от 0 до 399. Хотя это больше, чем 366 строк, необходимых для создания набора дней в 2020 году, избавиться от лишних строк очень легко, и вскоре я это продемонстрирую.

Следующим шагом является преобразование набора чисел в набор дат. Для этого я буду использовать функцию `date_add()` для добавления каждого числа в результирующем наборе к 1 января 2020 года. Затем я добавлю условие фильтрации, чтобы отбросить все даты, относящиеся к 2021 году:

```
mysql> SELECT DATE_ADD('2020-01-01',
->   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('2020-01-01',
->   INTERVAL (ones.num+tens.num+hundreds.num) DAY)
->   < '2021-01-01'
-> ORDER BY 1;
+-----+
| dt      |
+-----+
| 2020-01-01 |
| 2020-01-02 |
| 2020-01-03 |
| 2020-01-04 |
| 2020-01-05 |
| 2020-01-06 |
| 2020-01-07 |
| 2020-01-08 |
| ...
| ...
| ...
| 2020-02-26 |
| 2020-02-27 |
| 2020-02-28 |
| 2020-02-29 |
| 2020-03-01 |
| 2020-03-02 |
| 2020-03-03 |
| ...
| ...
| ...
| 2020-12-24 |
| 2020-12-25 |
| 2020-12-26 |
| 2020-12-27 |
| 2020-12-28 |
| 2020-12-29 |
| 2020-12-30 |
| 2020-12-31 |
+-----+
366 rows in set (0.03 sec)

```

Приятным в этом подходе является то, что набор результатов автоматически включает дополнительный високосный день (29 февраля) без вашего вмешательства, поскольку сервер базы данных вычисляет его, когда добавляет 59 дней к 1 января 2020 года.

Теперь, когда у вас есть механизм для создания всех дней в 2020 году, что же с ним делать? Ну, вас могут попросить создать отчет, который будет показывать каждый день в 2020 году вместе с количеством прокатов фильмов в этот день. Отчет должен содержать каждый день года, включая дни, когда

фильмы напрокат никто не брал. Вот как может выглядеть такой запрос (с использованием 2005 года для сопоставления данных в таблице rental):

```
mysql> SELECT days.dt, COUNT(r.rental_id) num_rentals
-> FROM rental r
-> RIGHT OUTER JOIN
-> (SELECT DATE_ADD('2005-01-01',
->     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('2005-01-01',
->     INTERVAL (ones.num + tens.num + hundreds.num) DAY)
->             < '2006-01-01'
-> ) days
-> ON days.dt = date(r.rental_date)
-> GROUP BY days.dt
-> ORDER BY 1;
+-----+-----+
| dt      | num_rentals |
+-----+-----+
| 2005-01-01 |          0 |
| 2005-01-02 |          0 |
| 2005-01-03 |          0 |
| 2005-01-04 |          0 |
| ...      |          0 |
| 2005-05-23 |          0 |
```

2005-05-24	8
2005-05-25	137
2005-05-26	174
2005-05-27	166
2005-05-28	196
2005-05-29	154
2005-05-30	158
2005-05-31	163
2005-06-01	0
...	
2005-06-13	0
2005-06-14	16
2005-06-15	348
2005-06-16	324
2005-06-17	325
2005-06-18	344
2005-06-19	348
2005-06-20	331
2005-06-21	275
2005-06-22	0
...	
2005-12-27	0
2005-12-28	0
2005-12-29	0
2005-12-30	0
2005-12-31	0

365 rows in set (8.99 sec)

Это один из наиболее интересных запросов в книге, поскольку он включает перекрестные соединения, внешние соединения, функцию для работы с датой, группировку, операцию с множествами (`union all`) и агрегатную функцию (`count()`). Это тоже не самое элегантное решение данной задачи, но оно должно служить примером того, как при небольшом творчестве и твердом знании языка можно сделать даже такую редко используемую функцию, как перекрестное соединение, мощным инструментом в своем наборе средств SQL.

Естественные соединения

Если вы ленивы, то можете выбрать тип соединения, который позволит вам указать таблицы, которые необходимо соединить, но при этом позволит серверу базы данных самому определять, какими должны быть условия соединения. Этот тип соединения, известный как *естественное соединение*, основывается при выводе условий соединения на идентичности имен столбцов

в нескольких таблицах. Например, таблица rental включает столбец с именем customer_id, который является внешним ключом к таблице customer, первичный ключ которой также имеет имя customer_id. Таким образом, вы можете попробовать написать запрос, который использует естественное соединение двух таблиц:

```
mysql> SELECT c.first_name, c.last_name, date(r.rental_date)
-> FROM customer c
->     NATURAL JOIN rental r;
Empty set (0.04 sec)
```

Поскольку вы указали естественное соединение, сервер проверяет определения таблиц и добавляет условие r.customer_id = c.customer_id для соединения двух таблиц. Это могло бы получиться, но в схеме Sakila все таблицы включают столбец last_update, чтобы знать, когда каждая строка была в последний раз изменена, поэтому сервер добавляет еще одно условие соединения — r.last_update = c.last_update, которое и приводит к тому, что запрос не возвращает никакие данные.

Единственный способ обойти эту проблему — использовать подзапрос, чтобы ограничить столбцы как минимум одной из таблиц:

```
mysql> SELECT cust.first_name, cust.last_name, date(r.rental_date)
-> FROM
->     (SELECT customer_id, first_name, last_name
->      FROM customer
->      ) cust
->     NATURAL JOIN rental r;
+-----+-----+-----+
| first_name | last_name | date(r.rental_date) |
+-----+-----+-----+
| MARY       | SMITH     | 2005-05-25      |
| MARY       | SMITH     | 2005-05-28      |
| MARY       | SMITH     | 2005-06-15      |
| MARY       | SMITH     | 2005-06-15      |
| MARY       | SMITH     | 2005-06-15      |
| MARY       | SMITH     | 2005-06-16      |
| MARY       | SMITH     | 2005-06-18      |
| MARY       | SMITH     | 2005-06-18      |
| ...        |           |                 |
| AUSTIN     | CINTRON   | 2005-08-21      |
| AUSTIN     | CINTRON   | 2005-08-21      |
| AUSTIN     | CINTRON   | 2005-08-21      |
| AUSTIN     | CINTRON   | 2005-08-23      |
| AUSTIN     | CINTRON   | 2005-08-23      |
| AUSTIN     | CINTRON   | 2005-08-23      |
+-----+-----+-----+
16044 rows in set (0.03 sec)
```

Подумайте и решите, стоит ли снижение износа пальцев и клавиатуры (благодаря отказу от указания условия соединения) дополнительных хлопот? Вряд ли! Так что следует избегать этого типа соединения и использовать внутренние соединения с явными условиями.

Проверьте свои знания

Предлагаемые здесь упражнения призваны закрепить понимание вами внешнего и перекрестного соединений. Ответы к упражнениям представлены в приложении Б.

УПРАЖНЕНИЕ 10.1

Используя следующие определения таблиц и данные, напишите запрос, который возвращает имя каждого клиента вместе с его суммами платежей:

Customer:

Customer_id	Name
1	John Smith
2	Kathy Jones
3	Greg Oliver

Payment:

Payment_id	Customer_id	Amount
101	1	8.99
102	3	4.99
103	1	7.99

Включите в результирующий набор всех клиентов, даже если для клиента нет записей о платежах.

УПРАЖНЕНИЕ 10.2

Измените запрос из упражнения 10.1 таким образом, чтобы использовать другой тип внешнего соединения (например, если вы использовали левое внешнее соединение в упражнении 10.1, на этот раз используйте правое внешнее соединение) так, чтобы результаты были идентичны полученным ранее.

УПРАЖНЕНИЕ 10.3

Разработайте запрос, который будет генерировать набор {1, 2, 3, ..., 99, 100}. (Указание: используйте перекрестное соединение как минимум с двумя подзапросами в предложении from.)

Условная логика

В некоторых ситуациях требуется, чтобы SQL-логика разветвлялась в том или ином направлении в зависимости от значений определенных столбцов или выражений. Из этой главы вы узнаете, как писать операторы, которые могут вести себя по-разному в зависимости от данных, полученных во время выполнения инструкции. Механизм условной логики в инструкциях SQL — это выражение case, которое можно использовать в инструкциях select, insert, update и delete.

Что такое условная логика

Условная логика — это просто возможность выбрать один из нескольких путей во время выполнения программы. Например, при запросе информации о клиенте вы можете захотеть включить столбец customer.active, в котором хранится 1 для обозначения активного и 0 — для обозначения неактивного клиента. Если результаты запроса используются для создания отчета, то вы можете захотеть перевести это значение в строку, чтобы улучшить удобочитаемость. Хотя каждая база данных включает встроенные функции для подобных ситуаций, общих стандартов нет, так что нужно запоминать, в какой базе данных какие функции имеются. К счастью, каждая реализация базы данных SQL включает выражение case, которое полезно во многих ситуациях, в том числе и для простого перевода:

```
mysql> SELECT first_name, last_name,
->   CASE
->     WHEN active = 1 THEN 'ACTIVE'
->     ELSE 'INACTIVE'
->   END activity_type
->   FROM customer;
+-----+-----+-----+
| first_name | last_name | activity_type |
+-----+-----+-----+
| MARY       | SMITH    | ACTIVE      |
| PATRICIA  | JOHNSON  | ACTIVE      |
```

LINDA	WILLIAMS	ACTIVE
BARBARA	JONES	ACTIVE
ELIZABETH	BROWN	ACTIVE
JENNIFER	DAVIS	ACTIVE
...		
KENT	ARSENAULT	ACTIVE
TERRANCE	ROUSH	INACTIVE
RENE	MCALISTER	ACTIVE
EDUARDO	HIATT	ACTIVE
TERRENCE	GUNDERSON	ACTIVE
ENRIQUE	FORSYTHE	ACTIVE
FREDDIE	DUGGAN	ACTIVE
WADE	DELVALLE	ACTIVE
AUSTIN	CINTRON	ACTIVE

599 rows in set (0.00 sec)

Этот запрос включает выражение case для генерации значения столбца activity_type, которое возвращает строку ACTIVE или INACTIVE в зависимости от значения столбца customer.active.

Выражение case

Все основные серверы баз данных включают встроенные функции, имитирующие инструкцию if-then-else, имеющуюся в большинстве языков программирования (например, в Oracle — функция decode(), в MySQL — if(), в SQL Server — coalesce()). Выражения case также предназначены для облегчения применения логики if-then-else, но при этом имеют следующие преимущества перед встроенными функциями.

- Выражение case является частью стандарта SQL (выпуск SQL92) и реализовано в Oracle Database, SQL Server, MySQL, PostgreSQL, IBM UDB и других СУБД.
- Выражения case встроены в грамматику SQL и могут быть включены в инструкции select, insert, update и delete.

В следующих двух подразделах представлены два разных типа выражений case. Материал сопровождается некоторыми примерами case-выражений в действии.

Поисковые выражения case

Выражение case, показанное ранее в этой главе, является примером поискового выражения case, имеющего следующий синтаксис:

```
CASE
    WHEN C1 THEN E1
    WHEN C2 THEN E2
    ...
    WHEN CN THEN EN
    [ELSE ED]
END
```

В этом определении символы C1, C2, ..., CN представляют условия, а символы E1, E2, ..., EN — выражения, возвращаемые выражением case. Если условие в предложении when вычисляется как истинное, выражение case возвращает соответствующее выражение E#. Кроме того, символ ED представляет выражение по умолчанию, которое выражение case возвращает, если ни одно из условий C1, C2, ..., CN не дает при вычислении true (предложение else является необязательным, поэтому оно заключено в квадратные скобки). Все выражения, возвращаемые различными предложениями when, должны иметь один и тот же тип (например, date, number, varchar).

Вот пример поискового выражения case:

```
CASE
    WHEN category.name IN ('Children','Family','Sports','Animation')
        THEN 'All Ages'
    WHEN category.name = 'Horror'
        THEN 'Adult'
    WHEN category.name IN ('Music','Games')
        THEN 'Teens'
    ELSE 'Other'
END
```

Это выражение case возвращает строку, которую можно использовать для классификации фильмов в зависимости от их категории. При обработке выражения case по порядку сверху вниз вычисляются предложения when; как только одно из условий в предложении when дает значение true, возвращается соответствующее выражение, а все оставшиеся предложения when игнорируются. Если ни одно из условий в предложениях when не является истинным, возвращается выражение в предложении else.

Хотя предыдущий пример возвращает строковые выражения, имейте в виду, что выражения case могут возвращать выражения любого типа, включая подзапросы. Вот еще одна версия показанного выше запроса, в которой для возврата количество прокатов — но только для активных клиентов! — используется подзапрос:

```
mysql> SELECT c.first_name, c.last_name,
->     CASE
->         WHEN active = 0 THEN 0
```

```

->      ELSE
->          (SELECT count(*) FROM rental r
->             WHERE r.customer_id = c.customer_id)
->      END num_rentals
->  FROM customer c;
+-----+-----+-----+
| first_name | last_name | num_rentals |
+-----+-----+-----+
| MARY       | SMITH     |      32 |
| PATRICIA   | JOHNSON   |      27 |
| LINDA      | WILLIAMS  |      26 |
| BARBARA    | JONES     |      22 |
| ELIZABETH  | BROWN     |      38 |
| JENNIFER   | DAVIS     |      28 |
| ...        |           |         |
| TERRANCE   | ROUSH     |       0 |
| RENE        | MCALISTER |      26 |
| EDUARDO    | HIATT     |      27 |
| TERRENCE   | GUNDERSON |      30 |
| ENRIQUE    | FORSYTHE  |      28 |
| FREDDIE    | DUGGAN    |      25 |
| WADE        | DELVALLE  |      22 |
| AUSTIN     | CINTRON   |      19 |
+-----+-----+-----+

```

599 rows in set (0.01 sec)

Эта версия запроса использует коррелированный подзапрос для получения количества прокатов для каждого активного клиента. В зависимости от процента активных клиентов использование этого подхода может быть более эффективным, чем соединение таблиц `customer` и `rental` и группировка по столбцу `customer_id`.

Простые выражения case

Простое выражение `case` очень похоже на поисковое выражение `case`, но является немного менее гибким. Вот его синтаксис.

```

CASE V0
  WHEN V1 THEN E1
  WHEN V2 THEN E2
  ...
  WHEN VN THEN EN
  [ELSE ED]
END

```

В этом определении `V0` представляет значение, а символы `V1, V2, ..., VN` — значения, с которыми необходимо сравнивать `V0`. Символы `E1, E2, ..., EN` обозначают выражения, которые должны быть возвращены выражением

case, а ED — выражение, которое должно быть возвращено, если ни одно из значений в наборе V1, V2, ..., VN не совпадает со значением V0.

Вот пример простого выражения case:

```
CASE category.name
    WHEN 'Children' THEN 'All Ages'
    WHEN 'Family' THEN 'All Ages'
    WHEN 'Sports' THEN 'All Ages'
    WHEN 'Animation' THEN 'All Ages'
    WHEN 'Horror' THEN 'Adult'
    WHEN 'Music' THEN 'Teens'
    WHEN 'Games' THEN 'Teens'
    ELSE 'Other'
END
```

Простые выражения case менее гибки, чем поисковые, потому что вы не можете указывать собственные условия, тогда как поисковые выражения case могут включать условия диапазона, условия неравенства и многокомпонентные условия с использованием and/or/not, поэтому я рекомендую использовать поисковые выражения case везде, кроме простейших случаев.

Примеры выражений case

В следующих разделах представлены различные примеры, иллюстрирующие полезность условной логики в инструкциях SQL.

Преобразования результирующего набора

Возможно, вы столкнулись с ситуацией, когда вы выполняете агрегирование конечного набора значений, таких как дни недели, но хотите, чтобы результирующий набор содержал единственную строку со столбцами, по одному для каждого значения, а не по одной строке для каждого значения. В качестве примера предположим, что требуется написать запрос, который показывает количество прокатов фильмов в мае, июне и июле 2005 года:

```
mysql> SELECT monthname(rental_date) rental_month,
->      count(*) num_rentals
->   FROM rental
-> WHERE rental_date BETWEEN '2005-05-01' AND '2005-08-01'
-> GROUP BY monthname(rental_date);
+-----+-----+
| rental_month | num_rentals |
+-----+-----+
| May          |      1156 |
| June         |      2311 |
```

```
| July | 6709 |
+-----+
3 rows in set (0.01 sec)
```

Однако на запрос накладывается условие — он должен вернуть одну строку данных с тремя столбцами (по одному для каждого из трех месяцев). Чтобы преобразовать этот результирующий набор в единую строку, нужно создать три столбца и в каждом столбце суммировать *только* те строки, которые относятся к рассматриваемому месяцу:

```
mysql> SELECT
    -> SUM(CASE WHEN monthname(rental_date) = 'May' THEN 1
    ->           ELSE 0 END) May_rentals,
    -> SUM(CASE WHEN monthname(rental_date) = 'June' THEN 1
    ->           ELSE 0 END) June_rentals,
    -> SUM(CASE WHEN monthname(rental_date) = 'July' THEN 1
    ->           ELSE 0 END) July_rentals
    -> FROM rental
    -> WHERE rental_date BETWEEN '2005-05-01' AND '2005-08-01';
+-----+-----+-----+
| May_rentals | June_rentals | July_rentals |
+-----+-----+-----+
|      1156 |        2311 |       6709 |
+-----+-----+-----+
1 row in set (0.01 sec)
```

Все три столбца в предыдущем запросе идентичны, за исключением значения месяца. Когда функция `monthname()` возвращает требуемое значение для конкретного столбца, выражение `case` возвращает значение 1; в противном случае возвращается 0. При суммировании по всем строкам каждый столбец возвращает количество счетов, открытых в этом месяце. Очевидно, что такие преобразования применимы только для небольшого количества значений; генерация по одному столбцу для каждого года, начиная с 1905 года, быстро станет утомительной.



Хотя это и несколько выходит за рамки данной книги, стоит отметить, что и SQL Server, и Oracle Database включают предложения `pivot` специально для такого рода запросов.

Проверка существования

Иногда нужно определить, существуют ли взаимосвязи между двумя сущностями, без учета количества. Например, вы можете захотеть узнать, снялся ли некоторый актер хотя бы в одном фильме с рейтингом G, без учета фактического количества фильмов. Вот запрос для генерации трех выходных

столбцов, один из которых показывает, снимался ли актер в фильмах с рейтингом G, еще один — для фильмов с рейтингом PG и третий — для фильмов с рейтингом NC-17:

```
mysql> SELECT a.first_name, a.last_name,
-> CASE
-> WHEN EXISTS (SELECT 1 FROM film_actor fa
->                 INNER JOIN film f ON fa.film_id = f.film_id
->                 WHERE fa.actor_id = a.actor_id
->                         AND f.rating = 'G') THEN 'Y'
-> ELSE 'N'
-> END g_actor,
-> CASE
-> WHEN EXISTS (SELECT 1 FROM film_actor fa
->                 INNER JOIN film f ON fa.film_id = f.film_id
->                 WHERE fa.actor_id = a.actor_id
->                         AND f.rating = 'PG') THEN 'Y'
-> ELSE 'N'
-> END pg_actor,
-> CASE
-> WHEN EXISTS (SELECT 1 FROM film_actor fa
->                 INNER JOIN film f ON fa.film_id = f.film_id
->                 WHERE fa.actor_id = a.actor_id
->                         AND f.rating = 'NC-17') THEN 'Y'
-> ELSE 'N'
-> END nc17_actor
-> FROM actor a
-> WHERE a.last_name LIKE 'S%' OR a.first_name LIKE 'S%';
+-----+-----+-----+-----+
| first_name | last_name | g_actor | pg_actor | nc17_actor |
+-----+-----+-----+-----+
| JOE        | SWANK     | Y       | Y       | Y       |
| SANDRA    | KILMER    | Y       | Y       | Y       |
| CAMERON   | STREEP    | Y       | Y       | Y       |
| SANDRA    | PECK      | Y       | Y       | Y       |
| SISSY      | SOBIESKI  | Y       | Y       | N     |
| NICK       | STALLONE  | Y       | Y       | Y       |
| SEAN       | WILLIAMS  | Y       | Y       | Y       |
| GROUCHO   | SINATRA   | Y       | Y       | Y       |
| SCARLETT  | DAMON     | Y       | Y       | Y       |
| SPENCER   | PECK      | Y       | Y       | Y       |
| SEAN       | GUINNESS  | Y       | Y       | Y       |
| SPENCER   | DEPP      | Y       | Y       | Y       |
| SUSAN      | DAVIS     | Y       | Y       | Y       |
| SIDNEY    | CROWE    | Y       | Y       | Y       |
| SYLVESTER | DERN     | Y       | Y       | Y       |
| SUSAN      | DAVIS     | Y       | Y       | Y       |
| DAN        | STREEP    | Y       | Y       | Y       |
| SALMA     | NOLTE    | Y       | N     | Y       |
| SCARLETT  | BENING    | Y       | Y       | Y       |
```

```

| JEFF          | SILVERSTONE | Y      | Y      | Y      |
| JOHN          | SUVARI      | Y      | Y      | Y      |
| JAYNE         | SILVERSTONE | Y      | Y      | Y      |
+-----+-----+-----+-----+
22 rows in set (0.00 sec)

```

Каждое выражение case включает коррелированный подзапрос к таблицам `film_actor` и `film`; один из них ищет фильмы с рейтингом G, второй — фильмы с рейтингом PG и третий — фильмы с рейтингом NC-17. Поскольку в каждом предложении `when` используется оператор `exists`, условия вычисляются как истинные, если актер появился как минимум в одном фильме с соответствующим рейтингом.

В других случаях может быть интересно, сколько именно строк имеется, но только до определенного предела. Например, в следующем запросе простое выражение case используется для подсчета количества копий в прокате для каждого фильма, возвращая строки 'Out Of Stock' (нет в прокате), 'Scarce' (редкий), 'Available' (доступный) или 'Common' (обычный (в большом количестве)):

```

mysql> SELECT f.title,
    -> CASE (SELECT count(*) FROM inventory i
    ->           WHERE i.film_id = f.film_id)
    -> WHEN 0 THEN 'Out Of Stock'
    -> WHEN 1 THEN 'Scarce'
    -> WHEN 2 THEN 'Scarce'
    -> WHEN 3 THEN 'Available'
    -> WHEN 4 THEN 'Available'
    -> ELSE 'Common'
    -> END film_availability
    -> FROM film f
    -> ;

```

```

+-----+-----+
| title          | film_availability |
+-----+-----+
| ACADEMY DINOSAUR | Common      |
| ACE GOLDFINGER   | Available   |
| ADAPTATION HOLES | Available . |
| AFFAIR PREJUDICE | Common      |
| AFRICAN EGG       | Available   |
| AGENT TRUMAN     | Common      |
| AIRPLANE SIERRA  | Common      |
| AIRPORT POLLOCK  | Available   |
| ALABAMA DEVIL    | Common      |
| ALADDIN CALENDAR  | Common      |
| ALAMO VIDEOTAPE   | Common      |
| ALASKA PHANTOM    | Common      |
| ALI FOREVER       | Available   |

```

```

| ALICE FANTASIA          | Out Of Stock      |
| ...                      |                   |
| YOUNG LANGUAGE           | Scarce            |
| YOUTH KICK               | Scarce            |
| ZHIVAGO CORE              | Scarce            |
| ZOOLANDER FICTION         | Common             |
| ZORRO ARK                 | Common             |
+-----+
1000 rows in set (0.01 sec)

```

В этом запросе я перестал вести подсчет количества после 5, присваивая любому другому числу больше 5 метку 'Common'.

Ошибки деления на нуль

При выполнении вычислений, включающих деление, всегда следует озабочиться тем, чтобы знаменатели никогда не были равны нулю. В то время как некоторые серверы баз данных, такие как Oracle Database, сообщают об ошибке, MySQL, встретив нулевой знаменатель, просто установит результат вычисления равным null, как показано в следующем примере:

```

mysql> SELECT 100 / 0;
+-----+
| 100 / 0 |
+-----+
| NULL    |
+-----+
1 row in set (0.00 sec)

```

Чтобы защитить свои расчеты от ошибок или, что еще хуже, от таинственным образом появляющихся значений null, следует “завернуть” все знаменатели в условную логику, как демонстрируется в следующем примере:

```

mysql> SELECT c.first_name, c.last_name,
->     sum(p.amount) tot_payment_amt,
->     count(p.amount) num_payments,
->     sum(p.amount) /
->     CASE WHEN count(p.amount) = 0 THEN 1
->           ELSE count(p.amount)
->     END avg_payment
->   FROM customer c
->   LEFT OUTER JOIN payment p
->     ON c.customer_id = p.customer_id
->   GROUP BY c.first_name, c.last_name;
+-----+-----+-----+-----+
|first_name |last_name |tot_payment_amt |num_payments |avg_payment |
+-----+-----+-----+-----+
|MARY      |SMITH      |        118.68 |          32 |    3.708750 |
|PATRICIA  |JOHNSON    |        128.73 |          27 |    4.767778 |

```

LINDA	WILLIAMS		135.74		26		5.220769	
BARBARA	JONES		81.78		22		3.717273	
ELIZABETH	BROWN		144.62		38		3.805789	
...								
EDUARDO	HIATT		130.73		27		4.841852	
TERENCE	GUNDERSON		117.70		30		3.923333	
ENRIQUE	FORSYTHE		96.72		28		3.454286	
FREDDIE	DUGGAN		99.75		25		3.990000	
WADE	DELVALLE		83.78		22		3.808182	
AUSTIN	CINTRON		83.81		19		4.411053	

599 rows in set (0.07 sec)

Этот запрос вычисляет среднюю сумму платежа для каждого клиента. Поскольку некоторые клиенты могут быть новыми и еще не брали напрокат ни одного фильма, лучше включить выражение case, чтобы знаменатель никогда не был равен нулю.

Условные обновления

При обновлении строк в таблице для создания значения столбца иногда требуется условная логика. Например, предположим, что каждую неделю выполняется задание, которое устанавливает столбец customer.active равным 0 для тех клиентов, которые ни разу не брали фильм напрокат за последние 90 дней. Вот инструкция, которая устанавливает значение 0 или 1 для каждого клиента:

```
UPDATE customer
SET active =
CASE
    WHEN 90 <= (SELECT datediff(now(), max(rental_date))
                  FROM rental r
                  WHERE r.customer_id = customer.customer_id)
        THEN 0
    ELSE 1
END
WHERE active = 1;
```

Этот оператор использует коррелированный подзапрос для определения количества дней с момента последнего взятия фильма напрокат для каждого клиента, и полученное значение сравнивается со значением 90; если возвращенное подзапросом значение равно 90 или больше, клиент помечается как неактивный.

Обработка значений null

Хотя значения null являются подходящими значениями для хранения в таблице в случае, если значение столбца неизвестно, извлекать нулевые значения для отображения или использования в выражениях не всегда целесообразно. Например, пусть необходимо при выводе на экран не оставлять поле пустым, а отобразить в нем слово *Unknown* (неизвестно). При выборке данных для замены строки, если значение равно null, можно использовать выражение case наподобие следующего:

```
SELECT c.first_name, c.last_name,
CASE
    WHEN a.address IS NULL THEN 'Unknown'
    ELSE a.address
END address,
CASE
    WHEN ct.city IS NULL THEN 'Unknown'
    ELSE ct.city
END city,
CASE
    WHEN cn.country IS NULL THEN 'Unknown'
    ELSE cn.country
END country
FROM customer c
LEFT OUTER JOIN address a
    ON c.address_id = a.address_id
LEFT OUTER JOIN city ct
    ON a.city_id = ct.city_id
LEFT OUTER JOIN country cn
    ON ct.country_id = cn.country_id;
```

При вычислениях значения null часто приводят к результату null, как показано в следующем примере:

```
mysql> SELECT (7 * 5) / ((3 + 14) * null);
+-----+
| (7 * 5) / ((3 + 14) * null) |
+-----+
|                         NULL |
+-----+
1 row in set (0.08 sec)
```

При выполнении вычислений выражения case полезны для перевода значения null в число (обычно 0 или 1), что позволяет вычислениям выдавать результат, не равный null.

Проверьте свои знания

Проверьте свое умение решать задачи с использованием условной логики на приведенных здесь примерах и сравните свои решения с решениями из приложения Б.

УПРАЖНЕНИЕ 11.1

Перепишите следующий запрос, в котором используется простое выражение case, так, чтобы получить те же результаты с использованием поискового выражения case. Страйтесь, насколько это возможно, использовать как можно меньше предложений when.

```
SELECT name,
CASE name
    WHEN 'English' THEN 'latin1'
    WHEN 'Italian' THEN 'latin1'
    WHEN 'French' THEN 'latin1'
    WHEN 'German' THEN 'latin1'
    WHEN 'Japanese' THEN 'utf8'
    WHEN 'Mandarin' THEN 'utf8'
    ELSE 'Unknown'
END character_set
FROM language;
```

УПРАЖНЕНИЕ 11.2

Перепишите следующий запрос так, чтобы результирующий набор содержал одну строку с пятью столбцами (по одному для каждого рейтинга). Назовите эти пять столбцов G, PG, PG_13, R и NC_17.

```
mysql> SELECT rating, count(*)
-> FROM film
-> GROUP BY rating;
+-----+-----+
| rating | count(*) |
+-----+-----+
| PG     |      194 |
| G      |      178 |
| NC-17  |      210 |
| PG-13  |      223 |
| R      |      195 |
+-----+-----+
5 rows in set (0.00 sec)
```

Транзакции

Все рассматривавшиеся до сих пор примеры в этой книге были отдельными, независимыми инструкциями SQL. Хотя это может быть нормой для каких-то разовых отчетов или сценария сопровождения базы данных, логика приложения часто включает несколько инструкций SQL, которые необходимо выполнять вместе как единую логическую единицу. В этой главе рассматриваются *транзакции*, которые являются механизмом, используемым для группировки набора инструкций SQL вместе таким образом, что либо выполняются все они, либо ни одна из них не выполняется.

Многопользовательские базы данных

Системы управления базами данных позволяют единственному пользователю запрашивать и изменять данные, но в современном мире вносить изменения в базу данных могут одновременно тысячи людей. Если каждый пользователь только выполняет запросы, как обычно и происходит в случае с хранилищем данных в обычное рабочее время, то это для сервера базы данных практически не составляет проблем. Но если некоторые пользователи добавляют и/или изменяют данные, то серверу приходится решать намного более сложные задачи.

Предположим, например, что вы составляете отчет, в котором суммируются данные о прокате фильмов за текущую неделю. Однако одновременно с выполнением вашего отчета происходят следующие события.

- Клиент берет фильм напрокат.
- Другой клиент возвращает фильм после установленного срока возврата и платит штраф за просрочку.
- Пункт проката получает пять новых фильмов.

Таким образом, пока генерируется отчет, несколько пользователей изменяют информацию в базе данных. Так какие числа должны появиться в отчете? Ответ зависит от того, как ваш сервер обрабатывает *блокировку*, описанную в следующем разделе.

Блокировка

Блокировка — это механизм, который сервер базы данных использует для управления одновременным использованием ресурсов данных. Когда какая-то часть базы данных заблокирована, другие пользователи, желающие изменить (или, возможно, прочесть) эти данные, должны ждать, пока блокировка не будет снята. Большинство серверов баз данных используют одну из следующих двух стратегий блокировки.

- Записывающие информацию в базу данных (“писатели”) должны запрашивать и получать от сервера блокировку записи для изменения данных, а извлекающие информацию (“читатели”) из базы данных для выполнения запроса должны запрашивать и получать от сервера блокировку чтения. В то время как несколько пользователей могут читать данные одновременно, за один раз для каждой таблицы (или ее части) выдается только одна блокировка записи, а запросы на чтение блокируются до тех пор, пока блокировка записи не будет снята.
- Пищущие в базу данных пользователи должны запрашивать и получать от сервера блокировку записи для изменения данных, но читателям не нужны никакие блокировки для запроса данных. Вместо этого сервер гарантирует, что каждый читатель с момента начала его запроса видит согласованное представление данных (данные кажутся теми же самыми, даже если другие пользователи могли внести в них изменения) до тех пор, пока запрос не завершится. Этот подход известен как *управление версиями* (*versioning*).

У обоих подходов есть свои плюсы и минусы. Первый подход может привести к долгому времени ожидания при наличии большого количества одновременных запросов на чтение и запись. Второй подход может быть проблематичным, если во время изменения данных поступают длительные запросы. Из трех обсуждаемых в этой книге серверов Microsoft SQL Server использует первый подход, Oracle Database использует второй подход, а MySQL использует оба подхода (в зависимости от вашего выбора *механизма хранения*, который мы обсудим далее в этой главе).

Гранулярность блокировок

Существует ряд различных стратегий, которые вы можете использовать при принятии решения о том, как именно блокировать ресурс. Сервер может применять блокировку на одном из трех разных уровней, или *гранулярностей*.

Блокировка таблиц

Не позволяет нескольким пользователям одновременно изменять данные в одной таблице.

Блокировка страниц

Не позволяет некоторым пользователям изменять данные в одной и той же странице (страница — это сегмент памяти, обычно в диапазоне от 2 до 16 Кбайт) таблицы одновременно.

Блокировка строк

Не позволяет некоторым пользователям одновременно изменять одну и ту же строку в таблице.

У этих подходов есть свои плюсы и минусы. Чтобы блокировать целые таблицы, требуется совсем немного времени, но этот подход по мере увеличения числа пользователей быстро приводит к неприемлемому времени ожидания. Блокировка строк требует большего объема дополнительных действий, но зато позволяет некоторым пользователям изменять одну и ту же таблицу, если они работают с разными строками. Из трех серверов, рассматриваемых в этой книге, Microsoft SQL Server использует блокировку страниц, строк и таблиц, Oracle Database использует только блокировку строк, а MySQL использует блокировку таблиц, страниц или строк (в зависимости от вашего выбора механизма хранения). SQL Server при определенных обстоятельствах наращивает блокировки от строки к странице и от страницы к таблице, тогда как Oracle Database никогда не увеличивает гранулярность блокировок.

Возвращаясь к нашему отчету — данные, которые появляются на страницах отчета, будут отражать либо состояние базы данных при запуске вашего отчета (если сервер использует подход с управлением версиями), либо состояние базы данных, когда сервер выполняет для приложения, генерирующего отчет, блокировку чтения (если ваш сервер использует блокировки чтения и записи).

Что такое транзакция

Если бы серверы баз данных безотказно работали все 100% времени, если бы пользователи всегда позволяли программам завершать выполнение и если бы приложения всегда завершались без фатальных ошибок, останавливающих выполнение, не было бы ничего, что следовало было бы обсуждать об одновременном доступе к базе данных. Однако мы не можем полагаться ни на что из перечисленного, поэтому необходимо поднять еще один вопрос, позволяющий некоторым пользователям получать доступ к одним и тем же данным.

Этой дополнительной частью головоломки параллелизма является *транзакция*, которая представляет собой механизм группировки нескольких инструкций SQL, так что либо все инструкции выполняются успешно, либо не выполняется ни одна из них (свойство “все или ничего”, известное как *атомарность*). Если вы попытаетесь перевести 500 долларов со своего сберегательного счета на текущий счет, вероятно, вы будете несколько расстроены, если деньги будут успешно сняты со сберегательного счета, но так и не поступят на текущий счет. Какой бы ни была причина сбоя (сервер был отключен на техническое обслуживание, истекло время запроса блокировки страницы в таблице учетных записей и т.д.), вы захотите вернуть свои 500 долларов.

Для защиты от такого рода ошибок программа, обрабатывающая ваш запрос на перевод денег, сначала начинает транзакцию, затем выдает SQL-запросы, необходимые для перевода денег с одного счета на другой, и, если все проходит успешно, завершает транзакцию, выполнив команду *commit* — фиксации изменений. Однако, если произойдет что-то непредвиденное, программа выдаст команду отката *rollback*, которая потребует от сервера отменить все изменения, сделанные с момента начала транзакции. Весь процесс может выглядеть примерно так:

```
START TRANSACTION;

/* Снятие денег с первого счета с проверкой
   наличия достаточной суммы на счету */
UPDATE account SET avail_balance = avail_balance - 500
WHERE account_id = 9988
AND avail_balance > 500;

IF <предыдущей инструкцией обновлена ровно одна строка> THEN
    /* Помещение денег на второй счет */
    UPDATE account SET avail_balance = avail_balance + 500
    WHERE account_id = 9989;
```

```
IF <предыдущей инструкцией обновлена ровно одна строка> THEN
    /* Все отработано успешно; делаем изменения постоянными */
    COMMIT;
ELSE
    /* Что-то пошло не так; откат изменений */
    ROLLBACK;
END IF;
ELSE
    /* Недостаточно денег или ошибка при внесении изменений */
    ROLLBACK;
END IF;
```



Хотя предыдущий блок кода и может показаться похожим на один из процедурных языков, предоставляемых крупными компаниями баз данных, такой как Oracle PL/SQL или Microsoft Transact-SQL, он написан на псевдокоде и не пытается имитировать какой-либо конкретный язык.

Показанный блок кода начинается с запуска транзакции, а затем пытается снять 500 долларов с одного счета и добавить их на другой. Если все пройдет хорошо, транзакция будет выполнена; если же что-то пойдет не так, произойдет откат транзакции, т.е. будут отменены все изменения данных с начала транзакции.

Используя транзакцию, программа гарантирует, что ваши 500 долларов останутся на первом счету или будут перенесены на второй счет без возможности их потери. Независимо от того, была ли транзакция зафиксирована или был выполнен откат, все полученные во время выполнения транзакции ресурсы (например, блокировки записи) высвобождаются после завершения транзакции.

Конечно, если программе удастся выполнить обе инструкции `update`, но сервер остановится до того, как будет выполнена фиксация или откат, то, когда работоспособность сервера будет восстановлена, будет выполнен откат. (Одна из задач, которые сервер базы данных должен выполнить перед подключением к сети, — найти все незавершенные транзакции, которые обрабатывались при отключении сервера, и откатить их.) Кроме того, если ваша программа завершает транзакцию и выполняет `commit`, но сервер выключается до того, как изменения будут внесены в постоянное хранилище (т.е. измененные данные все еще находятся в памяти и не сброшены на диск), то сервер базы данных должен повторно применить к данным изменения из вашей транзакции при перезапуске сервера (свойство, известное как устойчивость (*durability*)).

Запуск транзакции

Серверы баз данных обрабатывают создание транзакций одним из двух способов.

- Активная транзакция всегда связана с сеансом базы данных, поэтому нет необходимости в явном начале транзакции и нет никакого метода для этого. Когда текущая транзакция заканчивается, сервер автоматически начинает новую транзакцию для вашего сеанса.
- Если вы не начинаете транзакцию явно, отдельные инструкции SQL автоматически считаются совершенно независимыми одна от другой. Чтобы начать транзакцию, вы должны сначала выполнить соответствующую команду.

Из трех серверов Oracle Database использует первый подход, а Microsoft SQL Server и MySQL — второй подход. Одно из преимуществ подхода Oracle к транзакциям заключается в том, что даже если вы вводите только одну команду SQL, у вас есть возможность отменить изменения, если вам не понравился результат или если вы передумали. Таким образом, если вы забыли добавить предложение `where` в инструкцию `delete`, у вас будет возможность исправить содеянное (при условии, что вы вовремя сообразите, что удалить все 125 000 строк из таблицы не входило в ваши намерения). Однако в MySQL и SQL Server изменения, вызванные вашей инструкцией SQL после нажатия клавиши `<Enter>`, будут постоянными (если только ваш администратор баз данных может не получить исходные данные из резервной копии или каким-либо иным способом).

Стандарт SQL:2003 включает команду `start transaction`, которая используется, когда вы хотите явно начать транзакцию. В то время как MySQL соответствует стандарту, пользователи SQL Server должны ввести команду `begin transaction`. В обоих серверах, пока вы явно не начнете транзакцию, вы находитесь в так называемом *режиме автоматической фиксации (автофиксации)*, что означает, что результаты выполнения отдельных инструкций автоматически фиксируются сервером. Таким образом, вы можете принять решение о выполнении транзакции и выполнить команду ее запуска или просто позволить серверу фиксировать результаты выполнения отдельных инструкций.

И MySQL, и SQL Server позволяют отключать режим автоматической фиксации для отдельных сеансов работы, и в этом случае серверы будут действовать в отношении транзакций так же, как и сервер Oracle Database. Чтобы

отключить режим автоматической фиксации, в SQL Server вы вводите следующую команду:

```
SET IMPLICIT_TRANSACTIONS ON
```

MySQL позволяет отключить режим автоматической фиксации следующим образом:

```
SET AUTOCOMMIT=0
```

После выхода из режима автоматической фиксации все команды SQL выполняются в пределах области видимости транзакции и должны быть явно зафиксированы или отменены.



Небольшой совет: отключайте режим автоматической фиксации каждый раз при входе в систему и приобретите привычку выполнять все операторы SQL в транзакциях. Это по крайней мере может избавить вас от смущения при необходимости просить администратора базы данных восстановить данные, которые вы по неосторожности удалили.

Завершение транзакции

После того как транзакция началась (явно ли с помощью команды `start transaction` или неявно сервером базы данных), вы должны явно ее завершить, чтобы внесенные вами изменения стали постоянными. Это делается с помощью команды `commit`, которая дает указание серверу пометить изменения как постоянные и освободить все ресурсы (т.е. блокировки страниц или строк), использовавшиеся во время транзакции.

Если вы решите отменить все изменения, сделанные с момента начала транзакции, вы должны ввести команду `rollback`, которая требует от сервера вернуть данные в состояние до начала транзакции. После завершения отката любые ресурсы, используемые сеансом, освобождаются.

Наряду с выполнением команды `commit` или `rollback` имеется несколько других сценариев того, как может завершиться транзакция — как косвенный результат ваших действий или в результате чего-то, находящегося вне вашего контроля.

- Сервер выключается, и в этом случае ваша транзакция будет автоматически отменена при перезапуске сервера.
- Вы вводите инструкцию схемы SQL, такую как `alter table`, которая приводит к тому, что текущая транзакция должна быть зафиксирована, а новая транзакция должна быть запущена.

- Вы выполняете еще одну команду `start transaction`, которая вызовет завершение предыдущей транзакции, которая должна быть фиксирована.
- Сервер преждевременно завершает вашу транзакцию, поскольку обнаруживает взаимоблокировку и приходит к выводу, что в ней виновата ваша транзакция. В этом случае будет выполнен откат транзакции, и вы получите сообщение об ошибке.

Из этих четырех сценариев первый и третий достаточно просты, но другие два заслуживают дополнительного рассмотрения. Что касается второго сценария, то изменения базы данных, будь то добавление новой таблицы или индекса или удаление столбца из таблицы, не может быть отменено, поэтому команды, изменяющие вашу схему, должны происходить вне транзакции. Следовательно, если в настоящее время выполняется транзакция, сервер зафиксирует текущую транзакцию, выполнит команды инструкции схемы SQL, а затем автоматически начнет новую транзакцию для вашего сеанса. Сервер не сообщит вам о том, что произошло, поэтому вы должны быть осторожны, чтобы инструкции, составляющие единую логическую единицу, не были случайно разбиты сервером на несколько транзакций.

Четвертый сценарий связан с обнаружением взаимоблокировок. Взаимоблокировка возникает, когда две разные транзакции ждут ресурсов, которые в настоящее время захвачены другой транзакцией. Например, транзакция A могла только что обновить таблицу `account` и ожидает блокировки записи в таблице `transaction`, в то время как транзакция B вставила запись в таблицу `transaction` и ожидает блокировки записи в таблице `account`. Если обе транзакции изменяют одну и ту же страницу или строку (в зависимости от гранулярности блокировок, используемой сервером базы данных), то каждая из них будет вечно ждать завершения другой транзакции и освобождения необходимого ресурса. Серверы баз данных должны всегда отслеживать такие ситуации, чтобы не страдала пропускная способность; при обнаружении взаимоблокировки одна из транзакций (выбранная произвольно или по некоторым критериям) откатывается, чтобы можно было продолжить другую транзакцию. В большинстве случаев прекращенная транзакция может быть перезапущена вновь и будет успешной без возникновения новой взаимоблокировки.

В отличие от рассматривавшегося ранее второго сценария, сервер базы данных генерирует сообщение об ошибке с информацией об отмене транзакции

из-за обнаружения взаимоблокировки. Например, в случае MySQL вы получите ошибку 1213, которая содержит следующее сообщение:

Message: Deadlock found when trying to get lock; try restarting transaction¹

Как следует из сообщения об ошибке, разумно повторить транзакцию, которая был откачена из-за обнаружения взаимоблокировки. Однако, если взаимоблокировки будут распространенной ситуацией, то может потребоваться изменить приложения, которые обращаются к базе данных, чтобы уменьшить вероятность таких взаимоблокировок (одна из распространенных стратегий — гарантировать доступ к ресурсам всегда в одном и том же порядке, например всегда изменять данные учетной записи перед вставкой данных транзакции).

Точки сохранения транзакции

В некоторых случаях вы можете столкнуться в транзакции с проблемой, требующей отката, но при этом не захотеть отменять всю проделанную работу. Для таких ситуаций можно установить одну или несколько *точек сохранения* в транзакции и использовать их, чтобы выполнить откат к определенному месту вашей транзакции, а не к ее началу.

Выбор механизма хранения

При использовании Oracle Database или Microsoft SQL Server за низкоуровневые операции с базой данных (такие, как получение определенной строки из таблицы на основе значения первичного ключа) отвечает единый набор кода. Однако сервер MySQL спроектирован таким образом, чтобы для обеспечения низкоуровневой функциональности базы данных, включая блокировку ресурсов и управление транзакциями, могло использоваться несколько механизмов хранения. Начиная с версии 8.0, MySQL включает следующие механизмы хранения.

MyISAM

Нетранзакционный механизм, использующий табличную блокировку

MEMORY

Нетранзакционный механизм, используемый для таблиц в памяти

CSV

Транзакционный механизм, который хранит данные в файлах с данными с разделением запятыми

¹ Сообщение: при попытке блокировки обнаружена взаимоблокировка. Попытайтесь повторить транзакцию.

InnoDB

Транзакционный механизм, использующий блокировку на уровне строки

Merge

Специальный механизм, используемый для создания нескольких идентичных таблиц MyISAM в виде единой таблицы (так называемое разбиение таблицы)

Archive

Специальный механизм, используемый для хранения больших количеств неиндексированных данных, в основном для архивных целей

Хотя вы можете подумать, что требуется выбрать единственный механизм хранения для вашей базы данных, MySQL достаточно гибок, чтобы позволить вам выбирать механизмы хранения на основе таблиц. Однако для любых таблиц, которые могут участвовать в транзакциях, следует выбирать механизм InnoDB, который использует блокировку уровня строки и управление версиями для обеспечения наивысшего уровня параллелизма среди всех различных механизмов хранения.

Вы можете явно указать механизм хранения при создании таблицы, а также изменить существующую таблицу, указав для ее хранения другой механизм. Если вы не знаете, какой механизм хранения назначен той или иной таблице, можете использовать команду `show table`, как показано далее:

```
mysql> show table status like 'customer' \G;
***** 1. row ****
      Name: customer
      Engine: InnoDB
     Version: 10
   Row_format: Dynamic
       Rows: 599
 Avg_row_length: 136
  Data_length: 81920
Max_data_length: 0
 Index_length: 49152
  Data_free: 0
Auto_increment: 599
 Create_time: 2019-03-12 14:24:46
 Update_time: NULL
 Check_time: NULL
  Collation: utf8_general_ci
    Checksum: NULL
Create_options:
      Comment:
1 row in set (0.16 sec)
```

Взглянув на вторую строку, вы видите, что таблица `customer` уже использует механизм InnoDB. Если бы это было не так, вы могли бы назначить механизм InnoDB с помощью следующей команды:

```
ALTER TABLE customer ENGINE = INNODB;
```

Всем точкам сохранения должны быть даны имена, которые позволяют иметь несколько различных точек сохранения в пределах одной транзакции. Чтобы создать точку сохранения с именем `my_savepoint`, выполните следующую инструкцию:

```
SAVEPOINT my_savepoint;
```

Для отката к определенной точке сохранения просто вводится команда `rollback`, за которой следуют ключевые слова `to savepoint` и имя точки сохранения, например:

```
ROLLBACK TO SAVEPOINT my_savepoint;
```

Вот пример того, как можно использовать точки 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;
```

В результате этой транзакции мифический товар XYZ снят с производства, но ни один счет не закрыт.

При использовании точек сохранения помните следующее.

- Несмотря на название, при создании точки сохранения ничего не сохраняется. В конечном итоге вы должны выполнить `commit`, если хотите, чтобы ваша транзакция стала постоянной.
- Если вы выполняете откат без указания точки сохранения, все точки сохранения в пределах транзакции игнорируются и выполняется откат всей транзакции в целом.

Если вы используете SQL Server, для создания точки сохранения нужно использовать его нестандартную команду `save transaction`, а для отката к точке сохранения — `rollback transaction`. Каждая команда сопровождается именем точки сохранения.

Проверьте свои знания

Предложенное здесь упражнение призвано закрепить понимание вами транзакций. Ответ к упражнению представлен в приложении Б.

УПРАЖНЕНИЕ 12.1

Создайте логическую единицу работы для перевода 50 долларов со счета 123 на счет 789. Для этого вставьте две строки в таблицу `transaction` и обновите две строки в таблице `account`. Используйте следующие определения/данные таблиц:

Account:

account_id	avail_balance	last_activity_date
123	500	2019-07-10 20:53:27
789	75	2019-06-22 15:18:35

Transaction:

txn_id	txn_date	account_id	txn_type_cd	amount
1001	2019-05-15	123	C	500
1002	2019-06-01	789	C	75

Используйте `txn_type_cd = 'C'`, чтобы указать операцию кредита (добавление на счет), и `txn_type_cd = 'D'` для обозначения дебета (снятия со счета).

Индексы и ограничения

Поскольку основное внимание в этой книге уделяется методам программирования, предыдущие главы сосредоточены на элементах языка SQL, которые можно использовать для создания мощных инструкций `select`, `insert`, `update` и `delete`. Однако другие возможности баз данных также косвенно влияют на код, который вы пишете. В этой главе рассматриваются две из этих возможностей: индексы и ограничения.

Индексы

Когда вы вставляете в таблицу строку, сервер базы данных не пытается поместить данные в определенное конкретное место в таблице. Например, если вы добавляете строку в таблицу `customer`, сервер не размещает строки в числовом порядке значений столбца `customer_id` или в алфавитном порядке значений столбца `last_name`. Вместо этого он просто помещает данные в следующее доступное место в файле (сервер поддерживает список свободных мест для каждой таблицы). Когда вы запрашиваете таблицу `customer`, серверу приходится проверять каждую строку таблицы, чтобы ответить на ваш запрос. Например, предположим, что вы вводите следующий запрос:

```
mysql> SELECT first_name, last_name
-> FROM customer
-> WHERE last_name LIKE 'Y%';
+-----+-----+
| first_name | last_name |
+-----+-----+
| LUIS       | YANEZ      |
| MARVIN     | YEE        |
| CYNTHIA    | YOUNG      |
+-----+-----+
3 rows in set (0.09 sec)
```

Чтобы найти всех клиентов, чьи фамилии начинаются с Y, сервер должен посетить каждую строку в таблице `customer` и проверить содержимое

столбца `last_name`; если фамилия начинается с Y, то строка добавляется к результирующему набору. Этот тип доступа известен как *сканирование таблицы*.

Хотя этот метод неплохо работает для таблицы только с тремя строками, представьте, сколько времени может потребоваться для ответа на запрос, если таблица содержит три миллиона строк. При некотором количестве строк, большем, чем три, и меньшем трех миллионов, находится тот размер таблицы, для которого сервер не в состоянии ответить на запрос за разумное время без дополнительной помощи. Эта помощь приходит в виде одного или нескольких *индексов* для таблицы `customer`.

Даже если вы никогда не слышали об индексах баз данных, вы, безусловно, знаете, что индекс — это средство быстрого доступа (например, в данной книге имеется индекс — предметный указатель). Индекс — это просто механизм поиска конкретного объекта внутри ресурса. Таковым индексом является предметный указатель, который позволяет быстро находить определенное слово в книге, располагая термины в алфавитном порядке. Это позволяет читателю сначала быстро найти конкретную букву в пределах предметного указателя, а затем найти нужную запись и страницу или страницы, на которых можно найти искомое слово.

Таким же образом, как человек использует предметный указатель, чтобы найти нужные слова в книге, сервер базы данных использует индексы для определения местонахождения строк в таблице. Индексы — это специальные таблицы, которые, в отличие от обычных таблиц данных, хранятся в *определенном порядке*. Но вместо того, чтобы содержать все данные о некоторой записи, индекс содержит только столбец (или столбцы), используемый, чтобы найти строки в таблице данных, вместе с информацией, описывающей, где физически расположена эта строка. Таким образом, роль индексов состоит в том, чтобы облегчить поиск подмножества строк и столбцов таблицы без необходимости сканировать каждую строку в таблице.

Создание индекса

Возвращаясь к таблице `customer`, вы можете принять решение добавить индекс к столбцу `email`, чтобы ускорить любые запросы, которые работают со значением этого столбца, а также любые операции `update` или `delete`, которые указывают адрес электронной почты клиента. Вот как можно добавить такой индекс к базе данных MySQL:

```
mysql> ALTER TABLE customer  
-> ADD INDEX idx_email (email);
```

```
Query OK, 0 rows affected (1.87 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

Эта инструкция создает индекс (точнее, индекс В-дерева, но подробнее об этом речь пойдет ниже) для столбца `customer.email`; этот индекс получает имя `idx_email`. При наличии индекса оптимизатор запросов (который мы упоминали в главе 3, “Запросы”) может выбрать использование индекса, если сочтет это полезным. Если в таблице имеется более одного индекса, оптимизатор должен решить, применение какого именно индекса наиболее выгодно для конкретной инструкции SQL.



MySQL рассматривает индексы как необязательные компоненты таблицы. Вот почему в более ранних версиях для добавления или удаления индекса вы использовали бы команду `alter table`. Другие серверы баз данных, включая SQL Server и Oracle Database, рассматривают индексы как независимые объекты схемы. Таким образом, как для SQL Server, так и для Oracle индекс генерируется с помощью команды `create index`:

```
CREATE INDEX idx_email
ON customer (email);
```

Начиная с MySQL версии 5 доступна команда `create index`, хотя она и отображается в соответствующую команду `alter table`. Однако для создания индексов первичного ключа все равно требуется использовать команду `alter table`.

Все серверы баз данных позволяют просматривать доступные индексы. Пользователи MySQL могут использовать команду `show`, чтобы увидеть все индексы в определенной таблице, как в следующем примере:

```
mysql> SHOW INDEX FROM customer \G;
***** 1. row *****
      Table: customer
Non_unique: 0
      Key_name: PRIMARY
Seq_in_index: 1
Column_name: customer_id
      Collation: A
Cardinality: 599
     Sub_part: NULL
       Packed: NULL
      Null:
Index_type: BTREE
...
***** 2. row *****
```

```
        Table: customer
        Non_unique: 1
          Key_name: idx_fk_store_id
Seq_in_index: 1
Column_name: store_id
Collation: A
Cardinality: 2
Sub_part: NULL
Packed: NULL
Null:
Index_type: BTREE
...
***** 3. row *****
        Table: customer
        Non_unique: 1
          Key_name: idx_fk_address_id
Seq_in_index: 1
Column_name: address_id
Collation: A
Cardinality: 599
Sub_part: NULL
Packed: NULL
Null:
Index_type: BTREE
...
***** 4. row *****
        Table: customer
        Non_unique: 1
          Key_name: idx_last_name
Seq_in_index: 1
Column_name: last_name
Collation: A
Cardinality: 599
Sub_part: NULL
Packed: NULL
Null:
Index_type: BTREE
...
***** 5. row *****
        Table: customer
        Non_unique: 1
          Key_name: idx_email
Seq_in_index: 1
Column_name: email
Collation: A
Cardinality: 599
Sub_part: NULL
Packed: NULL
Null: YES
Index_type: BTREE
...
5 rows in set (0.06 sec)
```

Вывод демонстрирует, что в таблице customer есть пять индексов: один — для столбца customer_id с именем PRIMARY и четыре других — для столбцов store_id, address_id, last_name и email. Если вам интересно, откуда взялись эти индексы, то это я создал индекс для столбца email, а остальные были установлены как часть образца базы данных Sakila. Вот инструкция, используемая для создания таблицы:

```
CREATE TABLE customer (
    customer_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
    store_id TINYINT UNSIGNED NOT NULL,
    first_name VARCHAR(45) NOT NULL,
    last_name VARCHAR(45) NOT NULL,
    email VARCHAR(50) DEFAULT NULL,
    address_id SMALLINT UNSIGNED NOT NULL,
    active BOOLEAN NOT NULL DEFAULT TRUE,
    create_date DATETIME NOT NULL,
    last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (customer_id),
    KEY idx_fk_store_id (store_id),
    KEY idx_fk_address_id (address_id),
    KEY idx_last_name (last_name),
    ...
)
```

Когда таблица была создана, сервер MySQL автоматически сгенерировал индекс для столбца первичного ключа, которым в данном случае является customer_id, и присвоил индексу имя PRIMARY. Это особый тип индекса, используемый с ограничением первичного ключа, но об ограничениях я расскажу позже в этой главе.

Если после создания индекса вы решите, что индекс бесполезен, можете удалить его следующим образом:

```
mysql> ALTER TABLE customer
      -> DROP INDEX idx_email;
Query OK, 0 rows affected (0.50 sec)
Records: 0 Duplicates: 0 Warnings: 0
```



Пользователи SQL Server и Oracle Database должны использовать для удаления индекса команду drop index:

```
DROP INDEX idx_email;                                (Oracle)
DROP INDEX idx_email ON customer;                   (SQL Server)
```

MySQL теперь поддерживает и команду drop index, хотя она так же, как и команда create index, отображается в команду alter table.

Уникальные индексы

При разработке базы данных важно учитывать, каким столбцам разрешено содержать повторяющиеся данные, а каким — нет. Например, в таблице `customer` допустимо наличие двух клиентов с именем John Smith, поскольку в каждой строке будут свои идентификатор (`customer_id`), электронная почта и адрес, чтобы их можно было различить. Вы, однако, не хотите позволять двум разным клиентам иметь один и тот же адрес электронной почты. Этого можно добиться, создав **уникальный индекс** для столбца `customer.email`.

Уникальный индекс играет несколько ролей; наряду с предоставлением всех преимуществ обычного индекса он также служит механизмом для запрета повторяющихся значений в индексируемом столбце. При вставке строки или изменении индексированного столбца сервер базы данных проверяет уникальный индекс, чтобы узнать, не существует ли уже такое значение в другой строке таблицы. Вот как можно создать уникальный индекс для столбца `customer.email`:

```
mysql> ALTER TABLE customer
    -> ADD UNIQUE idx_email (email);
Query OK, 0 rows affected (0.64 sec)
Records: 0 Duplicates: 0 Warnings: 0
```



Пользователям SQL Server и Oracle Database нужно только добавить ключевое слово `unique` при создании индекса, например:

```
CREATE UNIQUE INDEX idx_email
ON customer (email);
```

При наличии индекса вы получите сообщение об ошибке, если попытаетесь добавить нового клиента с уже существующим адресом электронной почты:

```
mysql> INSERT INTO customer
    -> (store_id, first_name, last_name, email, address_id, active)
    -> VALUES
    -> (1, 'ALAN', 'KAHN', 'ALAN.KAHN@sakilacustomer.org', 394, 1);
ERROR 1062 (23000): Duplicate entry 'ALAN.KAHN@sakilacustomer.org'
for key 'idx_email'1
```

Вы не должны создавать уникальные индексы для столбца (столбцов) первичного ключа, поскольку сервер и без того проверяет уникальность значений первичного ключа. Однако вы можете создать более чем один уникальный индекс в одной и той же таблице, если считаете, что это оправдано.

¹ Дубликат значения 'ALAN.KAHN@sakilacustomer.org' для ключа 'idx_email'.

Многостолбцовые индексы

Наряду с рассматривавшимися до сих пор одностолбцовыми индексами можно строить индексы, охватывающие несколько столбцов. Если, например, вы ищете клиентов по имени и фамилии, то можете построить индекс по обоим столбцам вместе:

```
mysql> ALTER TABLE customer
-> ADD INDEX idx_full_name (last_name, first_name);
Query OK, 0 rows affected (0.35 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

Этот индекс будет полезен для запросов, в которых указываются имя и фамилия, или просто фамилия, но будет бесполезен для запросов, в которых указывается только имя клиента. Чтобы понять, почему это так, подумайте, как бы вы могли найти телефонный номер человека. Если вы знаете имя и фамилию человека, вы можете использовать телефонную книгу, чтобы быстро найти номер, так как телефонная книга упорядочена сначала по фамилии, а затем — по имени. Если же вы знаете только имя человека, вам нужно будет сканировать каждую запись в телефонной книге, чтобы найти все записи с указанным именем.

Поэтому при построении многостолбцовых индексов следует тщательно подумать о том, какой столбец указать первым, какой — вторым и так далее, чтобы сделать индекс максимально полезным. Однако имейте в виду, что, если это необходимо для обеспечения адекватного времени отклика, вас ничто не останавливает от построения нескольких индексов с использованием одного и того же набора столбцов, но в другом порядке.

Типы индексов

Индексирование — мощный инструмент, но, поскольку существует много разных типов данных, имеется не одна стратегия индексирования. В следующих разделах показаны различные типы индексирования, доступные у различных серверов.

Индексы B-tree

Все показанные до сих пор индексы представляют собой *индексы на основе сбалансированного дерева*, которые чаще известны как *индексы B-tree*. MySQL, Oracle Database и SQL Server по умолчанию используют B-tree индексирование, поэтому, если вы явно не запросите другой тип, то получите индекс B-tree. Как и следовало ожидать, индексы B-tree организованы в виде деревьев с одним или несколькими уровнями узлов ветвления, ведущих

к одному уровню листовых узлов. Узлы ветвления используются для навигации по дереву, а листья содержат фактические значения и информацию о местоположении. Например, индекс B-tree, построенный для столбца `customer.last_name`, может выглядеть так, как показано на рис. 13.1.

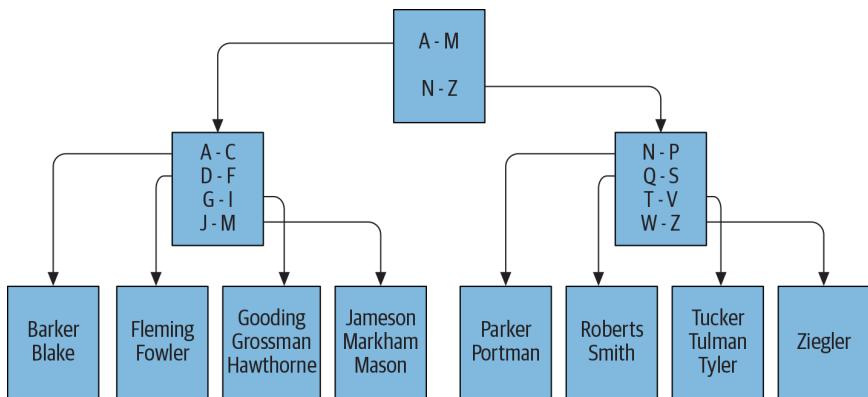


Рис. 13.1. Пример сбалансированного дерева B-tree

Если выдан запрос для извлечения всех клиентов, фамилии которых начинается с буквы G, сервер просматривает верхний узел ветвления (именуемый **корневым узлом**) и следует по ссылке к узлу ветвления, который обрабатывает фамилии, начинающиеся с букв от A до M. Этот узел, в свою очередь, направляет сервер к листовому узлу, содержащему фамилии, начинающиеся с букв от G до I. После этого сервер начинает чтение значений в листе, пока не столкнется со значением, которое уже не начинается с буквы G (в данном случае это фамилия Hawthorne).

При вставке, обновлении и удалении строк из таблицы `customer` сервер попытается сохранить дерево сбалансированным, чтобы с одной стороны корневого узла не оказалось узлов и листьев существенно больше, чем с другой. Сервер может добавлять или удалять узлы ветвления для более равномерного распределения значений и даже добавлять или удалять некоторые уровни узлов ветвления полностью. Поддерживая дерево сбалансированным, сервер может быстро добираться до листов при поиске требуемых значений без необходимости перемещения по многим уровням узлов ветвления.

Битовые индексы

Хотя индексы B-tree великолепны при обработке столбцов, которые содержат много разных значений, таких как имя/фамилия клиента, они могут стать слишком громоздкими при построении для столбца, который допускает

только небольшое количество значений. Например, вы можете решить сгенерировать индекс для столбца `customer.active`, чтобы быстро получать все активные или неактивные учетные записи. Но поскольку в этом столбце есть только два разных значения (которые хранятся как 1 для активных и 0 для неактивных клиентов), а также поскольку активных клиентов гораздо больше, чем неактивных, может оказаться трудно поддерживать сбалансированность деревьев при увеличении численности клиентов.

Для столбцов, которые содержат только небольшое количество значений при большом количестве строк (известные как *данные с низкой мощностью* (*кардинальностью*)), необходима другая стратегия индексации. Для более эффективной обработки такой ситуации Oracle Database включает *битовые индексы*, или *bitmap-индексы* (*bitmap indexes*), которые генерируют битовое представление для каждого значения, сохраненного в столбце. Если построить такой индекс для столбца `customer.active`, он будет поддерживать две битовые карты: одну — для значения 0, а другую — для значения 1. Когда вы пишете запрос для извлечения всех неактивных клиентов, сервер базы данных может обойти битовую карту для значения 0 и быстро извлечь нужные строки.

Битовые индексы — хорошее, компактное решение для индексирования для данных с низкой мощностью, но эта стратегия не будет эффективно работать, если количество значений, хранящихся в столбце, станет слишком большим по отношению к количеству строк (*данные с высокой мощностью*), поскольку сервер должен будет поддерживать слишком много битовых карт. Например, никогда не следует создавать *bitmap-индекс* для столбца первичного ключа, поскольку он обеспечивает максимальную возможную мощность (различные значения для каждой строки).

Пользователи Oracle могут генерировать битовые индексы, просто добавив ключевое слово `bitmap` в инструкцию `create index`:

```
CREATE BITMAP INDEX idx_active ON customer (active);
```

Битовые индексы обычно используются в средах хранилищ данных, где большие объемы данных обычно индексируются по столбцам, содержащим относительно небольшое количество значений (например, продажи по кварталам, географические регионы, товары, продавцы).

Текстовые индексы

Если ваша база данных хранит документы, вам может потребоваться разрешить пользователям искать слова или фразы в документах. Конечно,

нежелательно, чтобы сервер просматривал каждый документ и сканировал текст каждый раз, когда запрашивается поиск, но и традиционные стратегии индексирования в этой ситуации не работают. Чтобы справиться с такой ситуацией, MySQL, SQL Server и Oracle Database включают специальные механизмы индексации и поиска для документов; и SQL Server, и MySQL включают так называемые *полнотекстовые индексы*, а Oracle Database включает мощный набор инструментов, известный как *Oracle Text*. Поиск документов достаточно специализирован, поэтому показывать конкретный пример не-практически, но полезно хотя бы знать, что такое индексирование доступно.

Как используются индексы

Индексы обычно используются сервером для быстрого поиска строк в определенной таблице, после чего сервер посещает связанную таблицу для извлечения дополнительной информации, запрошенной пользователем. Рассмотрим следующий запрос:

```
mysql> SELECT customer_id, first_name, last_name
    -> FROM customer
    -> WHERE first_name LIKE 'S%' AND last_name LIKE 'P%';
+-----+-----+-----+
| customer_id | first_name | last_name |
+-----+-----+-----+
|      84 | SARA       | PERRY      |
|     197 | SUE        | PETERS     |
|    167 | SALLY      | PIERCE    |
+-----+-----+-----+
3 rows in set (0.00 sec)
```

Для этого запроса сервер может использовать любую из следующих стратегий.

- Сканировать все строки в таблице клиентов.
- Использовать индекс в столбце `last_name`, чтобы найти всех клиентов, чьи фамилии начинаются с `P`; затем посетить каждую строку таблицы клиентов, чтобы найти только те строки, имя которых начинается с `S`.
- Использовать индекс для столбцов `last_name` и `first_name`, чтобы найти всех клиентов, чьи фамилии начинаются с `P`, а имя — с `S`.

Третий вариант кажется наилучшим, поскольку индекс вернет все строки, необходимые для результирующего набора, без необходимости повторного посещения таблицы. Но откуда вы знаете, какой из трех вариантов будет использован? Чтобы узнать, как именно оптимизатор запросов MySQL решает

выполнить запрос, я использую инструкцию explain, которая просит сервер показать план выполнения для запроса, а не выполнить его:

```
mysql> EXPLAIN
-> SELECT customer_id, first_name, last_name
-> FROM customer
-> WHERE first_name LIKE 'S%' AND last_name LIKE 'P%' \G;
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: customer
    partitions: NULL
        type: range
possible_keys: idx_last_name,idx_full_name
           key: idx_full_name
      key_len: 274
         ref: NULL
        rows: 28
  filtered: 11.11
       Extra: Using where; Using index
1 row in set, 1 warning (0.00 sec)
```



Каждый сервер базы данных включает инструменты, позволяющие увидеть, как оптимизатор запросов обрабатывает ваш оператор SQL. SQL Server позволяет увидеть план выполнения с помощью команды `set show plan_text on` перед выполнением вашей инструкции SQL. Oracle Database включает инструкцию `explain plan`, которая записывает план выполнения в специальную таблицу с именем `plan_table`.

Взглянув на результаты запроса, мы увидим, что столбец `possible_keys` говорит о том, что сервер может решить использовать индекс `idx_last_name` или `idx_full_name`, а столбец `key` сообщает вам, что был выбран индекс `idx_full_name`. Кроме того, столбец `type` сообщает, что будет использоваться сканирование диапазона, т.е. что сервер базы данных будет искать диапазон значений в индексе, а не получать единственную строку.



Такой процесс анализа является примером настройки запроса. Настройка включает просмотр инструкции SQL и определение ресурсов, доступных серверу для ее выполнения. Вы можете решить изменить инструкцию SQL, подстроиться под ресурсы базы данных или сделать и то, и другое, чтобы инструкция выполнялась более эффективно. Настройка — это большая тема, и я действительно рекомендую вам либо прочесть руководство по настройке

вашего сервера, либо найти хорошую книгу по этой теме, чтобы увидеть все различные подходы, доступные для вашего сервера.

Обратная сторона индексов

Если индексы настолько хороши, почему бы не проиндексировать все, что только можно? Ключ к пониманию того, почему лучшее — враг хорошего, в том, что каждый индекс представляет собой таблицу (пусть и особый тип таблицы, но все же это таблица). Следовательно, каждый раз, когда строка добавляется в таблицу или удаляется из нее, должны быть изменены все индексы в этой таблице. При обновлении строки любые индексы для столбца (или столбцов), которые были затронуты, также должны быть изменены. Следовательно, чем больше у вас индексов, тем больше должен работать сервер, чтобы поддерживать все объекты схемы в актуальном состоянии — что приводит к замедлению работы.

Кроме того, индексы требуют дополнительного дискового пространства, а также некоторой осторожности со стороны администраторов, поэтому лучшая стратегия — добавлять индекс только тогда, когда в нем возникнет явная необходимость. Если вам нужен индекс только для специальных целей, например для ежемесячного обслуживания, можете добавить индекс, выполнить необходимые процедуры, а затем удалить индекс до тех пор, пока он вновь вам не понадобится. В случае хранилищ данных, где индексы имеют решающее значение в бизнес-время, когда пользователи составляют отчеты и выполняют запросы, но приводят к проблемам, когда ночью данные загружаются в хранилище, обычно перед загрузкой данных индексы удаляются, а затем воссоздаются перед тем, как хранилище вновь открывается для активной работы.

В общем случае вы должны стремиться не иметь ни слишком много индексов, ни слишком мало. Если вы не знаете, сколько индексов вам нужно, попробуйте по умолчанию использовать следующую стратегию.

- Убедитесь, что все столбцы первичного ключа проиндексированы (большинство серверов автоматически создают уникальные индексы при создании ограничений первичного ключа). Для многостолбцового первичного ключа рассмотрите возможность создания дополнительных индексов для подмножества столбцов первичного ключа или для всех столбцов первичного ключа, но в ином порядке, чем в определении ограничения первичного ключа.

- Создайте индексы для всех столбцов, на которые имеются ссылки в ограничениях внешнего ключа. Помните, что при удалении родительского элемента сервер проверяет, нет ли соответствующих дочерних строк, поэтому он должен выполнить запрос для поиска определенного значения в столбце. Если для такого столбца нет индекса, необходимо просканировать всю таблицу.
- Проиндексируйте все столбцы, которые будут часто использоваться для извлечения данных. Хорошими кандидатами являются большинство столбцов дат вместе с короткими (от 2 до 50 символов) строковыми столбцами.

После того как вы создали свой начальный набор индексов, просмотрите план выполнения сервером фактических запросов к вашим таблицам и измените свою стратегию индексирования так, чтобы она подходила для наиболее распространенных путей доступа.

Ограничения

Ограничение — это просто определенные условия, накладываемые на один или несколько столбцов таблицы. Имеется несколько различных типов ограничений, в том числе следующие.

Ограничения первичного ключа

Определяют столбец или столбцы, для которых гарантируется их уникальность в таблице

Ограничения внешнего ключа

Ограничивают содержимое одного или нескольких столбцов таким образом, что они могут содержать только значения, найденные в столбце первичного ключа другой таблицы (могут также ограничиваться допустимые значения в других таблицах при установке правил update cascade и/или delete cascade)

Ограничения уникальности

Ограничивают один или несколько столбцов таким образом, чтобы они содержали уникальные значения в таблице (ограничение первичного ключа — частный случай ограничения уникальности)

Проверочные ограничения

Ограничивают допустимые значения для столбца

Без ограничений согласованность базы данных сомнительна. Например, если сервер позволяет изменять идентификатор клиента в таблице `customer`, не меняя тот же идентификатор клиента в таблице `rental`, то вы получите данные о прокате, которые больше не указывают на корректные записи клиентов (известные как *потерянные строки* (*orphaned rows*)). При наличии же ограничений первичного ключа и ограничений внешнего ключа сервер либо сообщит об ошибке, если будет предпринята попытка изменить или удалить данные, на которые ссылаются другие таблицы, либо распространит изменения и на другие таблицы (подробнее об этом — чуть ниже).



Если вы хотите использовать ограничения внешнего ключа с сервером MySQL, для своих таблиц вы должны использовать механизм хранения InnoDB.

Создание ограничения

Ограничения обычно создаются одновременно со связанный таблицей с помощью инструкции `create table`. Чтобы проиллюстрировать это, ниже приведен пример из сценария генерации схемы образца базы данных Sakila:

```
CREATE TABLE customer (
    customer_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
    store_id TINYINT UNSIGNED NOT NULL,
    first_name VARCHAR(45) NOT NULL,
    last_name VARCHAR(45) NOT NULL,
    email VARCHAR(50) DEFAULT NULL,
    address_id SMALLINT UNSIGNED NOT NULL,
    active BOOLEAN NOT NULL DEFAULT TRUE,
    create_date DATETIME NOT NULL,
    last_update TIMESTAMP DEFAULT CURRENT_TIMESTAMP
        ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY (customer_id),
    KEY idx_fk_store_id (store_id),
    KEY idx_fk_address_id (address_id),
    KEY idx_last_name (last_name),
    CONSTRAINT fk_customer_address FOREIGN KEY (address_id)
        REFERENCES address (address_id)
        ON DELETE RESTRICT ON UPDATE CASCADE,
    CONSTRAINT fk_customer_store FOREIGN KEY (store_id)
        REFERENCES store (store_id)
        ON DELETE RESTRICT ON UPDATE CASCADE
)ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Таблица `customer` включает три ограничения: одно — чтобы указать, что столбец `customer_id` служит первичным ключом таблицы, и еще

два — чтобы указать, что столбцы `address_id` и `store_id` служат внешними ключами для таблиц `address` и `store`. В качестве альтернативы можно создать таблицу `customer` без ограничений внешнего ключа и добавить необходимые ограничения внешнего ключа с помощью инструкций `alter table`:

```
ALTER TABLE customer
ADD CONSTRAINT fk_customer_address FOREIGN KEY (address_id)
REFERENCES address (address_id) ON DELETE RESTRICT ON UPDATE CASCADE;

ALTER TABLE customer
ADD CONSTRAINT fk_customer_store FOREIGN KEY (store_id)
REFERENCES store (store_id) ON DELETE RESTRICT ON UPDATE CASCADE;
```

Обе эти инструкции включают несколько предложений `on`.

- `on delete restrict`, которое приводит к тому, что сервер выдаст ошибку, если в родительской таблице (`address` или `store`) будет удалена строка, на которую есть ссылка в дочерней таблице (`customer`).
- `on update cascade`, которое заставит сервер распространить изменение на значения первичного ключа родительской таблицы (`address` или `store`) на дочернюю таблицу (`customer`).

Предложение `on delete restrict` защищает от потерянных записей при удалении строк из родительской таблицы. Для иллюстрации возьмем строку в таблице `address` и покажем данные из таблиц `address` и `customer`, которые используют это значение:

```
mysql> SELECT c.first_name, c.last_name, c.address_id, a.address
-> FROM customer c
-> INNER JOIN address a
-> ON c.address_id = a.address_id
-> WHERE a.address_id = 123;
+-----+-----+-----+
|first_name |last_name |address_id |address
+-----+-----+-----+
|SHERRY    |MARSHALL |      123 |1987 Coacalco de Berriozbal Loop
+-----+-----+-----+
1 row in set (0.00 sec)
```

Результаты показывают, что существует одна строка `customer` (для `Sherry Marshall`), столбец `address_id` которой содержит значение 123.

Вот что произойдет, если вы попытаетесь удалить эту строку из родительской таблицы (`address`):

```
mysql> DELETE FROM address WHERE address_id = 123;
ERROR 1451 (23000): Cannot delete or update a parent row:
```

```
a foreign key constraint fails2 ('sakila`.`customer',
CONSTRAINT `fk_customer_address` FOREIGN KEY (`address_id`)
REFERENCES `address` (`address_id`)
ON DELETE RESTRICT ON UPDATE CASCADE)
```

Поскольку по крайней мере одна строка в дочерней таблице содержит значение 123 в столбце address_id, предложение on delete restrict ограничения внешнего ключа приводит к сообщению об ошибке.

Предложение on update cascade также защищает от потерянных записей, когда значение первичного ключа обновляется в родительской таблице, используя другую стратегию. Вот что произойдет, если вы измените значение в столбце address.address_id:

```
mysql> UPDATE address
   -> SET address_id = 9999
   -> WHERE address_id = 123;
Query OK, 1 row affected (0.37 sec)
Rows matched: 1 Changed: 1 Warnings: 0
```

Инструкция выполняется без ошибок, изменена одна строка. Но что произойдет со строкой Sherry Marshall в таблице customer? По-прежнему ли она указывает на идентификатор адреса 123, которого больше не существует? Чтобы выяснить это, давайте снова выполним последний запрос, но заменим предыдущее значение (123) новым значением (9999):

```
mysql> SELECT c.first_name, c.last_name, c.address_id, a.address
   -> FROM customer c
   -> INNER JOIN address a
   -> ON c.address_id = a.address_id
   -> WHERE a.address_id = 9999;
+-----+-----+-----+
|first_name |last_name |address_id |address
+-----+-----+-----+
|SHERRY    |MARSHALL |      9999 |1987 Coacalco de Berriozbal Loop|
+-----+-----+-----+
1 row in set (0.00 sec)
```

Как видите, возвращаются те же результаты, что и раньше (кроме нового значения идентификатора адреса), а это означает, что значение 9999 было автоматически обновлено в таблице customer. Это называется *каскадным обновлением*, и это второй механизм, используемый для защиты против потерянных строк.

² Невозможно удалить или обновить родительскую строку: нарушение ограничения внешнего ключа.

Наряду с `restrict` и `cascade` можно выбрать `set null`, что приведет к установке значения внешнего ключа в дочерней таблице равным `null`, когда в родительская таблице удаляется или обновляется строка. Всего существует шесть различных вариантов выбора при определении ограничения внешнего ключа:

- `on delete restrict`
- `on update cascade`
- `on delete set null`
- `on update restrict`
- `on update cascade`
- `on update set null`

Все они необязательны, поэтому вы можете выбрать нуль, один или два варианта (один — для `delete` и еще один — для `update`) при определении ограничений внешнего ключа.

Наконец, если вы хотите удалить ограничение первичного или внешнего ключа, то можете использовать оператор `alter table` еще раз, только вместо `add` укажите `drop`. В то время как сброс ограничения первичного ключа — действие необычное, ограничения внешнего ключа иногда сбрасываются при некоторых операциях по техническому обслуживанию, а затем вновь восстанавливаются.

Проверьте свои знания

Предлагаемые здесь упражнения призваны закрепить понимание вами индексов и ограничений. Ответы к упражнениям представлены в приложении Б.

УПРАЖНЕНИЕ 13.1

Сгенерируйте инструкцию `alter table` для таблицы `rental` так, чтобы генерировалось сообщение об ошибке, когда строка со значением, имеющимся в столбце `rent.customer_id`, удаляется из таблицы `customer`.

УПРАЖНЕНИЕ 13.2

Создайте многостолбцовый индекс в таблице `payment`, который может использоваться обоими приведенными далее запросами:

```
SELECT customer_id, payment_date, amount
FROM payment
WHERE payment_date > cast('2019-12-31 23:59:59' as datetime);

SELECT customer_id, payment_date, amount
FROM payment
WHERE payment_date > cast('2019-12-31 23:59:59' as datetime)
AND amount < 5;
```


Представления

Хорошо спроектированные приложения обычно предоставляют открытый интерфейс, сохраняя детали реализации скрытыми, что позволяет в будущем вносить изменения в дизайн, не затрагивая конечных пользователей. При разработке своей базы данных вы можете добиться аналогичного результата, сохраняя ваши таблицы закрытыми и позволяя пользователям получать доступ к данным только через набор *представлений*. В этой главе предпринята попытка определить, что такое представления, как они создаются и когда и как их можно использовать.

Что такое представление

Представление — это просто механизм запроса данных. В отличие от таблиц, представления не включают хранилище данных; вам не нужно беспокоиться о том, что представления займут место на диске. Вы создаете представление путем присвоения имени инструкции `select` с последующим сохранением запроса для использования другими пользователями. Другие пользователи могут затем использовать ваше представление для доступа к данным, как если бы они запрашивали таблицы непосредственно (на самом деле они могут даже не знать, что используют представления).

В качестве простого примера предположим, что вы хотите частично скрыть адрес электронной почты в таблице клиентов. Например, доступ к адресам электронной почты может потребоваться отделу маркетинга для рекламных акций, но в остальном политика конфиденциальности вашей компании требует, чтобы эти данные хранились в безопасности. Поэтому вместо того, чтобы разрешить прямой доступ к таблице клиентов, вы определяете представление с именем `customer_vw` и требуете, чтобы все, кроме персонала маркетинга, использовали его для доступа к данным о клиентах. Вот определение этого представления:

```

CREATE VIEW customer_vw
(
customer_id,
first_name,
last_name,
email
)
AS
SELECT
customer_id,
first_name,
last_name,
concat(substr(email,1,2), '*****', substr(email, -4)) email
FROM customer;

```

В первой части инструкции перечислены имена столбцов представления, которые могут отличаться от таковых в таблице, лежащей в основе представления. Вторая часть инструкции — инструкция выбора, которая должна содержать по одному выражению для каждого столбца в представлении. Столбец `email` создается путем соединения первых двух символов адреса электронной почты со строкой `'*****'`, а затем — с последними четырьмя символами электронного адреса.

Когда выполняется инструкция `create view`, сервер базы данных просто сохраняет определение представления для будущего использования. Запрос не выполняется, данные не извлекаются и не хранятся. После создания представления пользователи могут запрашивать его так же, как и таблицу:

```

mysql> SELECT first_name, last_name, email
-> FROM customer_vw;
+-----+-----+-----+
| first_name | last_name | email      |
+-----+-----+-----+
| MARY       | SMITH     | MA*****.org |
| PATRICIA   | JOHNSON   | PA*****.org |
| LINDA      | WILLIAMS  | LI*****.org |
| BARBARA    | JONES     | BA*****.org |
| ELIZABETH  | BROWN     | EL*****.org |
| ...        |           |             |
| ENRIQUE    | FORSYTHE  | EN*****.org |
| FREDDIE    | DUGGAN    | FR*****.org |
| WADE       | DELVALLE  | WA*****.org |
| AUSTIN     | CINTRON   | AU*****.org |
+-----+-----+-----+

```

599 rows in set (0.00 sec)

Несмотря на то что определение представления `customer_vw` включает четыре столбца таблицы `customer`, приведенный запрос извлекает только три из них. Как вы увидите ниже в главе, это важное отличие, если некоторые из столбцов в вашем представлении связаны с функциями или подзапросами.

С точки зрения пользователя, представление выглядит точно так же, как и таблица. Если вы хотите узнать, какие столбцы доступны в представлении, можете использовать команду MySQL (или Oracle Database) `describe` для его изучения:

```
mysql> describe customer_vw;
+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| customer_id | smallint(5) unsigned | NO | | 0 | |
| first_name | varchar(45) | NO | | NULL | |
| last_name | varchar(45) | NO | | NULL | |
| email | varchar(11) | YES | | NULL | |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

Вы можете использовать любые предложения инструкции `select` при запросе данных через представление, в том числе `group by`, `having` и `order by`:

```
mysql> SELECT first_name, count(*), min(last_name), max(last_name)
-> FROM customer_vw
-> WHERE first_name LIKE 'J%'
-> GROUP BY first_name
-> HAVING count(*) > 1
-> ORDER BY 1;
+-----+-----+-----+
| first_name | count(*) | min(last_name) | max(last_name) |
+-----+-----+-----+
| JAMIE | 2 | RICE | WAUGH |
| JESSIE | 2 | BANKS | MILAM |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

Кроме того, в запросе вы можете соединять представления с другими таблицами (или даже с другими представлениями):

```
mysql> SELECT cv.first_name, cv.last_name, p.amount
-> FROM customer_vw cv
-> INNER JOIN payment p
-> ON cv.customer_id = p.customer_id
-> WHERE p.amount >= 11;
+-----+-----+-----+
| first_name | last_name | amount |
+-----+-----+-----+
| KAREN | JACKSON | 11.99 |
| VICTORIA | GIBSON | 11.99 |
| VANESSA | SIMS | 11.99 |
| ALMA | AUSTIN | 11.99 |
| ROSEMARY | SCHMIDT | 11.99 |
| TANYA | GILBERT | 11.99 |
```

```
| RICHARD      | MCCRARY     | 11.99 |
| NICHOLAS    | BARFIELD    | 11.99 |
| KENT         | ARSENAULT   | 11.99 |
| TERRANCE    | ROUSH       | 11.99 |
+-----+-----+-----+
10 rows in set (0.01 sec)
```

Этот запрос соединяет представление `customer_vw` с таблицей `payment` для поиска клиентов, заплативших за прокат фильма не менее 11 долларов.

Зачем использовать представления

В предыдущем разделе я продемонстрировал простое представление, единственной целью которого была маскировка содержимого столбца `customer.email`. Хотя представления часто используются для этой цели, есть много иных причин для использования представлений, как подробно описано ниже.

Безопасность данных

Если вы создадите таблицу и разрешите пользователям запрашивать ее, они смогут получить доступ к каждому столбцу и каждой строке таблицы. Однако, как уже указывалось, ваша таблица может включать некоторые столбцы, содержащие конфиденциальные данные, такие как идентификационные номера или номера кредитных карт. Предоставлять такие данные всем пользователям — это не просто плохая идея; это может нарушить политику конфиденциальности вашей компании или даже законы страны.

Наилучший подход в таких ситуациях — оставить таблицу закрытой (т.е. не предоставлять разрешение выполнять `select` всем пользователям), а затем создать одно или несколько представлений, которые либо опускают, либо скрывают (например, с помощью подхода `'*****'`, использованного в столбце `customer_vw.email`) конфиденциальные столбцы. Вы также можете указать, к каким строкам могут иметь доступ пользователи, добавив в определение вашего представления предложение `where`. Например, следующее определение представления исключает неактивных клиентов:

```
CREATE VIEW active_customer_vw
(customer_id,
 first_name,
 last_name,
 email
)
AS
SELECT
```

```
customer_id,  
first_name,  
last_name,  
concat(substr(email,1,2), '*****', substr(email, -4)) email  
FROM customer  
WHERE active = 1;
```

Если вы обеспечите такое представление для отдела маркетинга, они смогут избежать отправки информации неактивным клиентам, потому что условие в предложении `where` представления будет всегда включено в их запросы.



У пользователей Oracle Database есть еще один вариант защиты как строк, так и столбцов таблиц: виртуальная частная база данных (Virtual Private Database, VPD). VPD позволяет назначать политики вашим таблицам, после чего сервер будет при необходимости изменять запрос пользователя так, чтобы обеспечить соблюдение назначенных политик. Например, если вы вводите политику, согласно которой сотрудники отделов продаж и маркетинга могут видеть только активных клиентов, ко всем их запросам к таблице `customer` будет добавлено условие `active = 1`.

Агрегация данных

Приложениям для создания отчетов обычно требуются агрегированные данные, а представления — отличное средство создания впечатления, что в базе данных сохраняются предварительно агрегированные данные. В качестве примера предположим, что приложение ежемесячно создает отчет, показывающий общий объем продаж по каждой категории фильмов, чтобы менеджеры могли решить, какие новые фильмы закупать. Вместо того чтобы позволять разработчикам приложений писать запросы, обращающиеся к базовым таблицам, им можно предложить следующее представление¹:

```
CREATE VIEW sales_by_film_category  
AS  
SELECT  
    c.name AS category,  
    SUM(p.amount) AS total_sales  
FROM payment AS p  
    INNER JOIN rental AS r ON p.rental_id = r.rental_id  
    INNER JOIN inventory AS i ON r.inventory_id = i.inventory_id  
    INNER JOIN film AS f ON i.film_id = f.film_id
```

¹ Это определение представления включено в образец базы данных Sakila вместе с шестью другими (некоторые из них будут использованы в следующих примерах).

```
INNER JOIN film_category AS fc ON f.film_id = fc.film_id  
INNER JOIN category AS c ON fc.category_id = c.category_id  
GROUP BY c.name  
ORDER BY total_sales DESC;
```

Этот подход дает вам как разработчику базы данных большую гибкость. Если в какой-то момент в будущем будет решено, что производительность запросов значительно улучшится, если данные будут предварительно сгруппированы в таблице, а не подытожены с использованием представления, вы сможете создать таблицу `film_category_sales`, загрузить в нее агрегированные данные и изменить определение представления `sales_by_film_category` так, что оно будет извлекать данные из этой таблицы. После этого все запросы, использующие представление `sales_by_film_category`, будут извлекать данные из новой таблицы `film_category_sales`, так что пользователи увидят улучшение производительности, никак не меняя свои запросы.

Сокрытие сложности

Одна из наиболее распространенных причин развертывания представлений — защита конечных пользователей от сложности запросов. Например, предположим, что каждый месяц создается отчет, в котором отображается информация обо всех фильмах наряду с категорией фильмов, количеством актеров, фигурирующих в фильме, общим наличным количеством копий и количеством прокатов каждого фильма. Вместо того чтобы ожидать, что составитель отчетов будет выбирать информацию из шести разных таблиц для сбора необходимых данных, вы можете предложить ему представление, которое выглядит следующим образом:

```
CREATE VIEW film_stats  
AS  
SELECT f.film_id, f.title, f.description, f.rating,  
(SELECT c.name  
FROM category c  
INNER JOIN film_category fc  
ON c.category_id = fc.category_id  
WHERE fc.film_id = f.film_id) category_name,  
(SELECT count(*))  
FROM film_actor fa  
WHERE fa.film_id = f.film_id  
) num_actors,  
(SELECT count(*))  
FROM inventory i  
WHERE i.film_id = f.film_id  
) inventory_cnt,  
(SELECT count(*))  
FROM inventory i
```

```
    INNER JOIN rental r
    ON i.inventory_id = r.inventory_id
   WHERE i.film_id = f.film_id
) num_rentals
FROM film f;
```

Это определение представления очень интересно, потому что, хотя через представление можно получить данные из шести разных таблиц, предложение `from` запроса включает только одну таблицу (`film`). Данные из остальных пяти таблиц генерируются с помощью скалярных подзапросов. Если кто-то использует это представление, но не обращается к столбцам `category_name`, `num_actors`, `inventory_cnt` и `num_rentals`, то ни один из подзапросов выполниться не будет. Этот подход позволяет использовать представление для предоставления описательной информации из таблицы `film` без необходимости соединения с пятью другими таблицами.

Соединение разделенных данных

Некоторые проекты базы данных разбивают большие таблицы на несколько таблиц, чтобы повысить производительность. Например, если таблица `payment` становится слишком большой, дизайнеры могут решить разбить ее на две таблицы: `payment_current`, которая содержит сведения за последние шесть месяцев, и `payment_historic`, которая содержит все более поздние платежи. Если требуется увидеть все платежи для конкретного клиента, нужно запрашивать обе таблицы. Создавая представление, которое запрашивает обе таблицы и объединяет полученные результаты, можно просмотреть результаты, как если бы все данные о платежах хранились в одной таблице. Вот определение такого представления:

```
CREATE VIEW payment_all
(payment_id,
 customer_id,
 staff_id,
 rental_id,
 amount,
 payment_date,
 last_update
)
AS
SELECT payment_id, customer_id, staff_id, rental_id,
       amount, payment_date, last_update
  FROM payment_historic
 UNION ALL
SELECT payment_id, customer_id, staff_id, rental_id,
```

```
amount, payment_date, last_update  
FROM payment_current;
```

Использование представления в этом случае — хорошая идея, потому что позволяет дизайнерам изменить структуру базовых данных без необходимости заставить всех пользователей базы данных модифицировать их запросы.

Обновляемые представления

Мы показали, как использовать представления для поиска данных пользователями. Но что следует сделать, если пользователям нужно также изменять данные? Решение позволить пользователям получать информацию с помощью представления, а затем вносить изменения в лежащие в их основе таблицы с помощью `update` или `insert` выглядит несколько непоследовательным, чтобы не сказать большего. Для этого MySQL, Oracle Database и SQL Server позволяют изменять данные через представления, если при этом соблюдаются определенные ограничения. В случае MySQL представление является обновляемым, если выполнены следующие условия.

- Не используются агрегатные функции (`max()`, `min()`, `avg()` и т.д.).
- Представление не использует положения `group by` и `having`.
- В предложениях `select` и `from` нет никаких подзапросов, а любой подзапрос в предложении `where` не ссылается на таблицы в предложении `from`.
- Представление не использует `union`, `union all` или `distinct`.
- Предложение `from` содержит как минимум одну таблицу или обновляемое представление.
- Если имеется более одной таблицы или представления, предложение `from` использует только внутренние соединения.

Чтобы продемонстрировать полезность обновленных представлений, лучше всего начать с определения простого представления, а затем перейти к более сложным представлениям.

Обновление простых представлений

Представление в начале главы очень простое, так что давайте начнем с него:

```

CREATE VIEW customer_vw
(
customer_id,
first_name,
last_name,
email
)
AS
SELECT
customer_id,
first_name,
last_name,
concat(substr(email,1,2), '*****', substr(email, -4)) email
FROM customer;

```

Представление `customer_vw` запрашивает единственную таблицу, и только один из четырех столбцов получается с помощью выражения. Это определение представления не нарушает приведенные выше ограничения, поэтому его можно использовать для изменения данных в таблице `customer`. Давайте воспользуемся данным представлением, чтобы обновить фамилию Mary Smith на Smith-Allen:

```

mysql> UPDATE customer_vw
-> SET last_name = 'SMITH-ALLEN'
-> WHERE customer_id = 1;
Query OK, 1 row affected (0.11 sec)
Rows matched: 1 Changed: 1 Warnings: 0

```

Как видите, мы получили сообщение об успешном изменении одной строки. Тем не менее давайте проверим лежащую в основе представления таблицу `customer`, чтобы убедиться в этом:

```

mysql> SELECT first_name, last_name, email
-> FROM customer
-> WHERE customer_id = 1;
+-----+-----+-----+
| first_name | last_name   | email          |
+-----+-----+-----+
| MARY      | SMITH-ALLEN | MARY.SMITH@sakilacustomer.org |
+-----+-----+-----+
1 row in set (0.00 sec)

```

Хотя таким образом можно изменить большую часть столбцов представления, модифицировать столбец электронной почты не удастся, так как он получен из выражения:

```

mysql> UPDATE customer_vw
-> SET email = 'MARY.SMITH-ALLEN@sakilacustomer.org'

```

```
-> WHERE customer_id = 1;
ERROR 1348 (HY000): Column 'email' is not updatable2
```

В данном конкретном случае это может быть и неплохо, поскольку основной причиной создания представления было сокрытие адресов электронной почты.

Если вы захотите вставить данные с помощью представления `customer_vw`, то потерпите неудачу. Представление, содержащее выводимые столбцы, не может быть использовано для вставки данных, даже если эти столбцы не включены в инструкцию. Например, следующая инструкция, используя представление `customer_vw`, пытается заполнить только столбцы `customer_id`, `first_name` и `last_name`:

```
mysql> INSERT INTO customer_vw
-> (customer_id,
-> first_name,
-> last_name)
-> VALUES (99999,'ROBERT','SIMPSON');
ERROR 1471 (HY000): The target table customer_vw of
the INSERT is not insertable-into3
```

Теперь, когда вы видели ограничения простых представлений, в следующем разделе будет показано использование представления, которое присоединяется к нескольким таблицам.

Обновление сложных представлений

В то время как представления с одной таблицей, безусловно, являются наиболее распространенными, многие представления, с которыми вы будете встречаться, будут включать несколько таблиц в предложении `from` их запроса. Следующее представление, например, соединяет таблицы `customer`, `address`, `city` и `country` так, чтобы можно было легко запросить все данные клиентов:

```
CREATE VIEW customer_details
AS
SELECT c.customer_id,
       c.store_id,
       c.first_name,
       c.last_name,
       c.address_id,
       c.active,
```

² Столбец 'email' является необновляемым.

³ Целевая таблица `customer_vw` инструкции `INSERT` не допускает вставку.

```
c.create_date,  
a.address,  
ct.city,  
cn.country,  
a.postal_code  
FROM customer c  
INNER JOIN address a  
    ON c.address_id = a.address_id  
INNER JOIN city ct  
    ON a.city_id = ct.city_id  
INNER JOIN country cn  
    ON ct.country_id = cn.country_id;
```

Это представление можно использовать для обновления данных в таблице `customer` или `address`, как показано в следующих инструкциях:

```
mysql> UPDATE customer_details  
    -> SET last_name = 'SMITH-ALLEN', active = 0  
    -> WHERE customer_id = 1;  
Query OK, 1 row affected (0.10 sec)  
Rows matched: 1 Changed: 1 Warnings: 0  
  
mysql> UPDATE customer_details  
    -> SET address = '999 Mockingbird Lane'  
    -> WHERE customer_id = 1;  
Query OK, 1 row affected (0.06 sec)  
Rows matched: 1 Changed: 1 Warnings: 0
```

Первая инструкция изменяет столбцы `customer.last_name` и `customer.active`, в то время как вторая модифицирует столбец `address.address`. Вас может заинтересовать, что произойдет, если попытаться обновить столбцы обеих таблиц в одной инструкции. Давайте попробуем:

```
mysql> UPDATE customer_details  
    -> SET last_name = 'SMITH-ALLEN',  
    -> active = 0,  
    -> address = '999 Mockingbird Lane'  
    -> WHERE customer_id = 1;  
ERROR 1393 (HY000): Can not modify more than one base table  
through a join view 'sakila.customer_details'4
```

Как видите, вам разрешено изменять базовые таблицы по отдельности, но не в одной инструкции. Давайте теперь попробуем *вставить* данные в обе таблицы для некоторых новых клиентов (со значениями `customer_id`, равными 9998 и 9999):

⁴ Невозможно изменить более одной базовой таблицы с помощью соединяющего представления 'sakila.customer_details'.

```
mysql> INSERT INTO customer_details
->   (customer_id, store_id, first_name, last_name,
->    address_id, active, create_date)
-> VALUES (9998, 1, 'BRIAN', 'SALAZAR', 5, 1, now());
Query OK, 1 row affected (0.23 sec)
```

Эта инструкция, которая заполняет столбцы только таблицы `customer`, отлично работает. Давайте теперь посмотрим, что произойдет, если мы расширим список столбцов и включим в него столбец из таблицы `address`:

```
mysql> INSERT INTO customer_details
->   (customer_id, store_id, first_name, last_name,
->    address_id, active, create_date, address)
-> VALUES (9999, 2, 'THOMAS', 'BISHOP', 7, 1, now(),
->          '999 Mockingbird Lane');
ERROR 1393 (HY000): Can not modify more than one base table
through a join view 'sakila.customer_details'5
```

Эта версия, которая включает в себя столбцы, охватывающие две разные таблицы, приводит к ошибке. Для того чтобы вставить данные с помощью сложного представления, нужно знать, откуда получен каждый из столбцов. Поскольку многие представления созданы для сокрытия сложности от конечных пользователей, явное знание пользователями структуры представления противоречит поставленной цели.



Oracle Database и SQL Server также разрешают вставку и обновление данных через представления, но, как и MySQL, предъявляют много ограничений. Если вы готовы писать код на PL/SQL или Transact-SQL, можете использовать функциональную возможность под названием “*триггеры вместо*” (*instead-of triggers*), которая, по сути, позволяет перехватывать инструкции `insert`, `update` и `delete` для представления и писать пользовательский код для внесения соответствующих изменений. Без применения такого рода функций, как правило, имеется слишком много ограничений, чтобы суметь выполнить обновление данных с помощью представлений в нетривиальных приложениях.

⁵ Невозможно изменить более одной базовой таблицы с помощью соединяющего представления 'sakila.customer_details'.

Проверьте свои знания

Предлагаемые здесь упражнения призваны закрепить понимание вами представлений. Ответы к упражнениям представлены в приложении Б.

УПРАЖНЕНИЕ 14.1

Создайте определение представления, которое можно использовать с помощью следующего запроса для генерации приведенного результирующего набора:

```
SELECT title, category_name, first_name, last_name  
FROM film_ctgry_actor  
WHERE last_name = 'FAWCETT';
```

title	category_name	first_name	last_name
ACE GOLDFINGER	Horror	BOB	FAWCETT
ADAPTATION HOLES	Documentary	BOB	FAWCETT
CHINATOWN GLADIATOR	New	BOB	FAWCETT
CIRCUS YOUTH	Children	BOB	FAWCETT
CONTROL ANTHEM	Comedy	BOB	FAWCETT
DARES PLUTO	Animation	BOB	FAWCETT
DARN FORRESTER	Action	BOB	FAWCETT
DAZED PUNK	Games	BOB	FAWCETT
DYNAMITE TARZAN	Classics	BOB	FAWCETT
HATE HANDICAP	Comedy	BOB	FAWCETT
HOMICIDE PEACH	Family	BOB	FAWCETT
JACKET FRISCO	Drama	BOB	FAWCETT
JUMANJI BLADE	New	BOB	FAWCETT
LAWLESS VISION	Animation	BOB	FAWCETT
LEATHERNECKS DWARFS	Travel	BOB	FAWCETT
OSCAR GOLD	Animation	BOB	FAWCETT
PELICAN COMFORTS	Documentary	BOB	FAWCETT
PERSONAL LADYBUGS	Music	BOB	FAWCETT
RAGING AIRPLANE	Sci-Fi	BOB	FAWCETT
RUN PACIFIC	New	BOB	FAWCETT
RUNNER MADIGAN	Music	BOB	FAWCETT
SADDLE ANTITRUST	Comedy	BOB	FAWCETT
SCORPION APOLLO	Drama	BOB	FAWCETT
SHAWSHANK BUBBLE	Travel	BOB	FAWCETT
TAXI KICK	Music	BOB	FAWCETT
BERETS AGENT	Action	JULIA	FAWCETT
BOILED DARES	Travel	JULIA	FAWCETT
CHISUM BEHAVIOR	Family	JULIA	FAWCETT
CLOSER BANG	Comedy	JULIA	FAWCETT
DAY UNFAITHFUL	New	JULIA	FAWCETT
HOPE TOOTSIE	Classics	JULIA	FAWCETT
LUKE MUMMY	Animation	JULIA	FAWCETT
MULAN MOON	Comedy	JULIA	FAWCETT

OPUS ICE	Foreign	JULIA	FAWCETT	
POLLOCK DELIVERANCE	Foreign	JULIA	FAWCETT	
RIDGEMONT SUBMARINE	New	JULIA	FAWCETT	
SHANGHAI TYCOON	Travel	JULIA	FAWCETT	
SHAWSHANK BUBBLE	Travel	JULIA	FAWCETT	
THEORY MERMAID	Animation	JULIA	FAWCETT	
WAIT CIDER	Animation	JULIA	FAWCETT	

+-----+-----+-----+-----+

40 rows in set (0.00 sec)

УПРАЖНЕНИЕ 14.2

Менеджер компании по прокату фильмов хотел бы иметь отчет, который включает в себя название каждой страны, а также общие платежи всех клиентов, которые живут в данной стране. Создайте определение представления, которое запрашивает таблицу `country` и использует для вычисления значения столбца `tot_payments` скалярный подзапрос.

Метаданные

Наряду с хранением всех данных, которые различные пользователи вносят в базу данных, сервер базы данных должен хранить информацию обо всех объектах базы данных (таблицах, представлениях, индексах и т.д.), которые были созданы для хранения этих данных. Сервер базы данных хранит эту информацию, что не удивительно, в самой базе данных. В этой главе обсуждается, как и где хранится эта информация, известная как *метаданные*, как получить к ней доступ и как ее использовать для построения гибких систем.

Данные о данных

Метаданные, по существу, являются данными о данных. Каждый раз, когда вы создаете объект базы данных, сервер базы данных должен записывать различную информацию о нем. Например, если вам необходимо создать таблицу с несколькими столбцами, ограничением первичного ключа, тремя индексами и ограничением внешнего ключа, сервер базы данных должен хранить следующую информацию.

- Имя таблицы
- Информация о хранении таблицы (табличное пространство, начальный размер и т.д.)
- Механизм хранения
- Имена столбцов
- Типы данных столбцов
- Значения столбцов по умолчанию
- Ограничения not null для столбцов
- Столбцы первичного ключа
- Имя первичного ключа
- Имя индекса первичного ключа

- Имена индексов
- Типы индексов (B-tree, bitmap)
- Индексированные столбцы
- Порядок сортировки индексов столбцов (по возрастанию или по убыванию)
- Информация о хранении индекса
- Имя внешнего ключа
- Столбцы внешнего ключа
- Связанные таблицы/столбцы для внешних ключей

Эти данные коллективно известны как *словарь данных* или *системный каталог*. Сервер базы данных должен хранить эти данные в постоянном хранилище и быть в состоянии быстро их получить для проверки и выполнения инструкций SQL. Кроме того, сервер базы данных должен защищать эти данные, чтобы их можно было модифицировать только через соответствующий механизм, такой как инструкция alter table.

При наличии стандартов для обмена метаданными между разными серверами каждый сервер базы данных использует свой механизм для публикации метаданных.

- Набор представлений, таких как представления user_tables и all_constraints в Oracle Database
- Набор системных процедур, таких как процедура sp_tables в SQL Server или пакет dbms_metadata в Oracle Database
- Специальная база данных, такая как база данных information_schema в MySQL

Наряду с системно хранимыми процедурами SQL Server, которые являются пережитком Sybase, SQL Server включает в себя специальную схему с именем information_schema, которая автоматически предоставляется в каждой базе данных. И MySQL, и SQL Server предоставляют этот интерфейс для соответствия стандарту ANSI SQL:2003. В оставшейся части этой главы рассматриваются объекты information_schema, которые доступны в MySQL и SQL Server.

information_schema

Все объекты, доступные в базе данных information_schema, являются представлениями. В отличие от утилиты describe, которую я использовал

в нескольких главах этой книги как способ показать структуру различных таблиц и представлений, представления в information_schema могут быть запрошены и, таким образом, использоваться программно (подробнее об этом — ниже в данной главе). Вот пример, который демонстрирует, как получить имена всех таблиц в базе данных Sakila:

```
mysql> SELECT table_name, table_type
-> FROM information_schema.tables
-> WHERE table_schema = 'sakila'
-> ORDER BY 1;
+-----+-----+
| TABLE_NAME | TABLE_TYPE |
+-----+-----+
| actor      | BASE TABLE |
| actor_info | VIEW        |
| address    | BASE TABLE |
| category   | BASE TABLE |
| city       | BASE TABLE |
| country   | BASE TABLE |
| customer  | BASE TABLE |
| customer_list | VIEW      |
| film       | BASE TABLE |
| film_actor | BASE TABLE |
| film_category | BASE TABLE |
| film_list  | VIEW        |
| film_text  | BASE TABLE |
| inventory  | BASE TABLE |
| language   | BASE TABLE |
| nicer_but_slower_film_list | VIEW      |
| payment    | BASE TABLE |
| rental     | BASE TABLE |
| sales_by_film_category | VIEW      |
| sales_by_store  | VIEW      |
| staff      | BASE TABLE |
| staff_list | VIEW        |
| store      | BASE TABLE |
+-----+-----+
23 rows in set (0.00 sec)
```

Как видите, представление information_schema.tables включает в себя и таблицы, и представления. Если вы хотите исключить представления, просто добавьте второе условие в предложение where:

```
mysql> SELECT table_name, table_type
-> FROM information_schema.tables
-> WHERE table_schema = 'sakila'
-> AND table_type = 'BASE TABLE'
-> ORDER BY 1;
+-----+-----+
```

```

| TABLE_NAME      | TABLE_TYPE |
+-----+-----+
| actor          | BASE TABLE |
| address         | BASE TABLE |
| category        | BASE TABLE |
| city            | BASE TABLE |
| country          | BASE TABLE |
| customer        | BASE TABLE |
| film            | BASE TABLE |
| film_actor      | BASE TABLE |
| film_category   | BASE TABLE |
| film_text       | BASE TABLE |
| inventory       | BASE TABLE |
| language         | BASE TABLE |
| payment          | BASE TABLE |
| rental           | BASE TABLE |
| staff            | BASE TABLE |
| store            | BASE TABLE |
+-----+
16 rows in set (0.00 sec)

```

Если вас интересует только информация о представлениях, можете запросить `information_schema.views`. Наряду с именами представлений вы можете получить дополнительную информацию, такую как флаг, который показывает, является ли представление обновляемым:

```

mysql> SELECT table_name, is_updatable
    -> FROM information_schema.views
    -> WHERE table_schema = 'sakila'
    -> ORDER BY 1;
+-----+-----+
| TABLE_NAME      | IS_UPDATABLE |
+-----+-----+
| actor_info      | NO          |
| customer_list   | YES         |
| film_list       | NO          |
| nicer_but_slower_film_list | NO |
| sales_by_film_category | NO |
| sales_by_store  | NO          |
| staff_list      | YES         |
+-----+
7 rows in set (0.00 sec)

```

Информация о столбцах как таблиц, так и представлений доступна с помощью представления `columns`. Следующий запрос показывает информацию о столбцах таблицы `film`:

```

mysql> SELECT column_name, data_type,
    ->     character_maximum_length char_max_len,
    ->     numeric_precision num_prcsn, numeric_scale num_scale

```

```

-> FROM information_schema.columns
-> WHERE table_schema = 'sakila' AND table_name = 'film'
-> ORDER BY ordinal_position;
+-----+-----+-----+-----+-----+
|COLUMN_NAME |DATA_TYPE |char_max_len |num_prcsn |num_scale |
+-----+-----+-----+-----+-----+
|film_id |smallint |NULL |5 |0 |
|title |varchar |255 |NULL |NULL |
|description |text |65535 |NULL |NULL |
|release_year |year |NULL |NULL |NULL |
|language_id |tinyint |NULL |3 |0 |
|original_language_id |tinyint |NULL |3 |0 |
|rental_duration |tinyint |NULL |3 |0 |
|rental_rate |decimal |NULL |4 |2 |
|length |smallint |NULL |5 |0 |
|replacement_cost |decimal |NULL |5 |2 |
|rating |enum |5 |NULL |NULL |
|special_features |set |54 |NULL |NULL |
|last_update |timestamp |NULL |NULL |NULL |
+-----+-----+-----+-----+-----+
13 rows in set (0.00 sec)

```

Столбец `ordinal_position` включен просто как средство для получения столбцов в порядке, в котором они были добавлены в таблицу.

Вы можете получить информацию об индексах таблицы с помощью представления `information_schema.statistics`, как показано в следующем запросе, который извлекает информацию об индексах, созданных для таблицы `rental`:

```

mysql> SELECT index_name, non_unique, seq_in_index, column_name
-> FROM information_schema.statistics
-> WHERE table_schema = 'sakila' AND table_name = 'rental'
-> ORDER BY 1, 3;
+-----+-----+-----+-----+
| INDEX_NAME | NON_UNIQUE | SEQ_IN_INDEX | COLUMN_NAME |
+-----+-----+-----+-----+
| idx_fk_customer_id | 1 | 1 | customer_id |
| idx_fk_inventory_id | 1 | 1 | inventory_id |
| idx_fk_staff_id | 1 | 1 | staff_id |
| PRIMARY | 0 | 1 | rental_id |
| rental_date | 0 | 1 | rental_date |
| rental_date | 0 | 2 | inventory_id |
| rental_date | 0 | 3 | customer_id |
+-----+-----+-----+-----+
7 rows in set (0.02 sec)

```

Таблица `rental` имеет в общей сложности пять индексов, один из которых — трехстолбцовый (`rental_date`), а другой является `PRIMARY`-индексом, используемым для ограничения первичного ключа.

Вы можете получить различные типы ограничений (внешнего ключа, первичного ключа, уникальный) с помощью представления information_schema.table_constraints. Вот запрос, который извлекает все ограничения из схемы Sakila:

```
mysql> SELECT constraint_name, table_name, constraint_type
-> FROM information_schema.table_constraints
-> WHERE table_schema = 'sakila'
-> ORDER BY 3,1;
```

constraint_name	table_name	constraint_type
fk_address_city	address	FOREIGN KEY
fk_city_country	city	FOREIGN KEY
fk_customer_address	customer	FOREIGN KEY
fk_customer_store	customer	FOREIGN KEY
fk_film_actor_actor	film_actor	FOREIGN KEY
fk_film_actor_film	film_actor	FOREIGN KEY
fk_film_category_category	film_category	FOREIGN KEY
fk_film_category_film	film_category	FOREIGN KEY
fk_film_language	film	FOREIGN KEY
fk_film_language_original	film	FOREIGN KEY
fk_inventory_film	inventory	FOREIGN KEY
fk_inventory_store	inventory	FOREIGN KEY
fk_payment_customer	payment	FOREIGN KEY
fk_payment_rental	payment	FOREIGN KEY
fk_payment_staff	payment	FOREIGN KEY
fk_rental_customer	rental	FOREIGN KEY
fk_rental_inventory	rental	FOREIGN KEY
fk_rental_staff	rental	FOREIGN KEY
fk_staff_address	staff	FOREIGN KEY
fk_staff_store	staff	FOREIGN KEY
fk_store_address	store	FOREIGN KEY
fk_store_staff	store	FOREIGN KEY
PRIMARY	film	PRIMARY KEY
PRIMARY	film_actor	PRIMARY KEY
PRIMARY	staff	PRIMARY KEY
PRIMARY	film_category	PRIMARY KEY
PRIMARY	store	PRIMARY KEY
PRIMARY	actor	PRIMARY KEY
PRIMARY	film_text	PRIMARY KEY
PRIMARY	address	PRIMARY KEY
PRIMARY	inventory	PRIMARY KEY
PRIMARY	customer	PRIMARY KEY
PRIMARY	category	PRIMARY KEY
PRIMARY	language	PRIMARY KEY
PRIMARY	city	PRIMARY KEY
PRIMARY	payment	PRIMARY KEY
PRIMARY	country	PRIMARY KEY
PRIMARY	rental	PRIMARY KEY

```

| idx_email           | customer      | UNIQUE      |
| idx_unique_manager | store         | UNIQUE      |
| rental_date        | rental        | UNIQUE      |
+-----+-----+-----+
41 rows in set (0.02 sec)

```

В табл. 15.1 показано множество представлений `information_schema`, доступных в MySQL версии 8.0.

Таблица 15.1. Представления `information_schema`

Представление	Предоставляемая информация
schemata	Базы данных
tables	Таблицы и представления
columns	Столбцы таблиц и представлений
statistics	Индексы
user_privileges	Кто имеет привилегии для различных объектов схемы
schema_privileges	Кто имеет привилегии для различных баз данных
table_privileges	Кто имеет привилегии для различных таблиц
column_privileges	Кто имеет привилегии для различных столбцов таблиц
character_sets	Доступные наборы символов
collations	Какие сравнения доступны для различных наборов символов
collation_character_set_applicability	Какие наборы символов доступны для различных сравнений
table_constraints	Ограничения — первичного ключа, внешнего ключа, уникальности
key_column_usage	Ограничения, связанные с каждым ключевым столбцом
routines	Сохраненные подпрограммы (процедуры и функции)
views	Представления
triggers	Триггеры таблиц
plugins	Подключаемые модули сервера
engines	Доступные механизмы хранения
partitions	Разбиения таблиц
events	Запланированные события
processlist	Выполняющиеся процессы
referential_constraints	Внешние ключи
parameters	Параметры хранимых процедур и функций
profiling	Информация о профилях пользователей

В то время как некоторые из этих представлений, такие как `engines`, `events` и `plugins`, специфичны для MySQL, многие из них доступны и в SQL Server. Если вы используете Oracle Database, обратитесь к онлайн-справочнику Oracle Database Reference Guide (<https://oreil.ly/qV7sE>) за

информацией о представлениях user_ , all_ и dba_ , а также о пакете dbms_ metadata.

Работа с метаданными

Как уже упоминалось, возможность получить информацию об объектах схемы через SQL-запросы открывает некоторые интересные возможности. В этом разделе показано несколько способов использования метаданных в ваших приложениях.

Сценарии генерации схемы

В то время как в одних проектных командах работают отдельные проектировщики баз данных на полной ставке, которые контролируют проектирование и реализацию базы данных, в других используется подход “проектирование голосованием”, позволяющее создавать объекты базы данных множеству сотрудников. Через несколько недель или месяцев разработки вам, возможно, понадобится генерировать сценарий, который будет создавать различные таблицы, индексы, представления и так далее, которые разработаны командой. Хотя имеется масса разнообразных инструментов и утилит для генерации такого типа сценариев, вы можете также запросить представление information_schema и создать такой сценарий самостоятельно.

В качестве примера давайте построим сценарий, который создает таблицу sakila.category. Вот команда создания таблицы, которую я извлек из сценария, используемого для создания этой базы данных:

```
CREATE TABLE category (
    category_id TINYINT UNSIGNED NOT NULL AUTO_INCREMENT,
    name VARCHAR(25) NOT NULL,
    last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
        ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY (category_id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

Хотя, безусловно, было бы легче создать сценарий с использованием процедурного языка (наподобие Transact-SQL или Java), поскольку все же это книга о SQL, я предпочитаю написать запрос, который будет генерировать инструкцию create table. Первый шаг состоит в том, чтобы запросить таблицу information_schema.columns и получить информацию о столбцах таблицы:

```
mysql> SELECT 'CREATE TABLE category (' create_table_statement
-> UNION ALL
```

```

-> SELECT cols.txt
-> FROM
->   (SELECT concat(' ',column_name, ' ', column_type,
->     CASE
->       WHEN is_nullable = 'NO' THEN ' not null'
->     ELSE ''
->   END,
->   CASE
->     WHEN extra IS NOT NULL AND extra LIKE 'DEFAULT_GENERATED%'
->       THEN concat(' DEFAULT ',column_default,substr(extra,18))
->     WHEN extra IS NOT NULL THEN concat(' ', extra)
->     ELSE ''
->   END,
->   ',') txt
->   FROM information_schema.columns
->   WHERE table_schema = 'sakila' AND table_name = 'category'
->   ORDER BY ordinal_position
-> ) cols
-> UNION ALL
-> SELECT ')';
-----+
| create_table_statement |
-----+
| CREATE TABLE category ( |
|   category_id tinyint(3) unsigned not null auto_increment, |
|   name varchar(25) not null , |
|   last_update timestamp not null DEFAULT CURRENT_TIMESTAMP |
|     on update CURRENT_TIMESTAMP, |
|   ) |
-----+
5 rows in set (0.00 sec)

```

Мы получили довольно близкий к нужному результат. Нам просто нужно добавить запросы к представлениям `table_constraints` и `key_column_usage`, чтобы получить информацию об ограничении первичного ключа:

```

mysql> SELECT 'CREATE TABLE category (' create_table_statement
-> UNION ALL
-> SELECT cols.txt
-> FROM
->   (SELECT concat(' ',column_name, ' ', column_type,
->     CASE
->       WHEN is_nullable = 'NO' THEN ' not null'
->     ELSE ''
->   END,
->   CASE
->     WHEN extra IS NOT NULL AND extra LIKE 'DEFAULT_GENERATED%'
->       THEN concat(' DEFAULT ',column_default,substr(extra,18))
->     WHEN extra IS NOT NULL THEN concat(' ', extra)
->     ELSE ''

```

```

->     END,
->     ',' ) txt
->   FROM information_schema.columns
-> WHERE table_schema = 'sakila' AND table_name = 'category'
-> ORDER BY ordinal_position
-> ) cols
-> UNION ALL
-> SELECT concat(' constraint primary key ()')
-> FROM information_schema.table_constraints
-> WHERE table_schema = 'sakila' AND table_name = 'category'
->     AND constraint_type = 'PRIMARY KEY'
-> UNION ALL
-> SELECT cols.txt
-> FROM
->   (SELECT concat(CASE WHEN ordinal_position > 1 THEN ' , '
->                      ELSE ' ' END, column_name) txt
->   FROM information_schema.key_column_usage
->   WHERE table_schema = 'sakila' AND table_name = 'category'
->     AND constraint_name = 'PRIMARY'
->   ORDER BY ordinal_position
-> ) cols
-> UNION ALL
-> SELECT ' )'
-> UNION ALL
-> SELECT ')';
+-----+
| create_table_statement |
+-----+
| CREATE TABLE category (
|   category_id tinyint(3) unsigned not null auto_increment,
|   name varchar(25) not null ,
|   last_update timestamp not null DEFAULT CURRENT_TIMESTAMP
|     on update CURRENT_TIMESTAMP,
|   constraint primary key (
|     category_id
|   )
| )
+-----+
8 rows in set (0.02 sec)

```

Чтобы увидеть, правильно ли сформирована инструкция, я вставлю вывод запроса в приглашение mysql (я изменил имя таблицы на category2, чтобы не было коллизий с уже имеющейся таблицей):

```
mysql> CREATE TABLE category2 (
->   category_id tinyint(3) unsigned not null auto_increment,
->   name varchar(25) not null ,
->   last_update timestamp not null DEFAULT CURRENT_TIMESTAMP
->     on update CURRENT_TIMESTAMP,
->   constraint primary key (
```

```
->      category_id  
-> )  
-> );  
Query OK, 0 rows affected (0.61 sec)
```

Инструкция выполнена без ошибок, и теперь у вас есть таблица category2 в базе данных Sakila. Для запроса для создания правильно сформированной инструкции create table для любой таблицы требуется больше работы (например, обработка индексов обработки и ограничений внешних ключей), но это задание я оставлю вам в качестве упражнения.



Если вы используете графический инструмент разработки, такой как Toad, Oracle SQL Developer или MySQL Workbench, вы сможете легко создавать такие сценарии без написания собственных запросов. Но я рассказываю вам о том, как писать свой запрос, на тот пожарный случай, если вы застряли на необитаемом острове и у вас под рукой только клиент командной строки MySQL...

Проверка базы данных

Многие организации обеспечивают окна для обслуживания базы данных, во время которых выполняются работы по изменению объектов существующей базы данных (например, добавление/удаление частей), а также могут быть развернуты новые объекты схемы базы данных и код. После выполнения сценариев обслуживания неплохо бы запустить сценарий проверки, чтобы гарантировать, что новые объекты схемы находятся на месте со всеми соответствующими столбцами, индексами, первичными ключами и т.д. Вот запрос, который возвращает количество столбцов, количество индексов и количество ограничений первичного ключа (0 или 1) для каждой таблицы в схеме Sakila:

```
mysql> SELECT tbl.table_name,  
->   (SELECT count(*) FROM information_schema.columns clm  
->     WHERE clm.table_schema = tbl.table_schema  
->       AND clm.table_name = tbl.table_name) num_columns,  
->   (SELECT count(*) FROM information_schema.statistics sta  
->     WHERE sta.table_schema = tbl.table_schema  
->       AND sta.table_name = tbl.table_name) num_indexes,  
->   (SELECT count(*) FROM information_schema.table_constraints tc  
->     WHERE tc.table_schema = tbl.table_schema  
->       AND tc.table_name = tbl.table_name  
->       AND tc.constraint_type = 'PRIMARY KEY') num_primary_keys  
->   FROM information_schema.tables tbl  
->   WHERE tbl.table_schema = 'sakila'
```

```

-> AND tbl.table_type = 'BASE TABLE'
-> ORDER BY 1;
+-----+
| TABLE_NAME | num_columns | num_indexes | num_primary_keys |
+-----+
| actor      |      4 |      2 |      1 |
| address    |      9 |      3 |      1 |
| category   |      3 |      1 |      1 |
| city       |      4 |      2 |      1 |
| country    |      3 |      1 |      1 |
| customer   |      9 |      7 |      1 |
| film       |     13 |      4 |      1 |
| film_actor |      3 |      3 |      1 |
| film_category |      3 |      3 |      1 |
| film_text  |      3 |      3 |      1 |
| inventory  |      4 |      4 |      1 |
| language   |      3 |      1 |      1 |
| payment    |      7 |      4 |      1 |
| rental     |      7 |      7 |      1 |
| staff      |     11 |      3 |      1 |
| store      |      4 |      3 |      1 |
+-----+
16 rows in set (0.01 sec)

```

Вы можете выполнить эту инструкцию до и после работ, а затем проверить наличие различий между двумя результирующими наборами, прежде чем объявлять выполненные работы успешными.

Динамическая генерация SQL

Некоторые языки, такие как PL/SQL в Oracle и Transact-SQL от Microsoft, являются надмножествами языка SQL, что означает, что они включают в себя инструкции SQL с его грамматикой наряду с обычными процедурными конструкциями, такими как if-then-else, или циклами типа while. Другие языки, такие как Java, включают возможность взаимодействия с реляционной базой данных, но не включают в свою грамматику инструкции SQL, что означает, что все инструкции SQL должны содержаться в виде строк.

Таким образом, большинство серверов реляционных баз данных, включая SQL Server, Oracle Database и MySQL, позволяют отправлять серверу инструкции SQL в виде строк. Отправка строк вместо непосредственного использования интерфейса SQL в общем случае известна как *динамическое выполнение SQL*. Язык Oracle PL/SQL, например, включает команду `execute immediate`, которую можно использовать для отправки строки для выполнения; SQL Server для динамического выполнения инструкций SQL включает системную процедуру `sp_executesql`.

MySQL для динамического выполнения SQL предоставляет инструкции `prepare`, `execute` и `deallocate`. Вот простой пример их применения:

```
mysql> SET @qry =
-> 'SELECT customer_id, first_name, last_name FROM customer';
Query OK, 0 rows affected (0.00 sec)

mysql> PREPARE dynsql1 FROM @qry;
Query OK, 0 rows affected (0.00 sec)
Statement prepared

mysql> EXECUTE dynsql1;
+-----+-----+-----+
| customer_id | first_name | last_name |
+-----+-----+-----+
|      505 | RAFAEL    | ABNEY     |
|      504 | NATHANIEL | ADAM       |
|      36  | KATHLEEN   | ADAMS     |
|      96  | DIANA     | ALEXANDER |
| ...          |           |           |
|      31  | BRENDA    | WRIGHT    |
|      318 | BRIAN     | WYMAN     |
|      402 | LUIS      | YANEZ     |
|      413 | MARVIN    | YEE       |
|      28  | CYNTHIA   | YOUNG     |
+-----+-----+-----+
599 rows in set (0.02 sec)

mysql> DEALLOCATE PREPARE dynsql1;
Query OK, 0 rows affected (0.00 sec)
```

Инструкция `set` просто присваивает строку переменной `qry`, которая затем передается механизму базы данных (для анализа, проверки безопасности и оптимизации) с помощью инструкции `prepare`. После выполнения инструкции с помощью вызова `execute` инструкция должна быть закрыта с использованием команды `deallocate prepare`, которая освобождает все ресурсы базы данных (например, курсоры), которые были использованы во время выполнения.

В следующем примере показано, как можно выполнить запрос, который включает заполнители, значения для которых могут быть указаны во время выполнения:

```
mysql> SET @qry = 'SELECT customer_id, first_name, last_name
FROM customer WHERE customer_id = ?';
Query OK, 0 rows affected (0.00 sec)

mysql> PREPARE dynsql2 FROM @qry;
Query OK, 0 rows affected (0.00 sec)
```

```

Statement prepared

mysql> SET @custid = 9;
Query OK, 0 rows affected (0.00 sec)

mysql> EXECUTE dynsql1 USING @custid;
+-----+-----+-----+
| customer_id | first_name | last_name |
+-----+-----+-----+
|         9 | MARGARET   | MOORE      |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> SET @custid = 145;
Query OK, 0 rows affected (0.00 sec)

mysql> EXECUTE dynsql2 USING @custid;
+-----+-----+-----+
| customer_id | first_name | last_name |
+-----+-----+-----+
|       145 | LUCILLE    | HOLMES     |
+-----+-----+-----+
1 row in set (0.00 sec)

mysql> DEALLOCATE PREPARE dynsql2;
Query OK, 0 rows affected (0.00 sec)

```

В этой последовательности запрос содержит placeholder (символ ? в конце инструкции), так что значение идентификатора клиента может быть указано во время выполнения. Инструкция подготавливается один раз, а затем выполняется дважды, один раз — для идентификатора клиента 9 и второй — для клиента с идентификатором 145, после чего инструкция закрывается.

Вы спрашиваете, какое отношение это все имеет к метаданным? Если вы собираетесь использовать динамический SQL для запроса к таблице, то почему бы не создать строку запроса с использованием метаданных, вместо того чтобы жестко прошивать в коде определение таблицы? В следующем примере генерируется та же динамическая SQL строка, что и в предыдущем примере, но имена столбцов она извлекает из представления `information_schema.columns`:

```

mysql> SELECT concat('SELECT ',
-> concat_ws(',', cols.col1, cols.col2, cols.col3, cols.col4,
->     cols.col5, cols.col6, cols.col7, cols.col8, cols.col9),
->     ' FROM customer WHERE customer_id = ?')
-> INTO @qry
-> FROM
-> (SELECT
->     max(CASE WHEN ordinal_position = 1 THEN column_name

```

```

->      ELSE NULL END) col1,
->      max(CASE WHEN ordinal_position = 2 THEN column_name
->          ELSE NULL END) col2,
->      max(CASE WHEN ordinal_position = 3 THEN column_name
->          ELSE NULL END) col3,
->      max(CASE WHEN ordinal_position = 4 THEN column_name
->          ELSE NULL END) col4,
->      max(CASE WHEN ordinal_position = 5 THEN column_name
->          ELSE NULL END) col5,
->      max(CASE WHEN ordinal_position = 6 THEN column_name
->          ELSE NULL END) col6,
->      max(CASE WHEN ordinal_position = 7 THEN column_name
->          ELSE NULL END) col7,
->      max(CASE WHEN ordinal_position = 8 THEN column_name
->          ELSE NULL END) col8,
->      max(CASE WHEN ordinal_position = 9 THEN column_name
->          ELSE NULL END) col9
->  FROM information_schema.columns
-> WHERE table_schema = 'sakila' AND table_name = 'customer'
-> GROUP BY table_name
-> ) cols;
Query OK, 1 row affected (0.00 sec)

```

mysql> SELECT @qry;

```

+-----+
| @qry |
+-----+
| SELECT customer_id,store_id,first_name,last_name,email,
|   address_id,active,create_date,last_update
| FROM customer WHERE customer_id = ? |
+-----+
1 row in set (0.00 sec)

```

```

mysql> PREPARE dynsql13 FROM @qry;
Query OK, 0 rows affected (0.00 sec)
Statement prepared

```

```

mysql> SET @custid = 45;
Query OK, 0 rows affected (0.00 sec)

```

```

mysql> EXECUTE dynsql13 USING @custid;
+-----+-----+-----+-----+
| customer_id | store_id | first_name | last_name
+-----+-----+-----+-----+
|        45 |         1 | JANET     | PHILLIPS
+-----+-----+-----+-----+
+-----+-----+-----+
| email                         | address_id | active
+-----+-----+-----+
| JANET.PHILLIPS@sakilacustomer.org |        49 |     1
+-----+-----+-----+

```

```
+-----+-----+
| create_date | last_update |
+-----+-----+
| 2006-02-14 22:04:36 | 2006-02-15 04:57:20 |
+-----+
1 row in set (0.00 sec)
```

```
mysql> DEALLOCATE PREPARE dynsql13;
Query OK, 0 rows affected (0.00 sec)
```

Запрос получает первые девять столбцов таблицы `customer`, создает строку с использованием функций `concat` и `concat_ws` и присваивает ее переменной `qry`. Выполняется строка запроса так же, как и ранее.



Как правило, лучше генерировать такой запрос с использованием процедурного языка, который включает циклические конструкции, такого как Java, PL/SQL, Transact-SQL или MySQL Stored Procedure Language. Я просто хотел продемонстрировать пример на чистом SQL, поэтому мне пришлось ограничить количество извлекаемых столбцов некоторым разумным числом, в этом примере равным девятыи.

Проверьте свои знания

Предлагаемые здесь упражнения призваны закрепить понимание вами методанных. Ответы к упражнениям представлены в приложении Б.

УПРАЖНЕНИЕ 15.1

Напишите запрос, который перечисляет все индексы в схеме `Sakila`. Не забудьте включить в результаты имена таблиц.

УПРАЖНЕНИЕ 15.2

Напишите запрос, генерирующий вывод, который можно было бы использовать для создания всех индексов таблицы `sakila.customer`. Вывод должен иметь вид

```
"ALTER TABLE <таблица> ADD INDEX <индекс> (<список_столбцов>)"
```

Аналитические функции

Объемы данных растут в ошеломляющем темпе, так что организации сталкиваются с проблемами их хранения, не говоря уже об их адекватной обработке. В то время как анализ данных традиционно выполняется за пределами серверов баз данных, с использованием таких специализированных инструментов или языков, как Excel, R и Python, язык SQL включает в себя набор функций, полезных для аналитической обработки данных. Если вам нужно сгенерировать рейтинги для определения лучших десяти продавцов вашей компании или если вы генерируете финансовый отчет для клиента и должны рассчитать средние отклонения за три месяца, можете использовать встроенные аналитические функции SQL для выполнения расчетов такого вида.

Концепции аналитических функций

После того как сервер базы данных выполнил все шаги, необходимые для выполнения запроса, включая соединения, фильтрацию, группирование и сортировку, результирующий набор создан и готов к возвращению вызывающей программе/коду. Представьте, что вы можете приостановить выполнение запроса в этот момент и прогуляться по результирующему набору, пока он все еще находится в памяти. Какие виды анализа данных вы можете захотеть выполнить? Если ваш результирующий набор содержит данные продаж, возможно, вы захотите сформировать рейтинги продавцов или регионов либо рассчитать процент различия в продажах в разные промежутки времени. Генерируя результаты для финансового отчета, вы можете захотеть рассчитать промежуточные итоги для каждого раздела отчета и общий итог для всего отчета. С помощью аналитических функций вы можете сделать все это и многое другое. Прежде чем погрузиться в детали, в следующих подразделах будут рассмотрены механизмы, используемые некоторыми из наиболее часто используемых аналитических функций.

Окна данных

Допустим, вы написали запрос, который генерирует ежемесячные итоги продаж в течение определенного периода времени. Например, следующий запрос подсчитывает итоговую сумму общих ежемесячных платежей за прокат фильмов за период с мая по август 2005 года:

```
mysql> SELECT quarter(payment_date) quarter,
->     monthname(payment_date) month_nm,
->     sum(amount) monthly_sales
->   FROM payment
-> WHERE year(payment_date) = 2005
-> GROUP BY quarter(payment_date), monthname(payment_date);
+-----+-----+-----+
| quarter | month_nm | monthly_sales |
+-----+-----+-----+
|      2 | May      |      4824.43 |
|      2 | June     |      9631.88 |
|      3 | July     |    28373.89 |
|      3 | August   |    24072.13 |
+-----+-----+-----+
4 rows in set (0.13 sec)
```

Глядя на результаты, можно увидеть, что самый высокий ежемесячный итог за четыре месяца был в июле, а за второй квартал — в июне. Но для того, чтобы определить эти самые высокие значения программно, нужно добавить к каждой строке дополнительные столбцы, показывающие максимальные значения за квартал и за весь период. Вот предыдущий запрос, но с добавленными новыми столбцами для вычисления указанных значений:

```
mysql> SELECT quarter(payment_date) quarter,
->     monthname(payment_date) month_nm,
->     sum(amount) monthly_sales,
->     max(sum(amount))
->       over () max_overall_sales,
->     max(sum(amount))
->       over (partition by quarter(payment_date)) max_qrtr_sales
->   FROM payment
-> WHERE year(payment_date) = 2005
-> GROUP BY quarter(payment_date), monthname(payment_date);
+-----+-----+-----+-----+
| quarter | month_nm | monthly_sales | max_overall_sales | max_qrtr_sales |
+-----+-----+-----+-----+
```

```

|      2 | May      |        4824.43 |      28373.89 |      9631.88 |
|      2 | June     |        9631.88 |      28373.89 |      9631.88 |
|      3 | July     |       28373.89 |      28373.89 |      28373.89 |
|      3 | August   |      24072.13 |      28373.89 |      28373.89 |
+-----+-----+-----+-----+
4 rows in set (0.09 sec)

```

Аналитические функции, используемые для генерации этих дополнительных столбцов, группируют строки в два разных набора: один набор, содержащий все строки в том же квартале, а второй — содержащий все строки. Для этого типа анализа аналитические функции включают возможность группирования строк в окна, которые эффективно разделяют данные для использования аналитической функцией без изменения общего результирующего набора. Окна определяются с использованием предложения over в сочетании с необязательной конструкцией partition by. В предыдущем запросе обе аналитические функции включают предложение over, но первое из них пустое, указывающее, что окно должно включать в себя весь набор результатов, тогда как второе указывает, что окно должно включать только строки из одного и того же квартала. Окно данных может содержать от одной строки до всех строк в результирующем наборе. Различные аналитические функции могут определять различные окна данных.

Локализованная сортировка

Вы можете не только разделить свой результирующий набор на окна данных для аналитических функций, но и указать порядок сортировки. Например, чтобы определить рейтинг для каждого месяца, в котором значение 1 отдается месяцу, имеющему самые высокие продажи, нужно указать, какой столбец (или столбцы) используется для вычисления рейтинга:

```

mysql> SELECT quarter(payment_date) quarter,
->    monthname(payment_date) month_nm,
->    sum(amount) monthly_sales,
->    rank() over (order by sum(amount) desc) sales_rank
->   FROM payment
->  WHERE year(payment_date) = 2005
->  GROUP BY quarter(payment_date), monthname(payment_date)
->  ORDER BY 1, month(payment_date);
+-----+-----+-----+-----+
| quarter | month_nm | monthly_sales | sales_rank |
+-----+-----+-----+-----+
|      2 | May      |        4824.43 |        4 |

```

```

|      2 | June     |      9631.88 |          3 |
|      3 | July     |     28373.89 |          1 |
|      3 | August   |     24072.13 |          2 |
+-----+-----+-----+
4 rows in set (0.00 sec)

```

Этот запрос включает в себя вызов функции `rank` (о которой будет рассказано в следующем разделе) и указывает, что для генерации рейтинга используется сумма столбца `amount` со значениями, отсортированными в порядке убывания. Таким образом, месяц, имеющий самые высокие продажи (в данном случае это июль), получит рейтинг 1.



Множественные предложения `order by`

В предыдущем примере имеется два предложения `order by`, одно — в конце запроса для определения, как должен быть отсортирован результирующий набор, и одно — в рамках функции `rank`, чтобы определить, как должны быть распределены значения рейтингов. Неудачно то, что одно и то же предложение используется для разных целей, но помните, что даже если вы используете аналитические функции с одним или несколькими предложениями `order by`, но хотите, чтобы результат был отсортирован определенным образом, вам все равно необходимо предложение `order by` в конце вашего запроса.

В некоторых случаях вы захотите использовать в одном и том же вызове аналитической функции как предложение `partition by`, так и предложение `order by`. Например, предыдущий пример можно изменить так, чтобы предоставлять поквартальные рейтинги, а не единый рейтинг для всего результирующего набора:

```

mysql> SELECT quarter(payment_date) quarter,
->    monthname(payment_date) month_nm,
->    sum(amount) monthly_sales,
->    rank() over (partition by quarter(payment_date)
->                  order by sum(amount) desc) qtr_sales_rank
->   FROM payment
->  WHERE year(payment_date) = 2005
->  GROUP BY quarter(payment_date), monthname(payment_date)
->  ORDER BY 1, month(payment_date);
+-----+-----+-----+
| quarter | month_nm | monthly_sales | qtr_sales_rank |
+-----+-----+-----+
|      2 | May     |      4824.43 |          2 |
|      2 | June   |      9631.88 |          1 |
+-----+-----+-----+

```

```
|      3 | July      |      28373.89 |          1 |
|      3 | August    |      24072.13 |          2 |
+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

Эти примеры были разработаны, чтобы проиллюстрировать использование предложения over; в следующих разделах будут подробно описаны различные аналитические функции.

Ранжирование

Люди любят ранжировать. Если вы посетите свои любимые сайты, посвященные новостям, спорту, путешествиям и так далее, то увидите заголовки, подобные следующим.

- Лучшие 10 мест для отпуска
- Лучшие фильмы XX века
- Наихудшие столицы Европы
- 50 лучших вузов страны

Компании также любят генерировать рейтинги, но для более практических целей. Знание того, что продается лучше/хуже остального, или какие регионы дают наименьший/наибольший доход, помогает организациям принимать стратегические решения.

Функции ранжирования

В стандарте SQL есть несколько функций ранжирования с разными подходами к обработке одинаковых данных.

`row_number`

Возвращает для каждой строки уникальное число с произвольно называемым рейтингом для одинаковых данных

`rank`

Возвращает при одинаковых данных один и тот же рейтинг с соответствующими пропусками в общем рейтинге

`dense_rank`

Возвращает при одинаковых данных один и тот же рейтинг без пропусков в общем рейтинге

Давайте рассмотрим пример, чтобы понять различия. Пусть отдел продаж хочет найти 10 лучших клиентов, чтобы предложить им взять фильм

напрокат бесплатно. Следующий запрос определяет количество прокатов фильмов для каждого клиента и сортирует результаты в порядке убывания:

```
mysql> SELECT customer_id, count(*) num_rentals
-> FROM rental
-> GROUP BY customer_id
-> ORDER BY 2 desc;
+-----+-----+
| customer_id | num_rentals |
+-----+-----+
|      148 |        46 |
|      526 |        45 |
|     236 |        42 |
|     144 |        42 |
|       75 |        41 |
|     469 |        40 |
|     197 |        40 |
|     137 |        39 |
|     468 |        39 |
|     178 |        39 |
|     459 |        38 |
|     410 |        38 |
|       5 |        38 |
|     295 |        38 |
|     257 |        37 |
|     366 |        37 |
|     176 |        37 |
|     198 |        37 |
|     267 |        36 |
|     439 |        36 |
|     354 |        36 |
|     348 |        36 |
|     380 |        36 |
|       29 |        36 |
|     371 |        35 |
|     403 |        35 |
|      21 |        35 |
| ...
|     136 |        15 |
|     248 |        15 |
|     110 |        14 |
|     281 |        14 |
|      61 |        14 |
|     318 |        12 |
+-----+-----+
```

599 rows in set (0.16 sec)

В результирующем наборе видно, что как третий, так и четвертый клиенты брали напрокат по 42 фильма. Должны ли они оба получить один и тот же рейтинг 3? И если да, то какой рейтинг должен иметь клиент, бравший

напрокат 41 фильм: рейтинг 4, или мы должны пропустить одно значение и дать ему рейтинг 5? Чтобы увидеть, как каждая функция обрабатывает соппадения при назначении рейтинга, в следующий запрос добавлены еще три столбца, каждый из которых использует свою функцию ранжирования:

```
mysql> SELECT customer_id, count(*) num_rentals,
->     row_number() over (order by count(*) desc) row_number_rnk,
->     rank() over (order by count(*) desc) rank_rnk,
->     dense_rank() over (order by count(*) desc) dense_rank_rnk
-> FROM rental
-> GROUP BY customer_id
-> ORDER BY 2 desc;
```

customer_id	num_rentals	row_number_rnk	rank_rnk	dense_rank_rnk
148	46	1	1	1
526	45	2	2	2
144	42	3	3	3
236	42	4	3	3
75	41	5	5	4
197	40	6	6	5
469	40	7	6	5
468	39	10	8	6
137	39	8	8	6
178	39	9	8	6
5	38	11	11	7
295	38	12	11	7
410	38	13	11	7
459	38	14	11	7
198	37	16	15	8
257	37	17	15	8
366	37	18	15	8
176	37	15	15	8
348	36	21	19	9
354	36	22	19	9
380	36	23	19	9
439	36	24	19	9
29	36	19	19	9
267	36	20	19	9
50	35	26	25	10
506	35	37	25	10
368	35	32	25	10
91	35	27	25	10
371	35	33	25	10
196	35	28	25	10
373	35	34	25	10
204	35	29	25	10
381	35	35	25	10
273	35	30	25	10
21	35	25	25	10

	403	35	36	25	10	
	274	35	31	25	10	
	66	34	42	38	11	
...						
	136	15	594	594	30	
	248	15	595	594	30	
	110	14	597	596	31	
	281	14	598	596	31	
	61	14	596	596	31	
	318	12	599	599	32	

599 rows in set (0.01 sec)

Третий столбец использует функцию `row_number` для назначения уникального рейтинга каждой строке без учета совпадений. Каждой из 599 строк присваивается значение рейтинга от 1 до 599 с рейтингом, произвольно назначаемым клиентам, которые имеют одинаковое количество взятых напрокат фильмов. Следующие же два столбца назначают в случае совпадения один и тот же рейтинг, но различаются наличием пропуска в использованных рангах. Взглянув на ряд 5 результирующего набора, можно увидеть, что функция `rank` пропускает значение 4 и присваивает следующему клиенту ранг 5, тогда как функция `dense_rank` присваивает значение 4.

Вернемся к первоначальному запросу. Как бы вы определили 10 лучших клиентов? Есть три возможных решения.

- Использовать функцию `row_number`, чтобы определить клиентов с рейтингом от 1 до 10, что приводит ровно к 10 клиентам в данном примере, но в других случаях может исключить ряд клиентов, имеющих такое же количество взятых напрокат фильмов, как и клиент с рейтингом 10.
- Использовать функцию `rank` для определения клиентов с рейтингом не более 10, что в нашем случае также даст ровно 10 клиентов.
- Использовать функцию `dense_rank` определения клиентов с рейтингом не более 10, что в нашем случае дает список из 37 клиентов.

Если в вашем результирующем наборе нет совпадений, то подходит любая из этих функций, но для множества ситуаций лучшим вариантом может быть функция `rank`.

Генерация нескольких рейтингов

Пример в предыдущем разделе генерирует один рейтинг для всего множества клиентов. Но что если вы хотите создать несколько наборов рейтингов

для одного и того же результирующего набора? Чтобы расширить предыдущий пример, пусть отдел продаж решает предложить бесплатный прокат фильмов для лучших пяти клиентов каждый месяц. Чтобы сгенерировать такие данные, можно добавить к запросу столбец rental_month:

```
mysql> SELECT customer_id,
->   monthname(rental_date) rental_month,
->   count(*) num_rentals
->  FROM rental
-> GROUP BY customer_id, monthname(rental_date)
-> ORDER BY 2, 3 desc;
+-----+-----+-----+
| customer_id | rental_month | num_rentals |
+-----+-----+-----+
|      119 | August      |      18 |
|       15 | August      |      18 |
|      569 | August      |      18 |
|     148 | August      |      18 |
|     141 | August      |      17 |
|      21 | August      |      17 |
|     266 | August      |      17 |
|     418 | August      |      17 |
|     410 | August      |      17 |
|     342 | August      |      17 |
|     274 | August      |      16 |
| ...
|      281 | August      |      2 |
|     318 | August      |      1 |
|      75 | February    |      3 |
|     155 | February    |      2 |
|     175 | February    |      2 |
|     516 | February    |      2 |
|     361 | February    |      2 |
|     269 | February    |      2 |
|     208 | February    |      2 |
|      53 | February    |      2 |
| ...
|      22 | February    |      1 |
|     472 | February    |      1 |
|     148 | July        |      22 |
|     102 | July        |      21 |
|     236 | July        |      20 |
|      75 | July        |      20 |
|      91 | July        |      19 |
|      30 | July        |      19 |
|      64 | July        |      19 |
|     137 | July        |      19 |
| ...
|     339 | May         |      1 |
|     485 | May         |      1 |
+-----+-----+-----+
```

```

|      116 | May          |      1 |
|      497 | May          |      1 |
|      180 | May          |      1 |
+-----+-----+-----+
2466 rows in set (0.02 sec)

```

Для того чтобы каждый месяц создавать новый набор рейтингов, нужно добавить в функцию rank нечто, описывающее, как разделить результирующий набор на различные окна данных (в нашем случае — месяцы). Это делается с использованием предложения partition by, добавляемого в предложение over:

```

mysql> SELECT customer_id,
    ->   monthname(rental_date) rental_month,
    ->   count(*) num_rentals,
    ->   rank() over (partition by monthname(rental_date)
    ->     order by count(*) desc) rank_rnk
    -> FROM rental
    -> GROUP BY customer_id, monthname(rental_date)
    -> ORDER BY 2, 3 desc;
+-----+-----+-----+-----+
| customer_id | rental_month | num_rentals | rank_rnk |
+-----+-----+-----+-----+
|      569 | August       |      18 |      1 |
|      119 | August       |      18 |      1 |
|      148 | August       |      18 |      1 |
|       15 | August       |      18 |      1 |
|      141 | August       |      17 |      5 |
|      410 | August       |      17 |      5 |
|      418 | August       |      17 |      5 |
|       21 | August       |      17 |      5 |
|      266 | August       |      17 |      5 |
|      342 | August       |      17 |      5 |
|      144 | August       |      16 |     11 |
|      274 | August       |      16 |     11 |
| ...
|      164 | August       |      2 |     596 |
|      318 | August       |      1 |     599 |
|       75 | February     |      3 |      1 |
|      457 | February     |      2 |      2 |
|       53 | February     |      2 |      2 |
|      354 | February     |      2 |      2 |
|      352 | February     |      1 |     24 |
|      373 | February     |      1 |     24 |
|      148 | July         |     22 |      1 |
|      102 | July         |     21 |      2 |
|      236 | July         |     20 |      3 |
|       75 | July         |     20 |      3 |
|       91 | July         |     19 |      5 |
|      354 | July         |     19 |      5 |

```

```

| 30 | July      | 19 | 5 |
| 64 | July      | 19 | 5 |
| 137 | July     | 19 | 5 |
| 526 | July     | 19 | 5 |
| 366 | July     | 19 | 5 |
| 595 | July     | 19 | 5 |
| 469 | July     | 18 | 13 |
| ...
| 457 | May      | 1 | 347 |
| 356 | May      | 1 | 347 |
| 481 | May      | 1 | 347 |
| 10  | May      | 1 | 347 |
+-----+-----+-----+
2466 rows in set (0.03 sec)

```

Взглянув на результаты, можно увидеть, что рейтинги для каждого месяца сбрасываются и начинаются с 1. Чтобы генерировать необходимые для отдела продаж результаты (5 лучших клиентов каждого месяца), можно просто обернуть предыдущий запрос в подзапрос и добавить условие фильтрации для исключения любых строк с рейтингом выше пяти:

```

SELECT customer_id, rental_month, num_rentals,
       rank_rnk ranking
FROM
  (SELECT customer_id,
         monthname(rental_date) rental_month,
         count(*) num_rentals,
         rank() over (partition by monthname(rental_date)
                       order by count(*) desc) rank_rnk
   FROM rental
   GROUP BY customer_id, monthname(rental_date)
  ) cust_rankings
WHERE rank_rnk <= 5
ORDER BY rental_month, num_rentals desc, rank_rnk;

```

Поскольку аналитические функции могут быть использованы только в предложении select, то, если нужно выполнить фильтрацию или группировку на основе результатов аналитической функции, часто приходится прибегать к вложенным запросам.

Функции отчетности

Наряду с генерацией рейтинга еще одно распространенное использование аналитических функций — поиск выбросов данных (например, минимального или максимального значения) для генерации сумм или средних по всему набору данных. Для этих видов аналитики используются агрегатные функции (min, max, avg, sum, count), но вместо того, чтобы использовать их с

предложением group by, они соединяются с предложением over. Вот пример, который генерирует ежемесячные и общие итоги для всех платежей размером не менее 10 долларов:

```
mysql> SELECT monthname(payment_date) payment_month,
->     amount,
->     sum(amount)
->     over (partition by monthname(payment_date)) monthly_total,
->     sum(amount) over () grand_total
->   FROM payment
-> WHERE amount >= 10
-> ORDER BY 1;
```

payment_month	amount	monthly_total	grand_total
August	10.99	521.53	1262.86
August	11.99	521.53	1262.86
August	10.99	521.53	1262.86
August	10.99	521.53	1262.86
...			
August	10.99	521.53	1262.86
August	10.99	521.53	1262.86
August	10.99	521.53	1262.86
July	10.99	519.53	1262.86
July	10.99	519.53	1262.86
July	10.99	519.53	1262.86
July	10.99	519.53	1262.86
...			
July	10.99	519.53	1262.86
July	10.99	519.53	1262.86
July	10.99	519.53	1262.86
June	10.99	165.85	1262.86
June	10.99	165.85	1262.86
June	10.99	165.85	1262.86
June	10.99	165.85	1262.86
June	10.99	165.85	1262.86
June	10.99	165.85	1262.86
June	10.99	165.85	1262.86
June	10.99	165.85	1262.86
June	10.99	165.85	1262.86
June	10.99	165.85	1262.86
June	10.99	165.85	1262.86
May	10.99	55.95	1262.86
May	10.99	55.95	1262.86
May	10.99	55.95	1262.86
May	10.99	55.95	1262.86

```
| May           | 11.99 |      55.95 |     1262.86 |
+-----+-----+-----+
114 rows in set (0.01 sec)
```

Столбец `grand_total` содержит то же значение (1262,86 доллара) для каждой строки, потому что предложение `over` пустое (что указывает, что суммирование проводится по всему результирующему набору). Столбец `monthly_total`, однако, содержит для каждого месяца другое значение, поскольку имеется предложение `partition by`, указывающее, что результирующий набор будет разделен на несколько окон данных (по одному для каждого месяца).

Хотя включение столбца, такого как `grand_total`, с одним и тем же значением для каждой строки выглядит нелепо, столбцы такого типа могут использоваться и для расчетов, как показано в следующем запросе:

```
mysql> SELECT monthname(payment_date) payment_month,
->     sum(amount) month_total,
->     round(sum(amount) / sum(sum(amount)) over (),
->            * 100, 2) pct_of_total
->   FROM payment
->  GROUP BY monthname(payment_date);
+-----+-----+-----+
| payment_month | month_total | pct_of_total |
+-----+-----+-----+
| May           |    4824.43 |       7.16 |
| June          |    9631.88 |      14.29 |
| July          |   28373.89 |      42.09 |
| August         |  24072.13 |      35.71 |
| February       |     514.18 |       0.76 |
+-----+-----+-----+
5 rows in set (0.04 sec)
```

Этот запрос рассчитывает общие платежи для каждого месяца, суммируя столбец `amount`, а затем рассчитывает для каждого месяца процент от общих платежей, используя сумму ежемесячных значений в качестве знаменателя при расчете.

Функции отчетности также могут быть использованы для сравнений, таких как следующий запрос, который использует выражение `case`, чтобы определить, является ли ежемесячный итог максимальным, минимальным или находящимся где-то посередине:

```
mysql> SELECT monthname(payment_date) payment_month,
->     sum(amount) month_total,
->     CASE sum(amount)
->       WHEN max(sum(amount)) over () THEN 'Highest'
->       WHEN min(sum(amount)) over () THEN 'Lowest'
```

```

->      ELSE 'Middle'
-> END descriptor
-> FROM payment
-> GROUP BY monthname(payment_date);
+-----+-----+-----+
| payment_month | month_total | descriptor |
+-----+-----+-----+
| May           |    4824.43 | Middle     |
| June          |   9631.88 | Middle     |
| July          | 28373.89 | Highest    |
| August         | 24072.13 | Middle     |
| February       |    514.18 | Lowest    |
+-----+-----+-----+
5 rows in set (0.04 sec)

```

Столбец `descriptor` действует как функция квазирейтинга, в том смысле, что помогает идентифицировать наибольшие/наименьшие значения в наборе строк.

Рамки окон

Как было описано ранее в главе, окна данных для аналитических функций определяются с использованием предложения `partition by`, которое позволяет группировать строки с общими значениями. Но что если требуется еще более тонкий контроль над строками для включения их в окно данных? Например, возможно, вы хотите создать “скользящее” итоговое значение, вычисляемое с начала года и до текущей строки. Для такого рода расчетов можно включить подпредложение “рамки” для точного определения, какие именно строки включаются в окно данных. Вот запрос, который суммирует платежи за каждую неделю и включает функцию отчетности для вычисления суммы:

```

mysql> SELECT yearweek(payment_date) payment_week,
->      sum(amount) week_total,
->      sum(sum(amount))
->      over (order by yearweek(payment_date)
->              rows unbounded preceding) rolling_sum
-> FROM payment
-> GROUP BY yearweek(payment_date)
-> ORDER BY 1;
+-----+-----+-----+
| payment_week | week_total | rolling_sum |
+-----+-----+-----+
| 200521 | 2847.18 | 2847.18 |
| 200522 | 1977.25 | 4824.43 |
| 200524 | 5605.42 | 10429.85 |
| 200525 | 4026.46 | 14456.31 |

```

```

| 200527 | 8490.83 | 22947.14 |
| 200528 | 5983.63 | 28930.77 |
| 200530 | 11031.22 | 39961.99 |
| 200531 | 8412.07 | 48374.06 |
| 200533 | 10619.11 | 58993.17 |
| 200534 | 7909.16 | 66902.33 |
| 200607 | 514.18 | 67416.51 |
+-----+
11 rows in set (0.04 sec)

```

Выражение для столбца `rolling_sum` включает предложение `rows unbounded preceding` для определения окна данных от начала результирующего набора и до текущей строки, включая ее. Окно данных состоит из одной строки для первой строки в результирующем наборе, две строки — для второй и т.д. Значение для последней строки представляет собой сумму всего результирующего набора.

Вместе с такими “скользящими” суммами можно рассчитать и соответствующие средние значения. Вот запрос, который рассчитывает трехнедельное “скользящее” среднее для общих платежей:

```

mysql> SELECT yearweek(payment_date) payment_week,
->     sum(amount) week_total,
->     avg(sum(amount))
->         over (order by yearweek(payment_date)
->             rows between 1 preceding and 1 following) rolling_3wk_avg
-> FROM payment
-> GROUP BY yearweek(payment_date)
-> ORDER BY 1;
+-----+-----+-----+
| payment_week | week_total | rolling_3wk_avg |
+-----+-----+-----+
| 200521 | 2847.18 | 2412.215000 |
| 200522 | 1977.25 | 3476.616667 |
| 200524 | 5605.42 | 3869.710000 |
| 200525 | 4026.46 | 6040.903333 |
| 200527 | 8490.83 | 6166.973333 |
| 200528 | 5983.63 | 8501.893333 |
| 200530 | 11031.22 | 8475.640000 |
| 200531 | 8412.07 | 10020.800000 |
| 200533 | 10619.11 | 8980.113333 |
| 200534 | 7909.16 | 6347.483333 |
| 200607 | 514.18 | 4211.670000 |
+-----+
11 rows in set (0.03 sec)

```

Столбец `rolling_3wk_avg` определяет окно данных, состоящее из текущей строки, предыдущей и следующей строк. Таким образом, окно данных состоит из трех строк, за исключением первых и последних строк, для

которых окно данных состоит только из двух строк (так как для первой строки нет предшествующей, а для последней строки — последующей).

Указание количества строк для окна данных во многих случаях работает нормально, но если в данных есть пробелы, то можно попробовать другой подход. В предыдущем результирующем наборе, например, есть данные для недель 200521, 200522 и 200524, но нет данных для недели 200523. Если вы хотите указать интервал даты, а не количество строк, то можете указать *диапазон* для вашего окна данных, как показано в следующем запросе:

```
mysql> SELECT date(payment_date), sum(amount),
->     avg(sum(amount)) over (order by date(payment_date)
->         range between interval 3 day preceding
->             and interval 3 day following) 7_day_avg
-> FROM payment
-> WHERE payment_date BETWEEN '2005-07-01' AND '2005-09-01'
-> GROUP BY date(payment_date)
-> ORDER BY 1;
+-----+-----+-----+
| date(payment_date) | sum(amount) | 7_day_avg |
+-----+-----+-----+
| 2005-07-05 | 128.73 | 1603.740000 |
| 2005-07-06 | 2131.96 | 1698.166000 |
| 2005-07-07 | 1943.39 | 1738.338333 |
| 2005-07-08 | 2210.88 | 1766.917143 |
| 2005-07-09 | 2075.87 | 2049.390000 |
| 2005-07-10 | 1939.20 | 2035.628333 |
| 2005-07-11 | 1938.39 | 2054.076000 |
| 2005-07-12 | 2106.04 | 2014.875000 |
| 2005-07-26 | 160.67 | 2046.642500 |
| 2005-07-27 | 2726.51 | 2206.244000 |
| 2005-07-28 | 2577.80 | 2316.571667 |
| 2005-07-29 | 2721.59 | 2388.102857 |
| 2005-07-30 | 2844.65 | 2754.660000 |
| 2005-07-31 | 2868.21 | 2759.351667 |
| 2005-08-01 | 2817.29 | 2795.662000 |
| 2005-08-02 | 2726.57 | 2814.180000 |
| 2005-08-16 | 111.77 | 1973.837500 |
| 2005-08-17 | 2457.07 | 2123.822000 |
| 2005-08-18 | 2710.79 | 2238.086667 |
| 2005-08-19 | 2615.72 | 2286.465714 |
| 2005-08-20 | 2723.76 | 2630.928571 |
| 2005-08-21 | 2809.41 | 2659.905000 |
| 2005-08-22 | 2576.74 | 2649.728000 |
| 2005-08-23 | 2523.01 | 2658.230000 |
+-----+-----+-----+
24 rows in set (0.03 sec)
```

Столбец `7_day_avg` определяет диапазон ± 3 дня и будет включать только строки, значения `payment_date` которых попадают в пределы этого диапазона. Например, для расчета 2005-08-16 включаются только значения для 08-16, 08-17, 08-18 и 08-19, так как для трех предшествующих дат (от 08-13 до 08-15) строк нет.

Запаздывание и опережение

Наряду с вычислением сумм и средних по окну данных еще одна распространенная задача отчетности включает в себя сравнение значений от одной строки к другой. Например, если вы генерируете ежемесячные суммы продаж, то можете попросить создать столбец, показывающий процентное отличие от предыдущего месяца, что потребует способа получения ежемесячных продаж из предыдущей строки. Это может быть достигнуто с помощью функции `lag`, которая извлекает значение столбца из предыдущей строки в результирующем наборе, или функции `lead`, которая получает значение столбца из следующей строки. Вот пример с использованием обеих упомянутых функций:

```
mysql> SELECT yearweek(payment_date) payment_week,
->     sum(amount) week_total,
->     lag(sum(amount), 1)
->         over (order by yearweek(payment_date)) prev_wk_tot,
->     lead(sum(amount), 1)
->         over (order by yearweek(payment_date)) next_wk_tot
-> FROM payment
-> GROUP BY yearweek(payment_date)
-> ORDER BY 1;
+-----+-----+-----+-----+
| payment_week | week_total | prev_wk_tot | next_wk_tot |
+-----+-----+-----+-----+
|    200521 |   2847.18 |      NULL |   1977.25 |
|    200522 |   1977.25 |   2847.18 |   5605.42 |
|    200524 |   5605.42 |   1977.25 |   4026.46 |
|    200525 |   4026.46 |   5605.42 |   8490.83 |
|    200527 |   8490.83 |   4026.46 |   5983.63 |
|    200528 |   5983.63 |   8490.83 |  11031.22 |
|    200530 |  11031.22 |   5983.63 |   8412.07 |
|    200531 |   8412.07 |  11031.22 |  10619.11 |
|    200533 |  10619.11 |   8412.07 |   7909.16 |
|    200534 |   7909.16 |  10619.11 |   514.18 |
|    200607 |    514.18 |   7909.16 |      NULL |
+-----+-----+-----+-----+
11 rows in set (0.03 sec)
```

Взглянув на результаты, мы видим, что недельная сумма 8490,83 для недели 200527 появляется в столбце `next_wk_tot` для недели 200525, а также в

столбце `prev_wk_tot` — для недели 200528. Так как в результирующем наборе нет строки, предшествующей неделе 200521, функция `lag` генерирует значение `null` для первой строки. Аналогично значение, создаваемое функцией `lead` для последней строки в результирующем наборе, также равно `null`. Обе функции допускают наличие необязательного второго параметра (который по умолчанию равен 1) для указания количества строк до/после текущей строки для получения значения столбца.

Вот как вы можете использовать функцию `lag`, чтобы создать разницу в процентах по отношению к предыдущей неделе:

```
mysql> SELECT yearweek(payment_date) payment_week,
->     sum(amount) week_total,
->     round((sum(amount) - lag(sum(amount), 1)
->             over (order by yearweek(payment_date)))
->           / lag(sum(amount), 1)
->             over (order by yearweek(payment_date)))
->           * 100, 1) pct_diff
->   FROM payment
-> GROUP BY yearweek(payment_date)
-> ORDER BY 1;
+-----+-----+-----+
| payment_week | week_total | pct_diff |
+-----+-----+-----+
| 200521 | 2847.18 | NULL      |
| 200522 | 1977.25 | -30.6    |
| 200524 | 5605.42 | 183.5    |
| 200525 | 4026.46 | -28.2    |
| 200527 | 8490.83 | 110.9    |
| 200528 | 5983.63 | -29.5    |
| 200530 | 11031.22 | 84.4     |
| 200531 | 8412.07 | -23.7    |
| 200533 | 10619.11 | 26.2     |
| 200534 | 7909.16 | -25.5    |
| 200607 | 514.18 | -93.5    |
+-----+-----+-----+
11 rows in set (0.07 sec)
```

Сравнение значений из разных строк в одном и том же результирующем наборе — обычная практика в системах отчетности, поэтому вы, скорее всего, найдете множество применений для функций `lag` и `lead`.

Конкатенация значений в столбце

Есть еще одна важная функция, о которой надо рассказать и которая, хотя технически и не является аналитической, работает с группами строк в окне данных. Функция `group_concat` используется для превращения набора

значений столбца в единую строку с разделителями, что является удобным способом денормализации вашего результирующего набора для генерации документов XML или JSON. Вот пример того, как эта функция может быть использована для генерации списка актеров (разделенных запятыми) для каждого фильма:

```
mysql> SELECT f.title,
->   group_concat(a.last_name order by a.last_name
->   separator ', ') actors
->   FROM actor a
->   INNER JOIN film_actor fa
->   ON a.actor_id = fa.actor_id
->   INNER JOIN film f
->   ON fa.film_id = f.film_id
->   GROUP BY f.title
->   HAVING count(*) = 3;
+-----+-----+
| title          | actors           |
+-----+-----+
| ANNIE IDENTITY | GRANT, KEITEL, MCQUEEN |
| ANYTHING SAVANNAH | MONROE, SWANK, WEST |
| ARK RIDGEMONT  | BAILEY, DEGENERES, GOLDBERG |
| ARSENIC INDEPENDENCE | ALLEN, KILMER, REYNOLDS |
| ...
| WHISPERER GIANT | BAILEY, PECK, WALKEN |
| WIND PHANTOM    | BALL, DENCH, GUINNESS |
| ZORRO ARK       | DEGENERES, MONROE, TANDY |
+-----+
119 rows in set (0.04 sec)
```

Этот запрос группирует строки по названию фильма и включает только те фильмы, в которых указаны ровно три актера. Функция `group_concat` действует как особый тип агрегирующей функции, которая собирает фамилии всех актеров, появляющихся в фильме, в одну строку. Если вы используете SQL Server, то можете использовать для этого функцию `string_agg`, а пользователи Oracle могут использовать функцию `listagg`.

Проверьте свои знания

Предлагаемые здесь упражнения призваны закрепить понимание вами аналитических функций. Ответы к упражнениям представлены в приложении Б.

Для всех упражнений этого раздела используйте набор данных из таблицы `Sales_Fact`:

```

Sales_Fact
+-----+-----+-----+
| year_no | month_no | tot_sales |
+-----+-----+-----+
| 2019 | 1 | 19228 |
| 2019 | 2 | 18554 |
| 2019 | 3 | 17325 |
| 2019 | 4 | 13221 |
| 2019 | 5 | 9964 |
| 2019 | 6 | 12658 |
| 2019 | 7 | 14233 |
| 2019 | 8 | 17342 |
| 2019 | 9 | 16853 |
| 2019 | 10 | 17121 |
| 2019 | 11 | 19095 |
| 2019 | 12 | 21436 |
| 2020 | 1 | 20347 |
| 2020 | 2 | 17434 |
| 2020 | 3 | 16225 |
| 2020 | 4 | 13853 |
| 2020 | 5 | 14589 |
| 2020 | 6 | 13248 |
| 2020 | 7 | 8728 |
| 2020 | 8 | 9378 |
| 2020 | 9 | 11467 |
| 2020 | 10 | 13842 |
| 2020 | 11 | 15742 |
| 2020 | 12 | 18636 |
+-----+-----+-----+
24 rows in set (0.00 sec)

```

УПРАЖНЕНИЕ 16.1

Напишите запрос, который извлекает каждую строку из Sales_Fact и добавляет столбец для генерации рейтинга на основе значений столбца tot_sales. Самое высокое значение должно получить рейтинг 1, а самое низкое — рейтинг 24.

УПРАЖНЕНИЕ 16.2

Измените запрос из предыдущего упражнения так, чтобы генерировалось два набора рейтингов от 1 до 12: один — для 2019 года и один — для 2020 года.

УПРАЖНЕНИЕ 16.3

Напишите запрос, который извлекает все данные за 2020 год, и включите столбец, который будет содержать значение tot_sales для предыдущего месяца.

Работа с большими базами данных

Во времена, когда реляционные базы данных только появились на свет, емкость жесткого диска измеряли в мегабайтах и базами данных в целом было легко управлять просто потому, что они не могли быть очень большими. Сегодня емкость отдельного жесткого диска выросла до 15 Тбайт, современный дисковый массив может хранить более 4 Пбайт данных, а хранение в облаке, по сути, не имеет границ. Хотя реляционные базы постоянно сталкиваются с новыми проблемами, связанными с ростом данных, имеются такие стратегии, как секционирование, кластеризация и шардинг (разделение данных между разными серверами), которые позволяют продолжать компаниям использовать реляционные базы данных, разделяя данные по различным хранилищам и серверам. Чтобы обрабатывать огромные объемы данных, другие компании решили перейти на платформы больших данных, как, например, Hadoop. В этой главе рассматриваются некоторые из упомянутых стратегий с акцентом на методиках масштабирования реляционных баз данных.

Секционирование

Когда именно таблица базы данных становится “слишком большой”? Если вы зададите этот вопрос десяти различным архитекторам/администраторам/разработчикам баз данных, то вы, вероятно, получите десять разных ответов. Однако большинство согласится, что перечисленные ниже задачи становятся все более трудными и/или затратными в смысле машинного времени по достижении таблицей размера в несколько миллионов строк.

- Выполнение запросов, требующих полного сканирования таблицы
- Создание/восстановление индексов
- Архивирование/удаление данных
- Создание статистик таблиц/индексов

- Перенос таблицы (например, в другое табличное пространство)
- Резервное копирование базы данных

Эти задачи могут начинаться в качестве рутинных, когда база данных невелика, но становятся трудоемкими по мере накопления большего количества данных, а затем превращаются в проблематичные или попросту невозможные из-за ограниченности административных временных окон. Лучший способ предотвратить возникновение административных проблем в будущем заключается в том, чтобы *секционировать*, или *разделять*, большие таблицы на части, или разделы, при первоначальном создании таблиц (хотя таблицы могут быть разделены и позже, легче сделать это изначально). Административные задачи могут выполняться для отдельных разделов, часто параллельно, а некоторые задачи могут полностью пропустить один или несколько разделов.

Концепции секционирования

Секционирование таблиц было введено в конце 1990-х годов в Oracle, но с тех пор каждый из основных серверов баз данных получил возможность секционирования таблиц и индексов. Когда таблица секционируется, создаются две или более таблиц-разделов, каждая из которых имеет такое же определение, но с неперекрывающимися подмножествами данных. Например, таблица, содержащая данные продаж, может быть разделена по месяцам, используя столбец, содержащий дату продажи, или разделена географически с использованием кода страны/области.

После секционирования сама таблица становится виртуальной концепцией. Данные хранятся в разных разделах, и любые индексы построены для данных в разделах таблицы. Однако пользователи базы данных все еще могут взаимодействовать с таблицей, не зная, что на самом деле она была секционирована. Это похоже на концепцию представлений в том, что пользователи взаимодействуют с объектами схемы, которые являются интерфейсами, а не фактическими таблицами. Хотя каждый раздел должен иметь одинаковое определение схемы (столбцы, типы столбцов и т.д.), имеется несколько административных функциональных возможностей, которые могут различаться для каждого раздела.

- Разделы могут храниться в разных табличных пространствах, которые могут находиться в различных физических хранилищах.
- Разделы могут быть сжаты с использованием различных схем сжатия.

- Локальные индексы (о них чуть позже) для некоторых разделов могут отсутствовать.
- Статистика таблицы для некоторых разделов может быть заморожена, при этом периодически обновляясь для других.
- Отдельные разделы могут быть закреплены в памяти или храниться во флеш-хранилище.

Таким образом, секционирование таблицы обеспечивает высокую гибкость хранения и администрирования данных, при этом предоставляя сообществу пользователей простоту работы с одной таблицей.

Секционирование таблицы

Схема секционирования, доступная в большинстве реляционных баз данных, представляет собой *горизонтальное секционирование*, которое назначает каждую отдельную строку ровно одному разделу. Таблицы также могут быть секционированы *вертикально*, когда выполняется присвоение наборов столбцов разным разделам, но это необходимо делать вручную. При секционировании таблицы горизонтально необходимо выбрать *ключ секционирования*, который является столбцом, значения которого используются для назначения строки определенному разделу. В большинстве случаев ключ секционирования таблицы состоит из одного столбца, а *функция секционирования* применяется к этому столбцу, чтобы определить, в какой раздел следует помещать ту или иную строку.

Секционирование индекса

Если ваша секционированная таблица имеет индексы, вы можете выбрать, должен ли конкретный индекс оставаться неразделенным, известным как *глобальный индекс*, или быть разбитым на части, так что каждый раздел будет иметь собственный индекс, который называется *локальным индексом*. Глобальные индексы объединяют все разделы таблицы и полезны для запросов, которые не указывают значение ключа секционирования. Пусть, например, ваша таблица секционирована по столбцу `sale_date` и пользователь выполняет следующий запрос:

```
SELECT sum(amount) FROM sales WHERE geo_region_cd = 'US'
```

Поскольку этот запрос не включает в себя условие фильтра для столбца `sale_date`, сервер должен будет выполнять поиск в каждом разделе, чтобы

найти общие продажи в США. Если же на столбце geo_region_cd построен глобальный индекс, то сервер сможет использовать его, чтобы быстро найти все строки, содержащие искомые продажи.

Методы секционирования

Хотя каждый сервер базы данных имеет свои уникальные возможности секционирования, в следующих трех разделах описаны распространенные методы секционирования, доступные у большинства серверов.

Секционирование по диапазону

Секционирование по диапазону стало первым методом секционирования, который был реализован, и он по-прежнему является одним из наиболее популярных. Хотя секционирование по диапазону можно использовать для нескольких различных типов столбцов, наиболее распространенным его использованием является секционирование таблиц по диапазонам даты. Например, таблица sales может быть секционирована с использованием столбца sale_date так, что данные для каждой недели будут храниться в отдельном разделе:

```
mysql> CREATE TABLE sales
->   (sale_id INT NOT NULL,
->    cust_id INT NOT NULL,
->    store_id INT NOT NULL,
->    sale_date DATE NOT NULL,
->    amount DECIMAL(9,2)
->  )
-> PARTITION BY RANGE (yearweek(sale_date))
-> (PARTITION s1 VALUES LESS THAN (202002),
->  PARTITION s2 VALUES LESS THAN (202003),
->  PARTITION s3 VALUES LESS THAN (202004),
->  PARTITION s4 VALUES LESS THAN (202005),
->  PARTITION s5 VALUES LESS THAN (202006),
->  PARTITION s999 VALUES LESS THAN (MAXVALUE)
-> );
```

Query OK, 0 rows affected (1.78 sec)

Эта инструкция создает шесть разных разделов: по одному для каждой из первых пяти недель 2020 года и шестой раздел с именем S999, чтобы хранить все строки за пределами указанных недель. Для этой таблицы выражение yearweek(sale_date) используется в качестве функции секционирования, а столбец sale_date служит ключом секционирования. Чтобы увидеть межданные, связанные с секционированными таблицами, можно использовать таблицу partitions в базе данных information_schema:

```

mysql> SELECT partition_name, partition_method, partition_expression
-> FROM information_schema.partitions
-> WHERE table_name = 'sales'
-> ORDER BY partition_ordinal_position;
+-----+-----+-----+
| PARTITION_NAME | PARTITION_METHOD | PARTITION_EXPRESSION |
+-----+-----+-----+
| s1             | RANGE          | yearweek(`sale_date`,0) |
| s2             | RANGE          | yearweek(`sale_date`,0) |
| s3             | RANGE          | yearweek(`sale_date`,0) |
| s4             | RANGE          | yearweek(`sale_date`,0) |
| s5             | RANGE          | yearweek(`sale_date`,0) |
| s999           | RANGE          | yearweek(`sale_date`,0) |
+-----+-----+-----+
6 rows in set (0.00 sec)

```

Одной из административных задач, которые необходимо будет выполнить для таблицы sales, является генерация новых разделов для хранения будущих данных (чтобы хранить данные, добавленные в раздел s999). Разные базы данных обрабатывают это задание по-разному, но в MySQL, чтобы разделить раздел S999 на три части, можно использовать предложение `reorganize partition` в команде `alter table`:

```

ALTER TABLE sales REORGANIZE PARTITION s999 INTO
(PARTITION s6 VALUES LESS THAN (202007),
 PARTITION s7 VALUES LESS THAN (202008),
 PARTITION s999 VALUES LESS THAN (MAXVALUE)
);

```

Если снова выполнить предыдущий запрос метаданных, то теперь можно увидеть восемь разделов:

```

mysql> SELECT partition_name, partition_method, partition_expression
-> FROM information_schema.partitions
-> WHERE table_name = 'sales'
-> ORDER BY partition_ordinal_position;
+-----+-----+-----+
| PARTITION_NAME | PARTITION_METHOD | PARTITION_EXPRESSION |
+-----+-----+-----+
| s1             | RANGE          | yearweek(`sale_date`,0) |
| s2             | RANGE          | yearweek(`sale_date`,0) |
| s3             | RANGE          | yearweek(`sale_date`,0) |
| s4             | RANGE          | yearweek(`sale_date`,0) |
| s5             | RANGE          | yearweek(`sale_date`,0) |
| s6             | RANGE          | yearweek(`sale_date`,0) |
| s7             | RANGE          | yearweek(`sale_date`,0) |
| s999           | RANGE          | yearweek(`sale_date`,0) |
+-----+-----+-----+
8 rows in set (0.00 sec)

```

Теперь добавим в таблицу несколько строк:

```
mysql> INSERT INTO sales
-> VALUES
-> (1, 1, 1, '2020-01-18', 2765.15),
-> (2, 3, 4, '2020-02-07', 5322.08);
Query OK, 2 rows affected (0.18 sec)
Records: 2 Duplicates: 0 Warnings: 0
```

В таблице теперь есть две строки, но в какие разделы они вставлены? Чтобы это выяснить, давайте используем подпредложение `partition` предложения `from`, чтобы подсчитать количество строк в каждом разделе:

```
mysql> SELECT concat('# of rows in S1 = ', count(*))
->          partition_rowcount
-> FROM sales PARTITION (s1) UNION ALL
-> SELECT concat('# of rows in S2 = ', count(*))
->          partition_rowcount
-> FROM sales PARTITION (s2) UNION ALL
-> SELECT concat('# of rows in S3 = ', count(*))
->          partition_rowcount
-> FROM sales PARTITION (s3) UNION ALL
-> SELECT concat('# of rows in S4 = ', count(*))
->          partition_rowcount
-> FROM sales PARTITION (s4) UNION ALL
-> SELECT concat('# of rows in S5 = ', count(*))
->          partition_rowcount
-> FROM sales PARTITION (s5) UNION ALL
-> SELECT concat('# of rows in S6 = ', count(*))
->          partition_rowcount
-> FROM sales PARTITION (s6) UNION ALL
-> SELECT concat('# of rows in S7 = ', count(*))
->          partition_rowcount
-> FROM sales PARTITION (s7) UNION ALL
-> SELECT concat('# of rows in S999 = ', count(*))
->          partition_rowcount
-> FROM sales PARTITION (s999);
+-----+
| partition_rowcount      |
+-----+
| # of rows in S1 = 0    |
| # of rows in S2 = 1    |
| # of rows in S3 = 0    |
| # of rows in S4 = 0    |
| # of rows in S5 = 1    |
| # of rows in S6 = 0    |
| # of rows in S7 = 0    |
| # of rows in S999 = 0  |
+-----+
8 rows in set (0.00 sec)
```

Результаты показывают, что одна строка была вставлена в раздел S2, а другая — в раздел S5. Возможность запрашивать определенный раздел требует знания схемы секционирования, поэтому маловероятно, что пользователи будут выполнять запросы такого вида; они обычно используются для административных видов деятельности.

Секционирование по списку

Если столбец, выбранный в качестве ключа секционирования, содержит, например, коды штатов (CA, TX, VA и т.д.), валюты (USD, EUR, JPY и т.д.) или иной набор перечислимых значений, можно использовать секционирование по списку, которое позволяет указать, какие значения будут назначены каждому разделу. Например, пусть таблица sales включает столбец geo_region_cd, который содержит следующие значения:

geo_region_cd	description
US_NE	United States North East
US_SE	United States South East
US_MW	United States Mid West
US_NW	United States North West
US_SW	United States South West
CAN	Canada
MEX	Mexico
EUR_E	Eastern Europe
EUR_W	Western Europe
CHN	China
JPN	Japan
IND	India
KOR	Korea

13 rows in set (0.00 sec)

Вы можете сгруппировать эти значения в географические регионы и создать раздел для каждого из них:

```
mysql> CREATE TABLE sales
    -> (sale_id INT NOT NULL,
    -> cust_id INT NOT NULL,
    -> store_id INT NOT NULL,
    -> sale_date DATE NOT NULL,
    -> geo_region_cd VARCHAR(6) NOT NULL,
    -> amount DECIMAL(9,2)
    -> )
    -> PARTITION BY LIST COLUMNS (geo_region_cd)
    -> (PARTITION NORTHAMERICA VALUES IN ('US_NE','US_SE','US_MW',
    -> 'US_NW','US_SW','CAN','MEX'))
```

```
-> PARTITION EUROPE VALUES IN ('EUR_E','EUR_W'),
-> PARTITION ASIA VALUES IN ('CHN','JPN','IND')
-> );
Query OK, 0 rows affected (1.13 sec)
```

В таблице есть три раздела и каждый из них включает в себя набор из двух или более значений geo_region_cd. Теперь давайте добавим в таблицу несколько строк:

```
mysql> INSERT INTO sales
-> VALUES
-> (1, 1, 1, '2020-01-18', 'US_NE', 2765.15),
-> (2, 3, 4, '2020-02-07', 'CAN', 5322.08),
-> (3, 6, 27, '2020-03-11', 'KOR', 4267.12);
ERROR 1526 (HY000): Table has no partition for value from column_list1
```

Похоже, возникла проблема — в сообщении об ошибке указано, что один из географических кодов региона не был назначен ни одному разделу. Глядя на инструкцию create table, я вижу, что забыл добавить Корею к разделу asia. Это можно исправить с помощью инструкции alter table:

```
mysql> ALTER TABLE sales REORGANIZE PARTITION ASIA INTO
-> (PARTITION ASIA VALUES IN ('CHN','JPN','IND', 'KOR'));
Query OK, 0 rows affected (1.28 sec)
Records: 0 Duplicates: 0 Warnings: 0
```

Давайте проверим метаданные, чтобы быть уверенными, что все получилось правильно:

```
mysql> SELECT partition_name, partition_expression,
-> partition_description
-> FROM information_schema.partitions
-> WHERE table_name = 'sales'
-> ORDER BY partition_ordinal_position;
+-----+-----+
|PARTITION_NAME|PARTITION_EXPRESSION|PARTITION_DESCRIPTION
+-----+-----+
|NORTHAMERICA |`geo_region_cd`    |'US_NE','US_SE','US_MW',
|              |                           |'US_NW','US_SW','CAN','MEX'
|EUROPE        |`geo_region_cd`    |'EUR_E','EUR_W'
|ASIA          |`geo_region_cd`    |'CHN','JPN','IND','KOR'
+-----+-----+
3 rows in set (0.00 sec)
```

Корея действительно была добавлена в раздел asia, и теперь вставка данных не вызывает никаких проблем:

¹ В таблице нет раздела для значения из column_list.

```
mysql> INSERT INTO sales
-> VALUES
->   (1, 1, 1, '2020-01-18', 'US_NE', 2765.15),
->   (2, 3, 4, '2020-02-07', 'CAN', 5322.08),
->   (3, 6, 27, '2020-03-11', 'KOR', 4267.12);
Query OK, 3 rows affected (0.26 sec)
Records: 3 Duplicates: 0 Warnings: 0
```

В то время как секционирование по диапазону позволяет разделу s999 перехватывать и хранить любые строки, которые не подходят для других разделов, секционирование по списку такой возможности не обеспечивает. Таким образом, если в некоторый момент вам может потребоваться добавить еще одно значение столбца (например, компания начнет продажи в Австралии), вам нужно будет изменить определение секционирования перед добавлением в таблицу строк с новым значением.

Секционирование по хешу

Если столбец ключа секционирования не обеспечивает секционирование по диапазону или по списку, есть третий вариант, который увеличивает равномерность распределения строк по множеству разделов. Сервер делает это, применяя функцию хеширования к значению столбца, и этот тип секционирования называется (что не удивительно) *секционированием по хешу*. В отличие от секционирования по списку, при котором столбец, выбранный в качестве ключа секционирования, должен содержать только небольшое количество значений, секционирование по хешу лучше всего работает тогда, когда столбец ключа секционирования содержит большое количество различных значений. Вот еще одна версия таблицы sales, но с четырьмя хеш-разделами, генерируемыми путем хеширования значений в столбце cust_id:

```
mysql> CREATE TABLE sales
->   (sale_id INT NOT NULL,
->    cust_id INT NOT NULL,
->    store_id INT NOT NULL,
->    sale_date DATE NOT NULL,
->    amount DECIMAL(9,2)
->  )
-> PARTITION BY HASH (cust_id)
-> PARTITIONS 4
->   (PARTITION H1,
->    PARTITION H2,
->    PARTITION H3,
->    PARTITION H4
->  );
Query OK, 0 rows affected (1.50 sec)
```

Строки при добавлении в таблицу sales будут равномерно распределены по всем четырем разделам, которые я назвал H1, H2, H3 и H4. Для того чтобы увидеть, насколько хорошо это работает, давайте добавим 16 строк, каждая с другим значением столбца cust_id:

```
mysql> INSERT INTO sales
-> VALUES
-> (1,1,1,'2020-01-18',1.1), (2,3,4,'2020-02-07',1.2),
-> (3,17,5,'2020-01-19',1.3), (4,23,2,'2020-02-08',1.4),
-> (5,56,1,'2020-01-20',1.6), (6,77,5,'2020-02-09',1.7),
-> (7,122,4,'2020-01-21',1.8), (8,153,1,'2020-02-10',1.9),
-> (9,179,5,'2020-01-22',2.0), (10,244,2,'2020-02-11',2.1),
-> (11,263,1,'2020-01-23',2.2), (12,312,4,'2020-02-12',2.3),
-> (13,346,2,'2020-01-24',2.4), (14,389,3,'2020-02-13',2.5),
-> (15,472,1,'2020-01-25',2.6), (16,502,1,'2020-02-14',2.7);
Query OK, 16 rows affected (0.19 sec)
Records: 16 Duplicates: 0 Warnings: 0
```

Если функция хеширования равномерно распределяет строки, в идеале мы должны увидеть в каждом разделе по четыре строки:

```
mysql> SELECT concat('# of rows in H1 = ', count(*))
->          partition_rowcount
-> FROM sales PARTITION (h1) UNION ALL
-> SELECT concat('# of rows in H2 = ', count(*))
->          partition_rowcount
-> FROM sales PARTITION (h2) UNION ALL
-> SELECT concat('# of rows in H3 = ', count(*))
->          partition_rowcount
-> FROM sales PARTITION (h3) UNION ALL
-> SELECT concat('# of rows in H4 = ', count(*))
->          partition_rowcount
-> FROM sales PARTITION (h4);
+-----+
| partition_rowcount |
+-----+
| # of rows in H1 = 4 |
| # of rows in H2 = 5 |
| # of rows in H3 = 3 |
| # of rows in H4 = 4 |
+-----+
4 rows in set (0.00 sec)
```

Учитывая, что вставлено только 16 строк, это весьма хорошее распределение; по мере увеличения количества строк каждый раздел должен содержать количество строк, близкое к 25% от общего количества — при наличии достаточно большого количества различных значений в столбце cust_id.

Композитное секционирование

Если вам нужен тонкий контроль над тем, как данные распределяются по разделам таблицы, можете использовать *композитное секционирование*, позволяющее использовать два разных типа секционирования для одной и той же таблицы. При композитном секционировании первый метод определяет разделы, а второй — *подразделы*. Вот пример, который вновь использует таблицу sales, теперь с применением как секционирования по диапазону, так и хеша:

```
mysql> CREATE TABLE sales
->   (sale_id INT NOT NULL,
->    cust_id INT NOT NULL,
->    store_id INT NOT NULL,
->    sale_date DATE NOT NULL,
->    amount DECIMAL(9,2)
->  )
-> PARTITION BY RANGE (yearweek(sale_date))
-> SUBPARTITION BY HASH (cust_id)
->   (PARTITION s1 VALUES LESS THAN (202002)
->     (SUBPARTITION s1_h1,
->      SUBPARTITION s1_h2,
->      SUBPARTITION s1_h3,
->      SUBPARTITION s1_h4),
->    PARTITION s2 VALUES LESS THAN (202003)
->     (SUBPARTITION s2_h1,
->      SUBPARTITION s2_h2,
->      SUBPARTITION s2_h3,
->      SUBPARTITION s2_h4),
->    PARTITION s3 VALUES LESS THAN (202004)
->     (SUBPARTITION s3_h1,
->      SUBPARTITION s3_h2,
->      SUBPARTITION s3_h3,
->      SUBPARTITION s3_h4),
->    PARTITION s4 VALUES LESS THAN (202005)
->     (SUBPARTITION s4_h1,
->      SUBPARTITION s4_h2,
->      SUBPARTITION s4_h3,
->      SUBPARTITION s4_h4),
->    PARTITION s5 VALUES LESS THAN (202006)
->     (SUBPARTITION s5_h1,
->      SUBPARTITION s5_h2,
->      SUBPARTITION s5_h3,
->      SUBPARTITION s5_h4),
->    PARTITION s999 VALUES LESS THAN (MAXVALUE)
->     (SUBPARTITION s999_h1,
->      SUBPARTITION s999_h2,
->      SUBPARTITION s999_h3,
->      SUBPARTITION s999_h4)
```

```
-> );  
Query OK, 0 rows affected (9.72 sec)
```

Имеется 6 разделов, каждый из которых имеет по 4 подраздела, в общей сложности — 24 подраздела. Теперь давайте заново вставим 16 строк рядов из более раннего примера для секционирования по хешу:

```
mysql> INSERT INTO sales  
-> VALUES  
-> (1,1,1,'2020-01-18',1.1), (2,3,4,'2020-02-07',1.2),  
-> (3,17,5,'2020-01-19',1.3), (4,23,2,'2020-02-08',1.4),  
-> (5,56,1,'2020-01-20',1.6), (6,77,5,'2020-02-09',1.7),  
-> (7,122,4,'2020-01-21',1.8), (8,153,1,'2020-02-10',1.9),  
-> (9,179,5,'2020-01-22',2.0), (10,244,2,'2020-02-11',2.1),  
-> (11,263,1,'2020-01-23',2.2), (12,312,4,'2020-02-12',2.3),  
-> (13,346,2,'2020-01-24',2.4), (14,389,3,'2020-02-13',2.5),  
-> (15,472,1,'2020-01-25',2.6), (16,502,1,'2020-02-14',2.7);
```

```
Query OK, 16 rows affected (0.22 sec)  
Records: 16 Duplicates: 0 Warnings: 0
```

Запрашивая таблицу sales, вы можете извлечь данные из одного из разделов; в таком случае вы извлекаете данные из четырех подразделов, связанных с разделом:

```
mysql> SELECT *  
-> FROM sales PARTITION (s3);  
+-----+-----+-----+-----+-----+  
| sale_id | cust_id | store_id | sale_date | amount |  
+-----+-----+-----+-----+-----+  
|      5 |      56 |        1 | 2020-01-20 |   1.60 |  
|     15 |     472 |        1 | 2020-01-25 |   2.60 |  
|      3 |      17 |        5 | 2020-01-19 |   1.30 |  
|      7 |     122 |        4 | 2020-01-21 |   1.80 |  
|     13 |     346 |        2 | 2020-01-24 |   2.40 |  
|      9 |     179 |        5 | 2020-01-22 |   2.00 |  
|     11 |     263 |        1 | 2020-01-23 |   2.20 |  
+-----+-----+-----+-----+-----+  
7 rows in set (0.00 sec)
```

Поскольку таблица разделена на подразделы, можно также получить данные из одного отдельного подраздела:

```
mysql> SELECT *  
-> FROM sales PARTITION (s3_h3);  
+-----+-----+-----+-----+-----+  
| sale_id | cust_id | store_id | sale_date | amount |  
+-----+-----+-----+-----+-----+  
|      7 |     122 |        4 | 2020-01-21 |   1.80 |  
|     13 |     346 |        2 | 2020-01-24 |   2.40 |  
+-----+-----+-----+-----+-----+  
2 rows in set (0.00 sec)
```

Этот запрос извлекает данные только из подраздела s3_h3 раздела s3.

Преимущества секционирования

Одним из основных преимуществ секционирования является то, что вам, возможно, придется взаимодействовать только с несколькими разделами, а не со всей таблицей в целом. Например, если ваша таблица секционирована по диапазону с использованием столбца `sales_date` и вы выполняете запрос, который включает в себя условие фильтра, такое как

```
WHERE sales_date BETWEEN '2019-12-01' AND '2020-01-15'
```

то сервер проверит метаданные таблицы, чтобы определить, какие разделы на самом деле следует включить. Эта концепция называется *отсечением разделов*, и это одно из наибольших преимуществ секционирования таблиц.

Точно так же, если вы выполняете запрос, который включает в себя соединение с секционированной таблицей, и запрос включает условие для столбца секционирования, то сервер может исключить любой раздел, который не содержит данных, относящихся к запросу (что известно как *пораздельное соединение*). Этот подход подобен отсечению разделов в том, что в обработку включаются только те разделы, которые содержат необходимые запросу данные.

С административной точки зрения одним из основных преимуществ секционирования является возможность быстрого удаления данных, которые больше не нужны. Например, пусть финансовые данные нужно хранить в течение семи лет. Если таблица была секционирована на основании дат транзакций, то любые разделы, содержащие данные старше семи лет, могут быть удалены. Еще одно административное преимущество секционированных таблиц — способность выполнять обновления в нескольких разделах одновременно, что может значительно сократить время, необходимое для работы со всеми строками таблицы.

Кластеризация

При наличии хранилища достаточного объема в сочетании с разумной стратегией секционирования в одной реляционной базе данных можно хранить очень много информации. Но что если вам нужно обеспечить одновременную работу тысяч пользователей или сгенерировать десятки тысяч отчетов во время ночного цикла? Даже если у вас есть хранилище данных достаточного размера, вам может не хватать вычислительной мощности процессоров, памяти или пропускной способности сети при использовании

одного сервера. Один из потенциальных ответов на эту проблему — *кластеризация*, которая позволяет нескольким серверам действовать как единая база данных.

Хотя имеется несколько различных архитектур кластеризации, в этом обсуждении для упрощения я имею в виду конфигурацию с общим диском/общим кешем, где каждый сервер кластера имеет доступ ко всем дискам, а кешированные на одном сервере данные могут быть доступны любым другим серверам в кластере. При наличии такого типа архитектуры сервер приложений может соединяться с любым из серверов баз данных в кластере с автоматическим переподключением к другому серверу кластера в случае сбоя. С восьмисерверным кластером вы должны иметь возможность одновременно работать с очень большим количеством пользователей и связанных с ними запросами/отчетами/рабочими местами.

Из коммерческих поставщиков баз данных лидером в этом сегменте является Oracle. Многие крупнейшие компании мира используют платформу Oracle Exadata для размещения очень больших баз данных с одновременным доступом тысяч пользователей. Тем не менее даже эта платформа не соответствует потребностям таких крупнейших компаний, как Google, Facebook, Amazon и др.

Шардинг

Допустим, вас наняли в качестве архитектора данных в новую компанию социальной сети. Вам говорят, что ожидается примерно один миллиард пользователей, каждый из которых будет генерировать в день в среднем 3,7 сообщения, и эти данные должны быть доступны в течение неопределенного срока. После небольших прик遁ок вы понимаете, что исчерпаете самую большую доступную платформу реляционной базы данных менее чем за год. Чтобы обеспечить работоспособность, имеется возможность разделять не только отдельные таблицы, но и всю базу данных. Этот подход, известный как *шардинг* (sharding), разделяет данные по нескольким базам данных (имеющимся *осколками* (shard)), так что он похож на секционирование таблиц, но в большем масштабе и с гораздо большей сложностью. Применяя эту стратегию для социальной сети, вы могли бы решить реализовать 100 отдельных баз данных, каждая из которых принимает данные примерно для 10 миллионов пользователей.

Шардинг — сложная тема, а так как эта книга лишь вводная, я воздержусь от описания этого подхода в деталях. Упомяну лишь несколько вопросов, которые должны быть решены.

- Нужно выбрать *ключ шардинга*, который является значением, используемым для определения того, к какой базе данных должно быть выполнено подключение.
- В то время как большие таблицы будут разделены на части, с отдельными строками, назначенными тому или иному осколку, меньшие таблицы могут реплицироваться во все осколки. Также должна быть определена стратегия того, как могут быть изменены ссылочные данные, чтобы изменения распространялись на все осколки.
- Если отдельные осколки становятся слишком большими (например, социальная сеть вырастает до двух миллиардов пользователей), вам понадобится план добавления дополнительных осколков и перераспределения данных между ними.
- При необходимости внесения изменений в схему требуется стратегия для развертывания изменений по всем осколкам, чтобы все схемы оставались синхронизированными.
- Если логика приложений должна получать доступ к данным, хранящимся в двух или более осколках, нужна стратегия того, как запросить несколько баз данных, а также как реализовать транзакции, распределенные по нескольким базам данных.

Если это кажется слишком сложным, то это потому, что так и есть... И к концу 2000-х годов многие компании начали искать новые подходы к этой проблеме. В следующем разделе рассмотрены другие стратегии обработки очень больших массивов данных, лежащие полностью за пределами царства реляционных баз данных.

Большие данные

Взвесив все плюсы и минусы шардинга, вы (архитектор данных большой социальной сети) решаете рассмотреть другие подходы к проблеме. Вместо того чтобы придумывать свой путь, вы пытаетесь извлечь выгоду из обзора работы, проделанной другими компаниями, занимающимися большими объемами данных, такими как Amazon, Google, Facebook и Twitter. Набор технологий, развитых этими (и другими) компаниями, стал известен как *большие*

данные — модный термин, который имеет несколько возможных определений. Один из способов определения границ больших данных называют “3V”.

Объем (Volume)

В этом контексте объем обычно означает миллиарды или триллионы точек данных.

Скорость (Velocity)

Это мера того, насколько быстро поступают данные.

Разнообразие (Variety)

Это означает, что данные не всегда структурированы (как в строках и столбцах реляционных баз данных) и могут быть не структурированы, например электронные письма, видео, фотографии, аудиофайлы и т.д.

Таким образом, один из способов охарактеризовать большие данные — считать таковыми любую систему, предназначенную для обработки огромного количества данных различных форматов, поступающих в быстром темпе. В следующих разделах кратко описаны некоторые технологии больших данных, которые развивались в течение последних 15 лет или около того.

Hadoop

Hadoop лучше всего описать как экосистему или набор технологий и инструментов, работающих совместно. Некоторые из основных компонентов Hadoop включают следующее.

Распределенная файловая система Hadoop (HDFS)

Как подразумевает название, HDFS позволяет управлять файлами на большом количестве серверов.

MapReduce

Эта технология обрабатывает большое количество структурированных и неструктурированных данных, разбивая задачи на многие маленькие части, которые можно выполнять параллельно на многих серверах.

YARN

Это диспетчер ресурсов и планировщик заданий для HDFS.

Вместе эти технологии позволяют хранить и обрабатывать файлы на сотнях или даже тысячах серверов, действующих как единая логическая система. Запросы данных с использованием MapReduce, в отличие от широко

применяемого Hadoop, обычно требуют участия программиста, что привело к разработке нескольких интерфейсов SQL, включая Hive, Impala и Drill.

NoSQL и базы данных документов

В реляционной базе данных данные обычно должны соответствовать заранее определенной схеме, состоящей из таблиц, которые состоят из столбцов, содержащих числа, строки, даты и т.д. Но что делать, если структура данных неизвестна заранее или если она известна, но часто меняется? Для многих компаний ответ заключается в том, чтобы объединить как определение данных, так и схему в документы с использованием формата наподобие XML или JSON, а затем хранить эти документы в базе данных. При этом различные типы данных могут храниться в одной и той же базе данных без необходимости модификаций схемы. Это облегчает хранение, но переносит нагрузку на запросы и аналитические инструменты, чтобы данные, хранящиеся в документах, имели смысл.

Базы данных документов являются подмножеством того, что называется базами данных NoSQL, которые обычно хранят данные с использованием простого механизма “ключ-значение”. Например, используя такую базу данных документов, как MongoDB, можно использовать идентификатор клиента в качестве ключа для хранения документа JSON, содержащего все данные клиента. Чтобы придать смысл хранимым данным, другие пользователи могут читать схему, сохраненную в документе.

Облачные вычисления

До появления больших данных большинство компаний должны были создать собственные центры обработки данных для размещения баз данных, серверов веб и приложений, используемых предприятием. С появлением облачных вычислений вы, по сути, можете перенести ваш центр обработки данных на внешние платформы, такие как Amazon Web Services (AWS), Microsoft Azure или Google Cloud. Одним из самых больших преимуществ размещения ваших служб в облаке является немедленная масштабируемость, которая позволяет увеличивать или уменьшать вычислительную мощность, необходимую для работы ваших служб. Эти платформы особенно любят стартапы, потому что они могут начать писать код, не тратя предварительно деньги на закупку серверов, хранения, сетей или лицензий на программное обеспечение.

Поскольку мы рассматриваем базы данных, перечислим некоторые предложения баз данных и аналитики AWS.

- Реляционные базы данных (MySQL, Aurora, PostgreSQL, MariaDB, Oracle и SQL Server)
- База данных в памяти (ElastiCache)
- База данных для складирования данных (Redshift)
- База данных NoSQL (DynamoDB)
- База данных документов (DocumentDB)
- База данных графов (Neptune)
- База данных временных рядов (TimeStream)
- Hadoop (EMR)
- Озера данных (Lake Formation)

Довольно легко увидеть, что в то время как в ландшафте баз данных до середины 2000-х годов доминировали реляционные базы данных, в настоящее время компании комбинируют в работе различные платформы, так что что реляционные базы данных со временем могут стать менее популярными.

Заключение

Базы данных становятся все больше, но в то же время технологии хранения, кластеризации и секционирования становятся все более надежными. Работа с огромным количеством данных может быть довольно сложной независимо от используемого технологического стека. Независимо от того, что вы используете (реляционные базы данных, большие платформы данных или множество серверов баз данных), язык SQL развивается таким образом, чтобы облегчить поиск данных в базах данных, построенных на различных технологиях. Этот вопрос и будет темой последней главы этой книги, в которой будет показано использование механизма SQL для запроса данных, хранящихся в нескольких форматах.

SQL и большие данные

Хотя большая часть материала этой книги охватывает различные функциональные возможности языка SQL при использовании реляционной базы данных, такой как MySQL, сам ландшафт данных за последнее десятилетие существенно изменился, и SQL тоже вынужден меняться, чтобы удовлетворять потребности нынешних быстро развивающихся сред. Многие организации, которые всего несколько лет назад использовали исключительно реляционные базы данных, теперь хранят данные в кластерах и нереляционных базах данных. Идет постоянный поиск способов обработки и понимания постоянно растущих объемов данных, и тот факт, что эти данные теперь разбросаны среди нескольких хранилищ данных, возможно в облаке, делает этот поиск непростой задачей.

SQL используется миллионами пользователей и интегрирован в тысячи приложений, а потому имеет смысл найти пути использовать SQL для работы с этими данными. За несколько последних лет появились новые инструменты, которые обеспечивают доступ SQL к структурированным, полуструктурным и неструктурированным данным, такие как Presto, Apache Drill и Toad Data Point. В этой главе рассматривается один из этих инструментов, Apache Drill, и демонстрируется, как для отчетности и анализа могут быть собраны вместе данные разных форматов, хранящиеся на разных серверах.

Введение в Apache Drill

Для SQL-доступа к данным, хранящимся в Hadoop, Spark и облачных распределенных файловых системах, были разработаны многочисленные инструменты и интерфейсы. Примеры включают Hive, который был одной из первых попыток позволить пользователям запрашивать данные, хранящиеся в Hadoop, а также Spark SQL, который представляет собой библиотеку для запроса данных, хранящихся в различных форматах в Spark. Относительным

новичком является Apache Drill — инструмент, появившийся в 2015 году и обладающий следующими возможностями.

- Облегчает запросы к данным в разных форматах, включая данные с разделителями, JSON, Parquet и журнальные файлы.
- Подключается к реляционным базам данных, Hadoop, HBase и Kafka, а также работает со специализированными форматами данных, такими как PCAP, BlockChain и др.
- Позволяет создавать пользовательские плагины для подключения к большинству любых других хранилищ данных.
- Не требует знания определений схем заранее.
- Поддерживает стандарт SQL:2003.
- Работает с популярными инструментами бизнес-аналитики, такими как Tableau и Apache Superset.

Используя Apache Drill, вы можете подключиться к любому количеству источников данных и выполнять запросы без первоначальной настройки репозитория метаданных. Вопросы установки и настройки Apache Drill выходят за рамки данной книги, так что, если вы заинтересованы в этих темах, я настоятельно рекомендую книгу *Learning Apache Drill* Чарльза Живра (Charles Givre) и Пола Роджерса (Paul Rogers).

Запрос файлов с помощью Apache Drill

Давайте начнем с использования Apache Drill для запроса данных, хранящихся в файле. Apache Drill умеет читать несколько различных форматов файлов, включая файлы захвата пакетов PCAP, которые используют бинарный формат и содержат информацию о пакетах, проходящих по сети. Все, что нужно сделать для запроса файла PCAP, — это настроить плагин распределенной файловой системы (dfs), указав путь к каталогу, содержащему файлы.

Первое, что я хотел бы сделать, — это узнать, какие столбцы доступны в файле, который я буду запрашивать. Apache Drill включает в себя частичную поддержку `information_schema` (см. главу 15, “Метаданные”), так что вы можете получить информацию высокого уровня о файлах данных в вашем рабочем пространстве:

```
apache drill> SELECT file_name, is_directory, is_file, permission
... . . . . > FROM information_schema.`files`
... . . . . > WHERE schema_name = 'dfs.data';
```

```
+-----+-----+-----+-----+
| file_name | is_directory | is_file | permission |
+-----+-----+-----+-----+
| attack-trace.pcap | false | true | rwxrwx-- |
+-----+-----+-----+
1 row selected (0.238 seconds)
```

Результаты показывают, что в моем рабочем пространстве данных есть один файл с именем attack-trace.pcap (что является полезной информацией), но я не могу запросить information_schema.columns, чтобы узнать, какие столбцы имеются в файле. Однако выполнение запроса, который не возвращает никаких строк из файла, показывает набор доступных столбцов¹:

```
apache drill> SELECT * FROM dfs.data.`attack-trace.pcap`
. . . . . > WHERE 1=2;
+-----+-----+-----+-----+
| type | network | timestamp | timestamp_micro | src_ip | dst_ip |
+-----+-----+-----+-----+
| src_port | dst_port | src_mac_address | dst_mac_address |
+-----+-----+-----+-----+
| tcp_session | tcp_ack | tcp_flags | tcp_flags_ns |
+-----+-----+-----+-----+
| tcp_flags_cwr | tcp_flags_ece | tcp_flags_ece_ecn_capable |
+-----+-----+-----+
| tcp_flags_ece_congestion_experienced | tcp_flags_urg |
+-----+-----+
| tcp_flags_ack | tcp_flags_psh | tcp_flags_RST | tcp_flags_syn |
+-----+-----+-----+
| tcp_flags_fin | tcp_parsed_flags | packet_length | is_corrupt |
+-----+-----+-----+
-----+
| data |
-----+
```

No rows selected (0.285 seconds)

Теперь, когда я знаю имена столбцов в файле PCAP, я готов писать запросы. Вот запрос, который считает количество пакетов, отправленных из каждого IP-адреса каждому целевому порту:

¹ Эти результаты показывают столбцы в файле на основании понимания Apache Drill структуры файлов PCAP. Если запросить файл, формат которого не известен Apache Drill, результирующий набор будет содержать массив строк с единственным столбцом с именем columns.

```

apache drill> SELECT src_ip, dst_port,
. . . . . > count(*) AS packet_count
. . . . . > FROM dfs.data.`attack-trace.pcap`
. . . . . > GROUP BY src_ip, dst_port;
+-----+-----+-----+
| src_ip      | dst_port | packet_count |
+-----+-----+-----+
| 98.114.205.102 |     445 |          18 |
| 192.150.11.111 |    1821 |           3 |
| 192.150.11.111 |    1828 |          17 |
| 98.114.205.102 |    1957 |           6 |
| 192.150.11.111 |    1924 |           6 |
| 192.150.11.111 |   8884 |          15 |
| 98.114.205.102 |  36296 |          12 |
| 98.114.205.102 |   1080 |         159 |
| 192.150.11.111 |   2152 |          112 |
+-----+-----+-----+
9 rows selected (0.254 seconds)

```

Вот еще один запрос, который посекундно агрегирует информацию о пакетах:

```

apache drill> SELECT trunc(extract(second from `timestamp`))
. . . . . >                               as packet_time,
. . . . . > count(*) AS num_packets,
. . . . . > sum(packet_length) AS tot_volume
. . . . . > FROM dfs.data.`attack-trace.pcap`
. . . . . > GROUP BY trunc(extract(second from `timestamp`));
+-----+-----+-----+
| packet_time | num_packets | tot_volume |
+-----+-----+-----+
|    28.0 |      15 |     1260 |
|    29.0 |      12 |     1809 |
|    30.0 |      13 |     4292 |
|    31.0 |       3 |      286 |
|    32.0 |       2 |      118 |
|    33.0 |      15 |     1054 |
|    34.0 |      35 |    14446 |
|    35.0 |      29 |    16926 |
|    36.0 |      25 |    16710 |
|    37.0 |      25 |    16710 |
|    38.0 |      26 |    17788 |
|    39.0 |      23 |    15578 |
|    40.0 |      25 |    16710 |
|    41.0 |      23 |    15578 |
|    42.0 |      30 |    20052 |
|    43.0 |      25 |    16710 |
|    44.0 |      22 |     7484 |
+-----+-----+-----+
17 rows selected (0.422 seconds)

```

В этом запросе необходимо заключить `timestamp` в обратные кавычки (`), потому что это зарезервированное слово.

Вы можете запросить файлы, хранящиеся локально, в вашей сети, в распределенной файловой системе или в облаке. Apache Drill имеетстроенную поддержку многих типов файлов, но можно также создавать собственные плагины, чтобы разрешить Apache Drill запрашивать другие типы файлов. В следующих двух разделах будут рассматриваться запросы данных, хранящихся в базе данных.

Запрос MySQL с использованием Apache Drill

Apache Drill может подключаться к любой реляционной базе данных с использованием драйвера JDBC, так что следующий логический шаг — показать, как Apache Drill может запросить пример базы данных Sakila, используемой в этой книге. Все, что вам нужно сделать, — это загрузить драйвер JDBC для MySQL и настроить Apache Drill для подключения к базе данных MySQL.



Вас может удивить, зачем использовать Apache Drill для запроса к MySQL. Одна из причин состоит в том, что (как вы увидите в конце главы) Apache Drill позволяет писать запросы, которые соединяют данные из разных источников, так что вы сможете написать запрос, который соединяет, например, данные из MySQL, Hadoop и файлов с разделителями.

Первый шаг состоит в выборе базы данных:

```
apache drill (information_schema)> use mysql.sakila;
+-----+
| ok | summary |
+-----+
| true | Default schema changed to [mysql.sakila] |
+-----+
1 row selected (0.062 seconds)
```

После выбора базы данных можно выполнить команду `show tables`, чтобы увидеть все таблицы, доступные в выбранной схеме:

```
apache drill (mysql.sakila)> show tables;
+-----+-----+
| TABLE_SCHEMA | TABLE_NAME |
+-----+-----+
| mysql.sakila | actor      |
| mysql.sakila | address    |
| mysql.sakila | category   |
```

```
| mysql.sakila | city
| mysql.sakila | country
| mysql.sakila | customer
| mysql.sakila | film
| mysql.sakila | film_actor
| mysql.sakila | film_category
| mysql.sakila | film_text
| mysql.sakila | inventory
| mysql.sakila | language
| mysql.sakila | payment
| mysql.sakila | rental
| mysql.sakila | sales
| mysql.sakila | staff
| mysql.sakila | store
| mysql.sakila | actor_info
| mysql.sakila | customer_list
| mysql.sakila | film_list
| mysql.sakila | nicer_but_slower_film_list
| mysql.sakila | sales_by_film_category
| mysql.sakila | sales_by_store
| mysql.sakila | staff_list
+-----+
24 rows selected (0.147 seconds)
```

Я начну с выполнения нескольких запросов, продемонстрированных в других главах. Вот простое соединение двух таблиц из главы 5, “Запросы к нескольким таблицам”:

Следующий запрос взят из главы 8, “Группировка и агрегация”, и включает в себя как предложение group by, так и предложение having:

Наконец, вот запрос из главы 16, “Аналитические функции”, который включает в себя три разных функции ранжировки:

	236	42	4	3	3
	75	41	5	5	4
	197	40	6	6	5
...					
	248	15	595	594	30
	61	14	596	596	31
	110	14	597	596	31
	281	14	598	596	31
	318	12	599	599	32

599 rows selected (1.827 seconds)

Эти несколько примеров демонстрируют способность Apache Drill выполнять достаточно сложные запросы к MySQL. Нужно помнить, что Apache Drill работает со многими реляционными базами данных, а не только с MySQL, поэтому некоторые возможности языка могут различаться (например, функции преобразования данных). За дополнительной информацией обратитесь к документации по реализации SQL в Apache Drill (<https://oreile.ly/d2jse>).

Запрос MongoDB с использованием Apache Drill

Теперь, после демонстрации использования Apache Drill для выполнения запросов базы данных Sakila в MySQL, выполним следующий логический шаг — преобразуем данные Sakila в другой распространенный формат, сохраним их в нереляционной базе данных и используем Apache Drill для запроса данных. Я решил преобразовать данные в JSON и сохранить их в MongoDB, одной из наиболее популярных не-SQL платформ для хранения документов. Apache Drill включает в себя плагин для MongoDB, а также понимает, как работать с документами JSON, так что загрузить JSON-файлы в MongoDB и начать писать запросы относительно легко.

Перед тем как погрузиться в запросы, давайте рассмотрим структуру JSON-файлов, поскольку они находятся не в нормированной форме. Первый из двух файлов JSON — `films.json`:

```
{"_id":1,
"Actors":[
{"First name":"PENELOPE","Last name":"GUINESS","actorId":1},
 {"First name":"CHRISTIAN","Last name":"GABLE","actorId":10},
 {"First name":"LUCILLE","Last name":"TRACY","actorId":20},
 {"First name":"SANDRA","Last name":"PECK","actorId":30},
 {"First name":"JOHNNY","Last name":"CAGE","actorId":40},
 {"First name":"MENA","Last name":"TEMPLE","actorId":53},
 {"First name":"WARREN","Last name":"NOLTE","actorId":108},
```

```
{"First name": "OPRAH", "Last name": "KILMER", "actorId": 162},  
 {"First name": "ROCK", "Last name": "DUKAKIS", "actorId": 188},  
 {"First name": "MARY", "Last name": "KEITEL", "actorId": 198}],  
 "Category": "Documentary",  
 "Description": "A Epic Drama of a Feminist And a Mad Scientist  
 who must Battle a Teacher in The Canadian Rockies",  
 "Length": "86",  
 "Rating": "PG",  
 "Rental Duration": "6",  
 "Replacement Cost": "20.99",  
 "Special Features": "Deleted Scenes, Behind the Scenes",  
 "Title": "ACADEMY DINOSAUR"},  
 {"_id": 2,  
 "Actors": [  
     {"First name": "BOB", "Last name": "FAWCETT", "actorId": 19},  
     {"First name": "MINNIE", "Last name": "ZELLWEGER", "actorId": 85},  
     {"First name": "SEAN", "Last name": "GUINNESS", "actorId": 90},  
     {"First name": "CHRIS", "Last name": "DEPP", "actorId": 160}],  
 "Category": "Horror",  
 "Description": "A Astounding Epistle of a Database Administrator  
 And a Explorer who must Find a Car in Ancient China",  
 "Length": "48",  
 "Rating": "G",  
 "Rental Duration": "3",  
 "Replacement Cost": "12.99",  
 "Special Features": "Trailers, Deleted Scenes",  
 "Title": "ACE GOLDFINGER"},  
 ...  
 {"_id": 999,  
 "Actors": [  
     {"First name": "CARMEN", "Last name": "HUNT", "actorId": 52},  
     {"First name": "MARY", "Last name": "TANDY", "actorId": 66},  
     {"First name": "PENELOPE", "Last name": "CRONYN", "actorId": 104},  
     {"First name": "WHOOPIE", "Last name": "HURT", "actorId": 140},  
     {"First name": "JADA", "Last name": "RYDER", "actorId": 142}],  
 "Category": "Children",  
 "Description": "A Fateful Reflection of a Waitress And a Boat  
 who must Discover a Sumo Wrestler in Ancient China",  
 "Length": "101",  
 "Rating": "R",  
 "Rental Duration": "5",  
 "Replacement Cost": "28.99",  
 "Special Features": "Trailers, Deleted Scenes",  
 "Title": "ZOOLANDER FICTION"}  
 {"_id": 1000,  
 "Actors": [  
     {"First name": "IAN", "Last name": "TANDY", "actorId": 155},  
     {"First name": "NICK", "Last name": "DEGENERES", "actorId": 166},  
     {"First name": "LISA", "Last name": "MONROE", "actorId": 178}],  
 "Category": "Comedy",  
 "Description": "A Intrepid Panorama of a Mad Scientist And a Boy
```

```
    "who must Redeem a Boy in A Monastery",
    "Length":"50",
    "Rating":"NC-17",
    "Rental Duration":"3",
    "Replacement Cost":"18.99",
    "Special Features":
    "Trailers,Commentaries,Behind the Scenes",
    "Title":"ZORRO ARK"}
```

В этой коллекции 1000 документов, и каждый документ содержит ряд скрытых атрибутов (Title, Rating, _id), а также включает в себя список под названием Actors, который содержит от 1 до N элементов, состоящих из идентификатора актера, атрибутов имени и фамилии каждого актера, появляющегося в фильме. Таким образом, этот файл содержит все данные из таблиц actor, film и film_actor в базе данных MySQL Sakila.

Второй файл — customer.json, который сочетает в себе данные из таблиц customer, address, city, country, rental и payment базы данных MySQL Sakila:

```
{"_id":1,
  "Address":"1913 Hanoi Way",
  "City":"Sasebo",
  "Country":"Japan",
  "District":"Nagasaki",
  "First Name":"MARY",
  "Last Name":"SMITH",
  "Phone":"28303384290",
  "Rentals":[
    {"rentalId":1185,
     "filmId":611,
     "staffId":2,
     "Film Title":"MUSKeteers WAIT",
     "Payments":[
       {"Payment Id":3,"Amount":5.99,
        "Payment Date":"2005-06-15 00:54:12"}],
     "Rental Date":"2005-06-15 00:54:12.0",
     "Return Date":"2005-06-23 02:42:12.0"},

    {"rentalId":1476,
     "filmId":308,
     "staffId":1,
     "Film Title":"FERRIS MOTHER",
     "Payments":[
       {"Payment Id":5,"Amount":9.99,
        "Payment Date":"2005-06-15 21:08:46"}],
     "Rental Date":"2005-06-15 21:08:46.0",
     "Return Date":"2005-06-25 02:26:46.0"},

    ...
    {"rentalId":14825,
```

```
"filmId":317,  
"staffId":2,  
"Film Title":"FIREBALL PHILADELPHIA",  
"Payments": [  
    {"Payment Id":30,"Amount":1.99,  
     "Payment Date":"2005-08-22 01:27:57"}],  
"Rental Date":"2005-08-22 01:27:57.0",  
"Return Date":"2005-08-27 07:01:57.0"}  
}
```

Этот файл содержит 599 записей (здесь показана только одна), которые загружаются в MongoDB в виде 599 документов в коллекции `customers`. Каждый документ содержит информацию об одном клиенте наряду со всеми платежами, выполненными этим клиентом. Кроме того, документы содержат вложенные списки, поскольку каждый прокат в списке `Rentals` также содержит список `Payments`.

После того как файлы JSON загружены, база данных Mongo содержит две коллекции (`films` и `customers`), а данные в этих коллекциях охватывают девять разных таблиц базы данных Sakila. Это довольно типичный сценарий, поскольку прикладные программисты обычно работают с коллекциями и в общем случае предпочитают не деконструировать данные для хранения в нормализованных реляционных таблицах. С точки зрения SQL главная проблема в том, чтобы определить, как сформировать эти данные так, чтобы они вели себя так, как если бы они хранились в нескольких таблицах.

Для иллюстрации давайте построим следующий запрос к коллекции `films`: найти всех актеров, которые появились в 10 или более фильмах, с рейтингом либо G или PG. Вот как выглядят необработанные данные:

```
| "actorId":"115"},  
| {"First name":"JEFF","Last name":"SILVERSTONE",  
| "actorId":"180"},  
| {"First name":"ROCK","Last name":"DUKAKIS",  
| "actorId":"188"}]  
PG [{"First name":"UMA","Last name":"WOOD",  
| "actorId":"13"},  
| {"First name":"HELEN","Last name":"VOIGHT",  
| "actorId":"17"},  
| {"First name":"CAMERON","Last name":"STREEP",  
| "actorId":"24"},  
| {"First name":"CARMEN","Last name":"HUNT",  
| "actorId":"52"},  
| {"First name":"JANE","Last name":"JACKMAN",  
| "actorId":"131"},  
| {"First name":"BELA","Last name":"WALKEN",  
| "actorId":"196"}]  
...  
G [{"First name":"ED","Last name":"CHASE",  
| "actorId":3},  
| {"First name":"JULIA","Last name":"MCQUEEN",  
| "actorId":27},  
| {"First name":"JAMES","Last name":"PITT",  
| "actorId":84},  
| {"First name":"CHRISTOPHER","Last name":"WEST",  
| "actorId":163},  
| {"First name":"MENA","Last name":"HOPPER",  
| "actorId":170}]}  
+-----+  
372 rows selected (0.432 seconds)
```

Поле `Actors` представляет собой список из одного или нескольких документов актера. Для того чтобы взаимодействовать с этими данными, как если бы это была таблица, для преобразования списка во вложенную таблицу, содержащую три поля, может использоваться команда `flatten`:

```
| "actorId":"64"}  
| PG | {"First name":"PENELOPE", "Last name":"CRONYN",  
| | "actorId":"104"}  
| PG | {"First name":"HARRISON", "Last name":"BALE",  
| | "actorId":"115"}  
| PG | {"First name":"JEFF", "Last name":"SILVERSTONE",  
| | "actorId":"180"}  
| PG | {"First name":"ROCK", "Last name":"DUKAKIS",  
| | "actorId":"188"}  
| PG | {"First name":"UMA", "Last name":"WOOD",  
| | "actorId":"13"}  
| PG | {"First name":"HELEN", "Last name":"VOIGHT",  
| | "actorId":"17"}  
| PG | {"First name":"CAMERON", "Last name":"STREEP",  
| | "actorId":"24"}  
| PG | {"First name":"CARMEN", "Last name":"HUNT",  
| | "actorId":"52"}  
| PG | {"First name":"JANE", "Last name":"JACKMAN",  
| | "actorId":"131"}  
| PG | {"First name":"BELA", "Last name":"WALKEN",  
| | "actorId":"196"}  
| ...  
| G | {"First name":"ED", "Last name":"CHASE",  
| | "actorId":"3"}  
| G | {"First name":"JULIA", "Last name":"MCQUEEN",  
| | "actorId":"27"}  
| G | {"First name":"JAMES", "Last name":"PITT",  
| | "actorId":"84"}  
| G | {"First name":"CHRISTOPHER", "Last name":"WEST",  
| | "actorId":"163"}  
| G | {"First name":"MENA", "Last name":"HOPPER",  
| | "actorId":"170"}  
+-----+  
2119 rows selected (0.718 seconds)
```

Этот запрос возвращает 2119, а не 372 строки, возвращаемых предыдущим запросом, что указывает на то, что в среднем в каждом фильме с рейтингом G или PG появляется 5,7 актера. Затем этот запрос можно использовать в качестве подзапроса для группировки данных по рейтингу и актер, как показано далее:

```

    . . . . . )>   FROM films f
    . . . . . )> WHERE f.Rating IN ('G','PG')
    . . . . . )>   ) g_pg_films
    . . . . . )> GROUP BY g_pg_films.Rating,
    . . . . . )>   g_pg_films.actor_list.`First name`,
    . . . . . )>   g_pg_films.actor_list.`Last name`
    . . . . . )> HAVING count(*) > 9;
+-----+
| Rating | first_name | last_name | num_films |
+-----+
| PG     | JEFF        | SILVERSTONE | 12      |
| G      | GRACE       | MOSTEL      | 10      |
| PG     | WALTER      | TORN        | 11      |
| PG     | SUSAN       | DAVIS        | 10      |
| PG     | CAMERON    | ZELLWEGER   | 15      |
| PG     | RIP         | CRAWFORD   | 11      |
| PG     | RICHARD    | PENN        | 10      |
| G      | SUSAN       | DAVIS        | 13      |
| PG     | VAL         | BOLGER      | 12      |
| PG     | KIRSTEN    | AKROYD      | 12      |
| G      | VIVIEN      | BERGEN     | 10      |
| G      | BEN          | WILLIS      | 14      |
| G      | HELEN       | VOIGHT      | 12      |
| PG     | VIVIEN      | BASINGER    | 10      |
| PG     | NICK         | STALLONE    | 12      |
| G      | DARYL       | CRAWFORD   | 12      |
| PG     | MORGAN      | WILLIAMS   | 10      |
| PG     | FAY          | WINSLET     | 10      |
+-----+
18 rows selected (0.466 seconds)

```

Внутренний запрос использует команду `flatten` для создания одной строки для каждого актера, который появился в фильме G или PG, а внешний запрос просто выполняет группировку этого набора данных.

Теперь давайте напишем запрос к коллекции `customers` в Mongo. Это немного более сложная задача, поскольку каждый документ содержит список прокатов фильмов, каждый из которых содержит список платежей. Чтобы сделать задание немного интереснее, давайте также выполним соединение с коллекцией `films`, чтобы увидеть, как Apache Drill обрабатывает соединения. Запрос должен вернуть всех клиентов, которые потратили более 80 долларов на прокат фильмов с рейтингом G или PG. Вот как выглядит такой запрос:

```

        . . . . . )>      cust_data.last_name,
        . . . . . )>      f.Rating,
        . . . . . )>      flatten(
        . . . . . )>          cust_data.rental_data.Payments)
        . . . . . )>          payment_data
        . . . . . )>      FROM films f
        . . . . . )>      INNER JOIN
        . . . . . )>          (SELECT c.`First Name` first_name,
        . . . . . )>                  c.`Last Name` last_name,
        . . . . . )>          flatten(c.Rentals) rental_data
        . . . . . )>          FROM customers c
        . . . . . )>      ) cust_data
        . . . . . )>      ON f._id =
        . . . . . )>          cust_data.rental_data.filmID
        . . . . . )>      WHERE f.Rating IN ('G','PG')
        . . . . . )>      ) cust_payments
        . . . . . )>      GROUP BY first_name, last_name
        . . . . . )>      HAVING
        . . . . . )>          sum(cast(
        . . . . . )>              cust_payments.payment_data.Amount
        . . . . . )>          as decimal(4,2))) > 80;
+
+-----+-----+-----+
| first_name | last_name | tot_payments |
+-----+-----+-----+
| ELEANOR    | HUNT       |      85.80 |
| GORDON     | ALLARD     |      85.86 |
| CLARA      | SHAW       |      86.83 |
| JACQUELINE | LONG       |      86.82 |
| KARL        | SEAL       |      89.83 |
| PRISCILLA   | LOWE       |      95.80 |
| MONICA     | HICKS      |      85.82 |
| LOUIS       | LEONE      |      95.82 |
| JUNE        | CARROLL    |      88.83 |
| ALICE       | STEWART    |      81.82 |
+-----+-----+-----+
10 rows selected (1.658 seconds)

```

Наиболее глубоко вложенный запрос, который я назвал `cust_data`, преобразует список `Rentals`, так что запрос `cust_payments` может соединяться с коллекцией `films`, а также преобразовывать список `Payments`. Внешний запрос группирует данные по имени клиента и применяет предложение `having`, чтобы отфильтровать клиентов, которые потратили не более 80 долларов на фильмы с рейтингом G или PG.

Apache Drill и несколько источников данных

Пока что я использовал Apache Drill, чтобы работать с несколькими таблицами, хранящимися в одной базе данных. Но что если данные хранятся

в разных базах данных? Например, данные клиента/прокатов/платежные данные хранятся в MongoDB, а каталог данных фильмов/актеров хранится в MySQL? Если Apache Drill настроен для подключения к обеим базам данных, вам нужно просто указать, где искать данные. Вот запрос из предыдущего раздела, но теперь вместо соединения с коллекцией фильмов, хранящихся в MongoDB, выполняется соединение с таблицей фильмов из MySQL:

```
+-----+-----+-----+
10 rows selected (1.874 seconds)
```

Поскольку я использую несколько баз данных в одном и том же запросе, я указываю полный путь к каждой таблице и коллекции, чтобы точно указать источник данных. В этом Apache Drill действительно очень хорош — с его помощью легко соединить данные из нескольких источников в одном запросе без необходимости преобразовывать данные и переносить их из одного источника в другой.

Будущее SQL

Будущее реляционных баз данных несколько туманно. Возможно, технологии больших данных последнего десятилетия будут продолжать развиваться и отбирать свою долю рынка. Возможно и появление новых технологий, которые обойдут Hadoop и не-SQL базы данных и заберут свою долю рынка у реляционных баз данных. Однако в настоящее время большинство компаний все еще выполняют свои бизнес-функции, используя реляционные базы данных, и чтобы изменить это положение дел, должно пройти немало времени.

Будущее же SQL представляется немного более ясным. В то время как язык SQL начинался как механизм для взаимодействия с данными в реляционных базах данных, такие инструменты, как Apache Drill, действуют в качестве слоя абстракции, облегчающего анализ данных с использованием различных платформ баз данных. Мне кажется, что эта тенденция продолжится, а SQL останется критически важным инструментом для анализа данных и генерации отчетности еще в течение многих лет.

ПРИЛОЖЕНИЕ А

Схема базы данных Sakila

На рис. А.1 представлена диаграмма модели данных используемого в книге примера базы данных Sakila. На ней отображены объекты, или таблицы, базы данных вместе с отношениями внешних ключей между таблицами. Вот несколько подсказок, которые помогут вам разобраться в условных обозначениях.

- Каждый прямоугольник представляет таблицу с именем таблицы, указанным над верхним левым углом прямоугольника. Первым в таблице указан столбец первичного ключа; затем идут столбцы, не являющиеся ключевыми.
- Линии между таблицами представляют собой отношения внешних ключей. Маркеры на концах линий представляют допустимое множество записей, которое может быть либо 0, либо 1, либо много. Например, если вы посмотрите на отношения между таблицами *customer* и *rental*, то увидите, что *rental* связан ровно с одним *customer*, но *customer* может иметь нуль, один или несколько записей *rental*.

Для получения дополнительной информации о диаграммах модели данных обратитесь к соответствующей странице Википедии (<https://oreil.ly/hLEeq>).

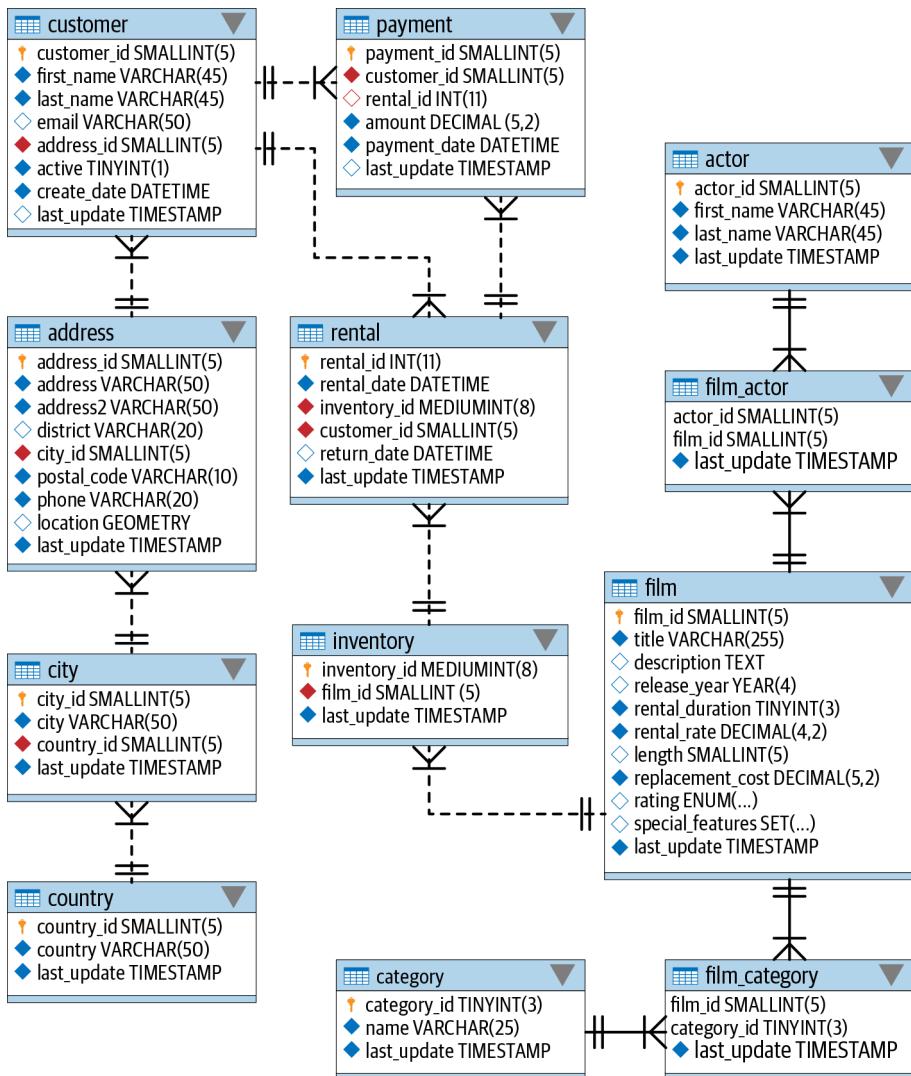


Рис. А.1. Схема базы данных

Ответы к упражнениям

Глава 3

УПРАЖНЕНИЕ 3.1

Получите идентификатор актера, а также имя и фамилию для всех актеров. Отсортируйте вывод сначала по фамилии, а затем — по имени.

```
mysql> SELECT actor_id, first_name, last_name
-> FROM actor
-> ORDER BY 3,2;
+-----+-----+-----+
| actor_id | first_name | last_name |
+-----+-----+-----+
|      58 | CHRISTIAN | AKROYD    |
|     182 | DEBBIE    | AKROYD    |
|      92 | KIRSTEN   | AKROYD    |
|     118 | CUBA       | ALLEN     |
|     145 | KIM        | ALLEN     |
|     194 | MERYL      | ALLEN     |
| ...   |           |           |
|     13 | UMA        | WOOD      |
|     63 | CAMERON   | WRAY      |
|    111 | CAMERON   | ZELLWEGER |
|    186 | JULIA      | ZELLWEGER |
|     85 | MINNIE    | ZELLWEGER |
+-----+-----+-----+
200 rows in set (0.02 sec)
```

УПРАЖНЕНИЕ 3.2

Получите идентификатор, имя и фамилию актера для всех актеров, чьи фамилии — 'WILLIAMS' или 'DAVIS'.

```
mysql> SELECT actor_id, first_name, last_name
-> FROM actor
-> WHERE last_name IN ('WILLIAMS','DAVIS');
+-----+-----+-----+
| actor_id | first_name | last_name |
+-----+-----+-----+
```

```
|      4 | JENNIFER    | DAVIS      |
| 101 | SUSAN       | DAVIS      |
| 110 | SUSAN       | DAVIS      |
| 72  | SEAN        | WILLIAMS   |
| 137 | MORGAN     | WILLIAMS   |
| 172 | GROUCHO    | WILLIAMS   |
+-----+-----+-----+
6 rows in set (0.03 sec)
```

УПРАЖНЕНИЕ 3.3

Напишите запрос к таблице rental, который возвращает идентификаторы клиентов, бравших фильмы напрокат 5 июля 2005 года (используйте столбец rental.rental_date; можете также использовать функцию date(), чтобы игнорировать компонент времени). Выведите по одной строке для каждого уникального идентификатора клиента.

```
mysql> SELECT DISTINCT customer_id
   -> FROM rental
   -> WHERE date(rental_date) = '2005-07-05';
+-----+
| customer_id |
+-----+
|      8 |
|     37 |
|     60 |
|    111 |
|    114 |
|    138 |
|    142 |
|    169 |
|    242 |
|    295 |
|    296 |
|    298 |
|    322 |
|    348 |
|    349 |
|    369 |
|    382 |
|    397 |
|    421 |
|    476 |
|    490 |
|    520 |
|    536 |
|    553 |
|    565 |
|    586 |
|    594 |
```

```
+-----+
27 rows in set (0.22 sec)
```

УПРАЖНЕНИЕ 3.4

Заполните пропущенные места (обозначенные как <#>) в следующем многостраничном запросе, чтобы получить показанные результаты:

```
mysql> SELECT c.email, r.return_date
-> FROM customer c
->     INNER JOIN rental <1>
->     ON c.customer_id = <2>
-> WHERE date(r.rental_date) = '2005-06-14'
-> ORDER BY <3> <4>;
+-----+
| email                         | return_date      |
+-----+
| DANIEL.CABRAL@sakilacustomer.org | 2005-06-23 22:00:38 |
| TERRANCE.ROUSH@sakilacustomer.org | 2005-06-23 21:53:46 |
| MIRIAM.MCKINNEY@sakilacustomer.org | 2005-06-21 17:12:08 |
| GWENDOLYN.MAY@sakilacustomer.org | 2005-06-20 02:40:27 |
| JEANETTE.GREENE@sakilacustomer.org | 2005-06-19 23:26:46 |
| HERMAN.DEVORE@sakilacustomer.org | 2005-06-19 03:20:09 |
| JEFFERY.PINSON@sakilacustomer.org | 2005-06-18 21:37:33 |
| MATTHEW.MAHAN@sakilacustomer.org | 2005-06-18 05:18:58 |
| MINNIE.ROMERO@sakilacustomer.org | 2005-06-18 01:58:34 |
| SONIA.GREGORY@sakilacustomer.org | 2005-06-17 21:44:11 |
| TERRENCE.GUNDERSON@sakilacustomer.org | 2005-06-17 05:28:35 |
| ELMER.NOE@sakilacustomer.org | 2005-06-17 02:11:13 |
| JOYCE.EDWARDS@sakilacustomer.org | 2005-06-16 21:00:26 |
| AMBER.DIXON@sakilacustomer.org | 2005-06-16 04:02:56 |
| CHARLES.KOWALSKI@sakilacustomer.org | 2005-06-16 02:26:34 |
| CATHERINE.CAMPBELL@sakilacustomer.org | 2005-06-15 20:43:03 |
+-----+
16 rows in set (0.03 sec)
```

- <1> заменяется на r.
- <2> заменяется r.customer_id.
- <3> заменяется на 2.
- <4> заменяется на desc.

Глава 4

В первых двух упражнениях вам потребуется следующее подмножество строк из таблицы payment:

```
+-----+
| payment_id | customer_id | amount | date(payment_date) |
+-----+-----+-----+
```

101	4	8.99	2005-08-18
102	4	1.99	2005-08-19
103	4	2.99	2005-08-20
104	4	6.99	2005-08-20
105	4	4.99	2005-08-21
106	4	2.99	2005-08-22
107	4	1.99	2005-08-23
108	5	0.99	2005-05-29
109	5	6.99	2005-05-31
110	5	1.99	2005-05-31
111	5	3.99	2005-06-15
112	5	2.99	2005-06-16
113	5	4.99	2005-06-17
114	5	2.99	2005-06-19
115	5	4.99	2005-06-20
116	5	4.99	2005-07-06
117	5	2.99	2005-07-08
118	5	4.99	2005-07-09
119	5	5.99	2005-07-09
120	5	1.99	2005-07-09

УПРАЖНЕНИЕ 4.1

Какие из идентификаторов платежей будут возвращены при следующих условиях фильтрации?

```
customer_id <> 5
AND (amount > 8 OR date(payment_date) = '2005-08-23')
```

Идентификаторы платежей 101 и 107.

УПРАЖНЕНИЕ 4.2

Какие из идентификаторов платежей будут возвращены при следующих условиях фильтрации?

```
customer_id = 5 AND
NOT (amount > 6 OR date(payment_date) = '2005-06-19')
```

Идентификаторы платежей 108, 110, 111, 112, 113, 115, 116, 117, 118, 119 и 120.

УПРАЖНЕНИЕ 4.3

Создайте запрос, который извлекает из таблицы payments все строки, в которых сумма равна 1,98, 7,98 или 9,98.

```
mysql> SELECT amount
-> FROM payment
-> WHERE amount IN (1.98, 7.98, 9.98);
```

```
+-----+
| amount |
+-----+
| 7.98 |
| 9.98 |
| 1.98 |
| 7.98 |
| 7.98 |
| 7.98 |
| 7.98 |
+-----+
7 rows in set (0.01 sec)
```

УПРАЖНЕНИЕ 4.4

Создайте запрос, который находит всех клиентов, в фамилиях которых содержатся буква A во второй позиции и буква W — в любом месте после A.

```
mysql> SELECT first_name, last_name
    -> FROM customer
    -> WHERE last_name LIKE '_A%W%';
+-----+-----+
| first_name | last_name  |
+-----+-----+
| KAY        | CALDWELL   |
| JOHN       | FARNSWORTH |
| JILL       | HAWKINS    |
| LEE         | HAWKS      |
| LAURIE     | LAWRENCE   |
| JEANNE     | LAWSON     |
| LAWRENCE   | LAWTON     |
| SAMUEL     | MARLOW     |
| ERICA       | MATTHEWS   |
+-----+-----+
9 rows in set (0.02 sec)
```

Глава 5

УПРАЖНЕНИЕ 5.1

Заполните пропущенные места (обозначенные как <#>) в следующем запросе так, чтобы получить показанные результаты:

```
mysql> SELECT c.first_name, c.last_name, a.address, ct.city
    -> FROM customer c
    -> INNER JOIN address <1>
    -> ON c.address_id = a.address_id
    -> INNER JOIN city ct
    -> ON a.city_id = <2>
    -> WHERE a.district = 'California';
```

```
+-----+-----+-----+-----+
| first_name | last_name | address | city |
+-----+-----+-----+-----+
| PATRICIA | JOHNSON | 1121 Loja Avenue | San Bernardino |
| BETTY | WHITE | 770 Bydgoszcz Avenue | Citrus Heights |
| ALICE | STEWART | 1135 Izumisano Parkway | Fontana |
| ROSA | REYNOLDS | 793 Cam Ranh Avenue | Lancaster |
| RENEE | LANE | 533 al-Ayn Boulevard | Compton |
| KRISTIN | JOHNSTON | 226 Brest Manor | Sunnyvale |
| CASSANDRA | WALTERS | 920 Kumbakonam Loop | Salinas |
| JACOB | LANCE | 1866 al-Qatif Avenue | El Monte |
| RENE | MCALISTER | 1895 Zhezqazghan Drive | Garden Grove |
+-----+-----+-----+-----+
```

9 rows in set (0.00 sec)

<1> заменяется на а.

<2> заменяется ct.city_id.

УПРАЖНЕНИЕ 5.2

Напишите запрос, который выводил бы названия всех фильмов, в которых играл актер с именем JOHN.

```
mysql> SELECT f.title
   -> FROM film f
   -> INNER JOIN film_actor fa
   -> ON f.film_id = fa.film_id
   -> INNER JOIN actor a
   -> ON fa.actor_id = a.actor_id
   -> WHERE a.first_name = 'JOHN';
+-----+
| title           |
+-----+
| ALLEY EVOLUTION |
| BEVERLY OUTLAW  |
| CANDLES GRAPES  |
| CLEOPATRA DEVIL |
| COLOR PHILADELPHIA |
| CONQUERER NUTS  |
| DAUGHTER MADIGAN |
| GLEAMING JAWBREAKER |
| GOLDMINE TYCOON  |
| HOME PITY        |
| INTERVIEW LIAISONS |
| ISHTAR ROCKETEER |
| JAPANESE RUN     |
| JERSEY SASSY     |
| LUKE MUMMY       |
| MILLION ACE      |
| MONSTER SPARTACUS |
| NAME DETECTIVE   |
```

```

| NECKLACE OUTBREAK      |
| NEWSIES STORY          |
| PET HAUNTING          |
| PIANIST OUTFIELD      |
| PINOCCHIO SIMON       |
| PITTSBURGH HUNCHBACK  |
| QUILLS BULL            |
| RAGING AIRPLANE        |
| ROXANNE REBEL          |
| SATISFACTION CONFIDENTIAL |
| SONG HEDWIG            |
+-----+
29 rows in set (0.07 sec)

```

УПРАЖНЕНИЕ 5.3

Создайте запрос, который возвращает все адреса в одном и том же городе. Вам нужно будет соединить таблицу адресов с самой собой, и каждая строка должна включать два разных адреса.

```

mysql> SELECT a1.address addr1, a2.address addr2, a1.city_id
->   FROM address a1
->   INNER JOIN address a2
-> WHERE a1.city_id = a2.city_id
->   AND a1.address_id <> a2.address_id;
+-----+-----+-----+
| addr1           | addr2           | city_id |
+-----+-----+-----+
| 47 MySakila Drive | 23 Workhaven Lane | 300    |
| 28 MySQL Boulevard | 1411 Lillydale Drive | 576    |
| 23 Workhaven Lane | 47 MySakila Drive | 300    |
| 1411 Lillydale Drive | 28 MySQL Boulevard | 576    |
| 1497 Yuzhou Drive | 548 Uruapan Street | 312    |
| 587 Benguela Manor | 43 Vilnius Manor | 42     |
| 548 Uruapan Street | 1497 Yuzhou Drive | 312    |
| 43 Vilnius Manor | 587 Benguela Manor | 42    |
+-----+-----+-----+
8 rows in set (0.00 sec)

```

Глава 6

УПРАЖНЕНИЕ 6.1

Пусть множество A = {L,M,N,O,P}, а множество B = {P,Q,R,S,T}. Какие множества будут генерированы следующими операциями?

- A union B
- A union all B

- A intersect B
 - A except B
1. A union B = {L M N O P Q R S T}
 2. A union all B = {L M N O P P Q R S T}
 3. A intersect B = {P}
 4. A except B = {L M N O}

УПРАЖНЕНИЕ 6.2

Напишите составной запрос, который находит имена и фамилии всех актеров и клиентов, чьи фамилии начинаются с буквы L.

```
mysql> SELECT first_name, last_name
-> FROM actor
-> WHERE last_name LIKE 'L%'
-> UNION
-> SELECT first_name, last_name
-> FROM customer
-> WHERE last_name LIKE 'L%';
```

first_name	last_name
MATTHEW	LEIGH
JOHNNY	LOLOBRIGIDA
MISTY	LAMBERT
JACOB	LANCE
RENEE	LANE
HEIDI	LARSON
DARYL	LARUE
LAURIE	LAWRENCE
JEANNE	LAWSON
LAWRENCE	LAWTON
KIMBERLY	LEE
LOUIS	LEONE
SARAH	LEWIS
GEORGE	LINTON
MAUREEN	LITTLE
DWIGHT	LOMBARDI
JACQUELINE	LONG
AMY	LOPEZ
BARRY	LOVELACE
PRISCILLA	LOWE
VELMA	LUCAS
WILLARD	LUMPKIN
LEWIS	LYMAN
JACKIE	LYNCH

```
+-----+-----+
24 rows in set (0.01 sec)
```

УПРАЖНЕНИЕ 6.3

Отсортируйте результаты выполнения упражнения 6.2 по столбцу last_name.

```
mysql> SELECT first_name, last_name
-> FROM actor
-> WHERE last_name LIKE 'L%'
-> UNION
-> SELECT first_name, last_name
-> FROM customer
-> WHERE last_name LIKE 'L%'
-> ORDER BY last_name;
+-----+-----+
| first_name | last_name |
+-----+-----+
| MISTY      | LAMBERT   |
| JACOB      | LANCE     |
| RENEE      | LANE      |
| HEIDI      | LARSON    |
| DARYL      | LARUE     |
| LAURIE     | LAWRENCE  |
| JEANNE     | LAWSON    |
| LAWRENCE   | LAWTON    |
| KIMBERLY   | LEE        |
| MATTHEW    | LEIGH     |
| LOUIS      | LEONE     |
| SARAH      | LEWIS     |
| GEORGE     | LINTON    |
| MAUREEN    | LITTLE    |
| JOHNNY     | LOLLOBRIGIDA |
| DWIGHT     | LOMBARDI  |
| JACQUELINE | LONG      |
| AMY         | LOPEZ     |
| BARRY       | LOVELACE  |
| PRISCILLA  | LOWE      |
| VELMA      | LUCAS     |
| WILLARD    | LUMPKIN   |
| LEWIS       | LYMAN     |
| JACKIE     | LYNCH     |
+-----+-----+
24 rows in set (0.00 sec)
```

Глава 7

УПРАЖНЕНИЕ 7.1

Напишите запрос, который возвращает символы строки 'Please find the substring in this string' с 17-го по 25-й.

```
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)
```

УПРАЖНЕНИЕ 7.2

Напишите запрос, который возвращает абсолютное значение и знак (-1, 0 или 1) числа -25,76823. Верните также число, округленное до ближайших двух знаков после запятой.

```
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)
```

УПРАЖНЕНИЕ 7.3

Напишите запрос, возвращающий для текущей даты только часть, соответствующую месяцу.

```
mysql> SELECT EXTRACT(MONTH FROM CURRENT_DATE());
+-----+
| EXTRACT(MONTH FROM CURRENT_DATE) |
+-----+
|          12                      |
+-----+
1 row in set (0.02 sec)
```

Глава 8

УПРАЖНЕНИЕ 8.1

Создайте запрос, который подсчитывает количество строк в таблице payment.

```
mysql> SELECT count(*) FROM payment;
+-----+
| count(*) |
+-----+
|    16049 |
+-----+
1 row in set (0.02 sec)
```

УПРАЖНЕНИЕ 8.2

Измените запрос из упражнения 8.1 так, чтобы подсчитать количество платежей, произведенных каждым клиентом. Выведите идентификатор клиента и общую уплаченную сумму для каждого клиента.

```
mysql> SELECT customer_id, count(*), sum(amount)
-> FROM payment
-> GROUP BY customer_id;
+-----+-----+-----+
| customer_id | count(*) | sum(amount) |
+-----+-----+-----+
|      1 |      32 |     118.68 |
|      2 |      27 |     128.73 |
|      3 |      26 |     135.74 |
|      4 |      22 |      81.78 |
|      5 |      38 |     144.62 |
| ...          |          |          |
|    595 |      30 |     117.70 |
|    596 |      28 |      96.72 |
|    597 |      25 |      99.75 |
|    598 |      22 |      83.78 |
|    599 |      19 |      83.81 |
+-----+-----+-----+
599 rows in set (0.03 sec)
```

УПРАЖНЕНИЕ 8.3

Измените запрос из упражнения 8.2, включив в него только тех клиентов, у которых имеется не менее 40 выплат.

```
mysql> SELECT customer_id, count(*), sum(amount)
-> FROM payment
-> GROUP BY customer_id
-> HAVING count(*) >= 40;
+-----+-----+-----+
| customer_id | count(*) | sum(amount) |
+-----+-----+-----+
|      75 |      41 |     155.59 |
|     144 |      42 |     195.58 |
|     148 |      46 |     216.54 |
|     197 |      40 |     154.60 |
+-----+-----+-----+
```

```
|      236 |       42 |     175.58 |
|      469 |       40 |     177.60 |
|      526 |       45 |     221.55 |
+-----+-----+-----+
7 rows in set (0.03 sec)
```

Глава 9

УПРАЖНЕНИЕ 9.1

Создайте запрос к таблице film, который использует условие фильтрации с некоррелированным подзапросом к таблице category, чтобы найти все боевики (category.name = 'Action').

```
mysql> SELECT title
    -> FROM film
    -> WHERE film_id IN
    -> (SELECT fc.film_id
    ->   FROM film_category fc INNER JOIN category c
    ->     ON fc.category_id = c.category_id
    ->   WHERE c.name = 'Action');
+-----+
| title           |
+-----+
| AMADEUS HOLY   |
| AMERICAN CIRCUS|
| ANTITRUST TOMATOES |
| ARK RIDGEMONT  |
| BAREFOOT MANCHURIAN |
| BERETS AGENT    |
| BRIDE INTRIGUE  |
| BULL SHAWSHANK  |
| CADDYSHACK JEDI   |
| CAMPUS REMEMBER |
| CASUALTIES ENCINO |
| CELEBRITY HORN   |
| CLUELESS BUCKET  |
| CROW GREASE     |
| DANCES NONE     |
| DARKO DORADO    |
| DARN FORRESTER  |
| DEVIL DESIRE    |
| DRAGON SQUAD    |
| DREAM PICKUP    |
| DRIFTER COMMANDMENTS |
| EASY GLADIATOR  |
| ENTRAPMENT SATISFACTION |
| EXCITEMENT EVE   |
| FANTASY TROOPERS |
| FIREHOUSE VIETNAM|
```

```
| FOOL MOCKINGBIRD          |
| FORREST SONS              |
| GLASS DYING                |
| GOSFORD DONNIE             |
| GRAIL FRANKENSTEIN         |
| HANDICAP BOONDOCK          |
| HILLS NEIGHBORS            |
| KISSING DOLLS              |
| LAWRENCE LOVE               |
| LORD ARIZONA                |
| LUST LOCK                  |
| MAGNOLIA FORRESTER          |
| MIDNIGHT WESTWARD          |
| MINDS TRUMAN                |
| MOCKINGBIRD HOLLYWOOD       |
| MONTEZUMA COMMAND           |
| PARK CITIZEN                |
| PATRIOT ROMAN               |
| PRIMARY GLASS                |
| QUEST MUSSOLINI              |
| REAR TRADING                 |
| RINGS HEARTBREAKERS          |
| RUGRATS SHAKESPEARE          |
| SHRUNK DIVINE                |
| SIDE ARK                     |
| SKY MIRACLE                  |
| SOUTH WAIT                   |
| SPEAKEASY DATE                |
| STAGECOACH ARMAGEDDON        |
| STORY SIDE                   |
| SUSPECTS QUILLS              |
| TRIP NEWTON                  |
| TRUMAN CRAZY                  |
| UPRISING UPTOWN                |
| WATERFRONT DELIVERANCE        |
| WEREWOLF LOLA                  |
| WOMEN DORADO                  |
| WORST BANGER                  |
+-----+
64 rows in set (0.06 sec)
```

УПРАЖНЕНИЕ 9.2

Переработайте запрос из упражнения 9.1, используя *коррелированный подзапрос* к таблицам category и film_category для получения тех же результатов.

```
mysql> SELECT f.title
      -> FROM film f
      -> WHERE EXISTS
```

```
-> (SELECT 1
->   FROM film_category fc INNER JOIN category c
->     ON fc.category_id = c.category_id
->   WHERE c.name = 'Action'
->     AND fc.film_id = f.film_id);
+-----+
| title          |
+-----+
| AMADEUS HOLY      |
| AMERICAN CIRCUS    |
| ANTITRUST TOMATOES  |
| ARK RIDGEMONT    |
| BAREFOOT MANCHURIAN  |
| BERETS AGENT      |
| BRIDE INTRIGUE    |
| BULL SHAWSHANK    |
| CADDYSHACK JEDI    |
| CAMPUS REMEMBER   |
| CASUALTIES ENCINO  |
| CELEBRITY HORN     |
| CLUELESS BUCKET    |
| CROW GREASE       |
| DANCES NONE       |
| DARKO DORADO      |
| DARN FORRESTER    |
| DEVIL DESIRE      |
| DRAGON SQUAD      |
| DREAM PICKUP      |
| DRIFTER COMMANDMENTS |
| EASY GLADIATOR    |
| ENTRAPMENT SATISFACTION |
| EXCITEMENT EVE    |
| FANTASY TROOPERS  |
| FIREHOUSE VIETNAM  |
| FOOL MOCKINGBIRD  |
| FORREST SONS      |
| GLASS DYING       |
| GOSFORD DONNIE    |
| GRAIL FRANKENSTEIN |
| HANDICAP BOONDOCK  |
| HILLS NEIGHBORS   |
| KISSING DOLLS     |
| LAWRENCE LOVE     |
| LORD ARIZONA      |
| LUST LOCK         |
| MAGNOLIA FORRESTER |
| MIDNIGHT WESTWARD  |
| MINDS TRUMAN      |
| MOCKINGBIRD HOLLYWOOD |
| MONTEZUMA COMMAND  |
| PARK CITIZEN      |
```

```

| PATRIOT ROMAN          |
| PRIMARY GLASS          |
| QUEST MUSSOLINI        |
| REAR TRADING           |
| RINGS HEARTBREAKERS    |
| RUGRATS SHAKESPEARE    |
| SHRUNK DIVINE          |
| SIDE ARK                |
| SKY MIRACLE             |
| SOUTH WAIT               |
| SPEAKEASY DATE          |
| STAGECOACH ARMAGEDDON   |
| STORY SIDE               |
| SUSPECTS QUILLS         |
| TRIP NEWTON              |
| TRUMAN CRAZY            |
| UPRISING UPTOWN          |
| WATERFRONT DELIVERANCE   |
| WEREWOLF LOLA            |
| WOMEN DORADO             |
| WORST BANGER             |
+-----+

```

64 rows in set (0.02 sec)

УПРАЖНЕНИЕ 9.3

Соедините следующий запрос с подзапросом к таблице film_actor, чтобы показать уровень мастерства каждого актера:

```

SELECT 'Hollywood Star' level, 30 min_roles, 99999 max_roles
UNION ALL
SELECT 'Prolific Actor' level, 20 min_roles, 29 max_roles
UNION ALL
SELECT 'Newcomer' level, 1 min_roles, 19 max_roles

```

Подзапрос к таблице film_actor должен подсчитывать количество строк для каждого актера с использованием group by actor_id, и результат подсчета должен сравниваться со столбцами min_roles/max_roles, чтобы определить, какой уровень мастерства имеет каждый актер.

```

mysql> SELECT actr.actor_id, grps.level
-> FROM
-> (SELECT actor_id, count(*) num_roles
->   FROM film_actor
->   GROUP BY actor_id
-> ) actr
-> INNER JOIN
-> (SELECT 'Hollywood Star' level, 30 min_roles, 99999 max_roles
-> UNION ALL
-> SELECT 'Prolific Actor' level, 20 min_roles, 29 max_roles

```

```

-> UNION ALL
-> SELECT 'Newcomer' level, 1 min_roles, 19 max_roles
-> ) grps
-> ON actr.num_roles BETWEEN grps.min_roles AND grps.max_roles;
+-----+
| actor_id | level      |
+-----+
|     1    | Newcomer   |
|     2    | Prolific Actor |
|     3    | Prolific Actor |
|     4    | Prolific Actor |
|     5    | Prolific Actor |
|     6    | Prolific Actor |
|     7    | Hollywood Star |
| ...
| 195   | Prolific Actor |
| 196   | Hollywood Star |
| 197   | Hollywood Star |
| 198   | Hollywood Star |
| 199   | Newcomer   |
| 200   | Prolific Actor |
+-----+
200 rows in set (0.03 sec)

```

Глава 10

УПРАЖНЕНИЕ 10.1

Используя следующие определения таблиц и данные, напишите запрос, который возвращает имя каждого клиента вместе с его суммами платежей:

Customer:

Customer_id	Name
1	John Smith
2	Kathy Jones
3	Greg Oliver

Payment:

Payment_id	Customer_id	Amount
101	1	8.99
102	3	4.99
103	1	7.99

Включите в результатирующий набор всех клиентов, даже если для клиента нет записей о платежах.

```

mysql> SELECT c.name, sum(p.amount)
-> FROM customer c LEFT OUTER JOIN payment p
-> ON c.customer_id = p.customer_id

```

```

-> GROUP BY c.name;
+-----+
| name      | sum(p.amount) |
+-----+
| John Smith |      16.98 |
| Kathy Jones |      NULL |
| Greg Oliver |      4.99 |
+-----+
3 rows in set (0.00 sec)

```

УПРАЖНЕНИЕ 10.2

Измените запрос из упражнения 10.1 таким образом, чтобы использовать другой тип внешнего соединения (например, если вы использовали левое внешнее соединение в упражнении 10.1, на этот раз используйте правое внешнее соединение) так, чтобы результаты были идентичны полученным ранее.

```

MySQL> SELECT c.name, sum(p.amount)
-> FROM payment p RIGHT OUTER JOIN customer c
-> ON c.customer_id = p.customer_id
-> GROUP BY c.name;
+-----+
| name      | sum(p.amount) |
+-----+
| John Smith |      16.98 |
| Kathy Jones |      NULL |
| Greg Oliver |      4.99 |
+-----+
3 rows in set (0.00 sec)

```

УПРАЖНЕНИЕ 10.3

Разработайте запрос, который будет генерировать набор {1, 2, 3, ..., 99, 100}. (Указание: используйте перекрестное соединение как минимум с двумя подзапросами в предложении 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

УПРАЖНЕНИЕ 11.1

Перепишите следующий запрос, в котором используется простое выражение case, так, чтобы получить те же результаты с использованием поискового выражения case. Страйтесь, насколько это возможно, использовать как можно меньше предложений when.

```
SELECT name,
CASE name
    WHEN 'English' THEN 'latin1'
    WHEN 'Italian' THEN 'latin1'
    WHEN 'French' THEN 'latin1'
    WHEN 'German' THEN 'latin1'
    WHEN 'Japanese' THEN 'utf8'
    WHEN 'Mandarin' THEN 'utf8'
    ELSE 'Unknown'
END character_set
FROM language;

SELECT name,
CASE
    WHEN name IN ('English','Italian','French','German')
        THEN 'latin1'
    WHEN name IN ('Japanese','Mandarin')
        THEN 'utf8'
    ELSE 'Unknown'
END character_set
FROM language;
```

УПРАЖНЕНИЕ 11.2

Перепишите следующий запрос так, чтобы результирующий набор содержал одну строку с пятью столбцами (по одному для каждого рейтинга). Назовите эти пять столбцов G, PG, PG_13, R и NC_17.

```

mysql> SELECT rating, count(*)
-> FROM film
-> GROUP BY rating;
+-----+
| rating | count(*) |
+-----+
| PG     |      194 |
| G      |      178 |
| NC-17  |      210 |
| PG-13  |      223 |
| R      |      195 |
+-----+
5 rows in set (0.00 sec)

mysql> SELECT
->   sum(CASE WHEN rating = 'G' THEN 1 ELSE 0 END) g,
->   sum(CASE WHEN rating = 'PG' THEN 1 ELSE 0 END) pg,
->   sum(CASE WHEN rating = 'PG-13' THEN 1 ELSE 0 END) pg_13,
->   sum(CASE WHEN rating = 'R' THEN 1 ELSE 0 END) r,
->   sum(CASE WHEN rating = 'NC-17' THEN 1 ELSE 0 END) nc_17
-> FROM film;
+-----+-----+-----+-----+
| g   | pg  | pg_13 | r   | nc_17 |
+-----+-----+-----+-----+
| 178 | 194 | 223  | 195 | 210  |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

```

Глава 12

УПРАЖНЕНИЕ 12.1

Создайте логическую единицу работы для перевода 50 долларов со счета 123 на счет 789. Для этого вставьте две строки в таблицу transaction и обновите две строки в таблице account. Используйте следующие определения/данные таблиц:

Account:

account_id	avail_balance	last_activity_date
123	500	2019-07-10 20:53:27
789	75	2019-06-22 15:18:35

Transaction:

txn_id	txn_date	account_id	txn_type_cd	amount
1001	2019-05-15	123	C	500
1002	2019-06-01	789	C	75

Используйте `txn_type_cd = 'C'`, чтобы указать операцию кредита (дебетование на счет), и `txn_type_cd = 'D'` для обозначения дебета (снятия со счета).

```
START TRANSACTION;

INSERT INTO transaction
  (txnid, txn_date, account_id, txn_type_cd, amount)
VALUES
  (1003, now(), 123, 'D', 50);

INSERT INTO transaction
  (txnid, txn_date, account_id, txn_type_cd, amount)
VALUES
  (1004, now(), 789, 'C', 50);

UPDATE account
SET avail_balance = available_balance - 50,
  last_activity_date = now()
WHERE account_id = 123;

UPDATE account
SET avail_balance = available_balance + 50,
  last_activity_date = now()
WHERE account_id = 789;

COMMIT;
```

Глава 13

УПРАЖНЕНИЕ 13.1

Сгенерируйте инструкцию `alter table` для таблицы `rental` так, чтобы генерировалось сообщение об ошибке, когда строка со значением, имеющимся в столбце `rent.customer_id`, удаляется из таблицы `customer`.

```
ALTER TABLE rental
ADD CONSTRAINT fk_rental_customer_id FOREIGN KEY (customer_id)
REFERENCES customer (customer_id) ON DELETE RESTRICT;
```

УПРАЖНЕНИЕ 13.2

Создайте многостолбцовый индекс в таблице `payment`, который может использоваться обоими приведенными далее запросами:

```
SELECT customer_id, payment_date, amount
FROM payment
WHERE payment_date > cast('2019-12-31 23:59:59' as datetime);

SELECT customer_id, payment_date, amount
```

```

FROM payment
WHERE payment_date > cast('2019-12-31 23:59:59' as datetime)
AND amount < 5;

SELECT customer_id, payment_date, amount
FROM payment
WHERE payment_date > cast('2019-12-31 23:59:59' as datetime);

SELECT customer_id, payment_date, amount
FROM payment
WHERE payment_date > cast('2019-12-31 23:59:59' as datetime)
AND amount < 5;

CREATE INDEX idx_payment01
ON payment (payment_date, amount);

```

Глава 14

УПРАЖНЕНИЕ 14.1

Создайте определение представления, которое можно использовать с помощью следующего запроса для генерации приведенного результирующего набора:

```

SELECT title, category_name, first_name, last_name
FROM film_ctgry_actor
WHERE last_name = 'FAWCETT';
+-----+-----+-----+-----+
| title          | category_name | first_name | last_name |
+-----+-----+-----+-----+
| ACE GOLDFINGER | Horror        | BOB        | FAWCETT    |
| ADAPTATION HOLES | Documentary   | BOB        | FAWCETT    |
| CHINATOWN GLADIATOR | New         | BOB        | FAWCETT    |
| CIRCUS YOUTH | Children      | BOB        | FAWCETT    |
| CONTROL ANTHEM | Comedy        | BOB        | FAWCETT    |
| DARES PLUTO | Animation     | BOB        | FAWCETT    |
| DARN FORRESTER | Action        | BOB        | FAWCETT    |
| DAZED PUNK | Games         | BOB        | FAWCETT    |
| DYNAMITE TARZAN | Classics      | BOB        | FAWCETT    |
| HATE HANDICAP | Comedy        | BOB        | FAWCETT    |
| HOMICIDE PEACH | Family        | BOB        | FAWCETT    |
| JACKET FRISCO | Drama         | BOB        | FAWCETT    |
| JUMANJI BLADE | New          | BOB        | FAWCETT    |
| LAWLESS VISION | Animation     | BOB        | FAWCETT    |
| LEATHERNECKS DWARFS | Travel       | BOB        | FAWCETT    |
| OSCAR GOLD | Animation     | BOB        | FAWCETT    |
| PELICAN COMFORTS | Documentary   | BOB        | FAWCETT    |
| PERSONAL LADYBUGS | Music         | BOB        | FAWCETT    |
| RAGING AIRPLANE | Sci-Fi        | BOB        | FAWCETT    |
| RUN PACIFIC | New          | BOB        | FAWCETT    |
| RUNNER MADIGAN | Music         | BOB        | FAWCETT    |

```

SADDLE ANTITRUST	Comedy	BOB	FAWCETT
SCORPION APOLLO	Drama	BOB	FAWCETT
SHAWSHANK BUBBLE	Travel	BOB	FAWCETT
TAXI KICK	Music	BOB	FAWCETT
BERETS AGENT	Action	JULIA	FAWCETT
BOILED DARES	Travel	JULIA	FAWCETT
CHISUM BEHAVIOR	Family	JULIA	FAWCETT
CLOSER BANG	Comedy	JULIA	FAWCETT
DAY UNFAITHFUL	New	JULIA	FAWCETT
HOPE TOOTSIE	Classics	JULIA	FAWCETT
LUKE MUMMY	Animation	JULIA	FAWCETT
MULAN MOON	Comedy	JULIA	FAWCETT
OPUS ICE	Foreign	JULIA	FAWCETT
POLLOCK DELIVERANCE	Foreign	JULIA	FAWCETT
RIDGEMONT SUBMARINE	New	JULIA	FAWCETT
SHANGHAI TYCOON	Travel	JULIA	FAWCETT
SHAWSHANK BUBBLE	Travel	JULIA	FAWCETT
THEORY MERMAID	Animation	JULIA	FAWCETT
WAIT CIDER	Animation	JULIA	FAWCETT

40 rows in set (0.00 sec)

```
CREATE VIEW film_ctgry_actor
AS
SELECT f.title,
       c.name category_name,
       a.first_name,
       a.last_name
  FROM film f
 INNER JOIN film_category fc
   ON f.film_id = fc.film_id
 INNER JOIN category c
   ON fc.category_id = c.category_id
 INNER JOIN film_actor fa
   ON fa.film_id = f.film_id
 INNER JOIN actor a
   ON fa.actor_id = a.actor_id;
```

УПРАЖНЕНИЕ 14.2

Менеджер компании по прокату фильмов хотел бы иметь отчет, который включает в себя название каждой страны, а также общие платежи всех клиентов, которые живут в данной стране. Создайте определение представления, которое запрашивает таблицу country и использует для вычисления значения столбца tot_payments скалярный подзапрос.

```
CREATE VIEW country_payments
AS
SELECT c.country,
       (SELECT sum(p.amount)
```

```

FROM city ct
    INNER JOIN address a
        ON ct.city_id = a.city_id
    INNER JOIN customer cst
        ON a.address_id = cst.address_id
    INNER JOIN payment p
        ON cst.customer_id = p.customer_id
    WHERE ct.country_id = c.country_id
) tot_payments
FROM country c

```

Глава 15

УПРАЖНЕНИЕ 15.1

Напишите запрос, который перечисляет все индексы в схеме Sakila. Не забудьте включить в результаты имена таблиц.

```

mysql> SELECT DISTINCT table_name, index_name
    -> FROM information_schema.statistics
    -> WHERE table_schema = 'sakila';

```

TABLE_NAME	INDEX_NAME
actor	PRIMARY
actor	idx_actor_last_name
address	PRIMARY
address	idx_fk_city_id
address	idx_location
category	PRIMARY
city	PRIMARY
city	idx_fk_country_id
country	PRIMARY
film	PRIMARY
film	idx_title
film	idx_fk_language_id
film	idx_fk_original_language_id
film_actor	PRIMARY
film_actor	idx_fk_film_id
film_category	PRIMARY
film_category	fk_film_category_category
film_text	PRIMARY
film_text	idx_title_description
inventory	PRIMARY
inventory	idx_fk_film_id
inventory	idx_store_id_film_id
language	PRIMARY
staff	PRIMARY
staff	idx_fk_store_id
staff	idx_fk_address_id
store	PRIMARY

```

| store          | idx_unique_manager      |
| store          | idx_fk_address_id       |
| customer      | PRIMARY                  |
| customer      | idx_email                |
| customer      | idx_fk_store_id         |
| customer      | idx_fk_address_id       |
| customer      | idx_last_name           |
| customer      | idx_full_name           |
| rental         | PRIMARY                  |
| rental         | rental_date              |
| rental         | idx_fk_inventory_id     |
| rental         | idx_fk_customer_id      |
| rental         | idx_fk_staff_id          |
| payment        | PRIMARY                  |
| payment        | idx_fk_staff_id          |
| payment        | idx_fk_customer_id      |
| payment        | fk_payment_rental       |
| payment        | idx_payment01            |
+-----+
45 rows in set (0.00 sec)

```

УПРАЖНЕНИЕ 15.2

Напишите запрос, генерирующий вывод, который можно было бы использовать для создания всех индексов таблицы sakila.customer. Вывод должен иметь вид

```
"ALTER TABLE <таблица> ADD INDEX <индекс> (<список_столбцов>)"
```

Вот одно решение, использующее предложение with:

```

mysql> WITH idx_info AS
    -> (SELECT s1.table_name, s1.index_name,
    ->       s1.column_name, s1.seq_in_index,
    ->       (SELECT max(s2.seq_in_index)
    ->          FROM information_schema.statistics s2
    ->         WHERE s2.table_schema = s1.table_schema
    ->           AND s2.table_name = s1.table_name
    ->           AND s2.index_name = s1.index_name) num_columns
    ->   FROM information_schema.statistics s1
    ->  WHERE s1.table_schema = 'sakila'
    ->    AND s1.table_name = 'customer'
    ->  )
    -> SELECT concat(
    ->   CASE
    ->     WHEN seq_in_index = 1 THEN
    ->       concat('ALTER TABLE ', table_name, ' ADD INDEX ',
    ->               index_name, '(', column_name)
    ->     ELSE concat(' , ', column_name)
    ->   END,
    ->   CASE

```

```

->      WHEN seq_in_index = num_columns THEN '' ;
->      ELSE ''
->    END
->  ) index_creation_statement
-> FROM idx_info
-> ORDER BY index_name, seq_in_index;
+
+-----+
| index_creation_statement
+-----+
| ALTER TABLE customer ADD INDEX idx_email (email);
| ALTER TABLE customer ADD INDEX idx_fk_address_id (address_id);
| ALTER TABLE customer ADD INDEX idx_fk_store_id (store_id);
| ALTER TABLE customer ADD INDEX idx_full_name (last_name
| , first_name);
| ALTER TABLE customer ADD INDEX idx_last_name (last_name);
| ALTER TABLE customer ADD INDEX PRIMARY (customer_id);
+
+-----+
7 rows in set (0.00 sec)

```

Однако после прочтения главы 16, “Аналитические функции”, вы сможете написать следующее решение:

```

mysql> SELECT concat('ALTER TABLE ', table_name, ' ADD INDEX ',
->      index_name, ' (',
->      group_concat(column_name order by
->                      seq_in_index separator ', '),
->      ')';
->  ) index_creation_statement
-> FROM information_schema.statistics
-> WHERE table_schema = 'sakila'
->   AND table_name = 'customer'
-> GROUP BY table_name, index_name;
+
+-----+
| index_creation_statement
+-----+
| ALTER TABLE customer ADD INDEX idx_email (email);
| ALTER TABLE customer ADD INDEX idx_fk_address_id (address_id);
| ALTER TABLE customer ADD INDEX idx_fk_store_id (store_id);
| ALTER TABLE customer ADD INDEX idx_full_name (last_name,
|                                         first_name);
| ALTER TABLE customer ADD INDEX idx_last_name (last_name);
| ALTER TABLE customer ADD INDEX PRIMARY (customer_id);
+
+-----+
6 rows in set (0.00 sec)

```

Глава 16

Для всех упражнений этого раздела используйте набор данных из таблицы Sales_Fact.

```

Sales_Fact
+-----+-----+-----+
| year_no | month_no | tot_sales |
+-----+-----+-----+
| 2019 | 1 | 19228 |
| 2019 | 2 | 18554 |
| 2019 | 3 | 17325 |
| 2019 | 4 | 13221 |
| 2019 | 5 | 9964 |
| 2019 | 6 | 12658 |
| 2019 | 7 | 14233 |
| 2019 | 8 | 17342 |
| 2019 | 9 | 16853 |
| 2019 | 10 | 17121 |
| 2019 | 11 | 19095 |
| 2019 | 12 | 21436 |
| 2020 | 1 | 20347 |
| 2020 | 2 | 17434 |
| 2020 | 3 | 16225 |
| 2020 | 4 | 13853 |
| 2020 | 5 | 14589 |
| 2020 | 6 | 13248 |
| 2020 | 7 | 8728 |
| 2020 | 8 | 9378 |
| 2020 | 9 | 11467 |
| 2020 | 10 | 13842 |
| 2020 | 11 | 15742 |
| 2020 | 12 | 18636 |
+-----+-----+-----+
24 rows in set (0.00 sec)

```

УПРАЖНЕНИЕ 16.1

Напишите запрос, который извлекает каждую строку из Sales_Fact и добавляет столбец для генерации рейтинга на основе значений столбца tot_sales. Самое высокое значение должно получить рейтинг 1, а самое низкое — рейтинг 24.

```

mysql> SELECT year_no, month_no, tot_sales,
-> rank() over (order by tot_sales desc) sales_rank
-> FROM sales_fact;
+-----+-----+-----+-----+
| year_no | month_no | tot_sales | sales_rank |
+-----+-----+-----+-----+
| 2019 | 12 | 21436 | 1 |
| 2020 | 1 | 20347 | 2 |
| 2019 | 1 | 19228 | 3 |
| 2019 | 11 | 19095 | 4 |
| 2020 | 12 | 18636 | 5 |
| 2019 | 2 | 18554 | 6 |

```

```

| 2020 | 2 | 17434 | 7 |
| 2019 | 8 | 17342 | 8 |
| 2019 | 3 | 17325 | 9 |
| 2019 | 10 | 17121 | 10 |
| 2019 | 9 | 16853 | 11 |
| 2020 | 3 | 16225 | 12 |
| 2020 | 11 | 15742 | 13 |
| 2020 | 5 | 14589 | 14 |
| 2019 | 7 | 14233 | 15 |
| 2020 | 4 | 13853 | 16 |
| 2020 | 10 | 13842 | 17 |
| 2020 | 6 | 13248 | 18 |
| 2019 | 4 | 13221 | 19 |
| 2019 | 6 | 12658 | 20 |
| 2020 | 9 | 11467 | 21 |
| 2019 | 5 | 9964 | 22 |
| 2020 | 8 | 9378 | 23 |
| 2020 | 7 | 8728 | 24 |
+-----+

```

24 rows in set (0.02 sec)

УПРАЖНЕНИЕ 16.2

Измените запрос из предыдущего упражнения так, чтобы генерировались два набора рейтингов от 1 до 12: один — для 2019 года и один — для 2020 года.

```

mysql> SELECT year_no, month_no, tot_sales,
->   rank() over (partition by year_no
->                 order by tot_sales desc) sales_rank
-> FROM sales_fact;
+-----+-----+-----+-----+
| year_no | month_no | tot_sales | sales_rank |
+-----+-----+-----+-----+
| 2019 | 12 | 21436 | 1 |
| 2019 | 1 | 19228 | 2 |
| 2019 | 11 | 19095 | 3 |
| 2019 | 2 | 18554 | 4 |
| 2019 | 8 | 17342 | 5 |
| 2019 | 3 | 17325 | 6 |
| 2019 | 10 | 17121 | 7 |
| 2019 | 9 | 16853 | 8 |
| 2019 | 7 | 14233 | 9 |
| 2019 | 4 | 13221 | 10 |
| 2019 | 6 | 12658 | 11 |
| 2019 | 5 | 9964 | 12 |
| 2020 | 1 | 20347 | 1 |
| 2020 | 12 | 18636 | 2 |
| 2020 | 2 | 17434 | 3 |
| 2020 | 3 | 16225 | 4 |

```

```

| 2020 | 11 | 15742 | 5 |
| 2020 | 5 | 14589 | 6 |
| 2020 | 4 | 13853 | 7 |
| 2020 | 10 | 13842 | 8 |
| 2020 | 6 | 13248 | 9 |
| 2020 | 9 | 11467 | 10 |
| 2020 | 8 | 9378 | 11 |
| 2020 | 7 | 8728 | 12 |
+-----+

```

24 rows in set (0.00 sec)

УПРАЖНЕНИЕ 16.3

Напишите запрос, который извлекает все данные за 2020 год, и включите столбец, который будет содержать значение tot_sales для предыдущего месяца.

```
mysql> SELECT year_no, month_no, tot_sales,
->     lag(tot_sales) over (order by month_no) prev_month_sales
->     FROM sales_fact
->     WHERE year_no = 2020;
```

```

+-----+
| year_no | month_no | tot_sales | prev_month_sales |
+-----+
| 2020 | 1 | 20347 | NULL |
| 2020 | 2 | 17434 | 20347 |
| 2020 | 3 | 16225 | 17434 |
| 2020 | 4 | 13853 | 16225 |
| 2020 | 5 | 14589 | 13853 |
| 2020 | 6 | 13248 | 14589 |
| 2020 | 7 | 8728 | 13248 |
| 2020 | 8 | 9378 | 8728 |
| 2020 | 9 | 11467 | 9378 |
| 2020 | 10 | 13842 | 11467 |
| 2020 | 11 | 15742 | 13842 |
| 2020 | 12 | 18636 | 15742 |
+-----+

```

12 rows in set (0.00 sec)

Предметный указатель

A

all 75, 202
any 203
Apache Drill 34
работа с MySQL 353
asc 87

B

between 98
bigint 45

C

case 239
char 41
clob 44, 146

D

date 47, 169
datetime 47, 169
delete 61
desc 53, 87
describe 53
distinct 74, 185
double 46

E

exists 207
explain 273

F

float 46
from 75

G

group by 84

H

Hadoop 346

I

in 199
insert 57

int 45
interval 173

J

JSON 27, 347

L

like 105
longtext 44

M

mediumint 45
mediumtext 44
MySQL 37, 42, 122, 145
блокировка 253
вставка в строку 158
деление на нуль 247
диапазон дат 48
метаданные 296
механизмы хранения 259
одинарные кавычки 149
создание индекса 265
специальные символы 149
транзакции 256
удаление индекса 267
удаление строк 210

N

NoSQL 347
not 93
null 107, 186

O

Oracle Database 33, 40, 41, 122, 145, 158
блокировка 253
виртуальная частная база данных 285
вставка в строку 159
деление на нуль 247
диапазон дат 48
итоговые данные 192
конкатенация строк 151
метаданные 296
настройка часовогопояса 168

одинарные кавычки 149
поиск подстрок 154
символьные данные 44
создание индекса 265
специальные символы 150
транзакции 256
удаление индекса 267
order by 85

P

PL/SQL 30

R

regexp 107

S

select 70
smallint 45
SQL
 динамическое выполнение 306
SQL Server 23, 33, 44, 122, 145, 158
блокировка 253
 вставка в строку 159
диапазон дат 48
конкатенация строк 151
метаданные 296
символьные данные 44
создание индекса 265
специальные символы 149
транзакции 256
удаление индекса 267

T

text 44, 146
time 47, 169
timestamp 47, 169
tinyint 45
tinytext 44
Transact-SQL 30

U

update 61
UTC 167

V

varchar 41, 146

W

when 241

where 81

X

XML 27, 60, 347

Y

year 47

A

Автоинкремент 56
Автофиксация 256
Атомарность 254

Б

База данных 19
 секционирование 331
Блокировка 252
 гранулярность 253
 записи 252
 чтения 252
Большие данные 346

В

Взаимоблокировка 258
Выражение case 240
 поисковое 240
 простое 242

Д

Декартово произведение 115, 228

З

Запись 26
Запрос
 группировка 84
 оптимизатор 68
 план выполнения 68, 273
 подзапрос 76, 195
 представление 78
 содержащий 77, 195
 сортировка 85
 составной 133
 удаление дубликатов 74
 фильтрация 81
from 75
select 70

И

Иерархия

с одним родителем 21
со многими родителями 21
Индекс 264
битовый 270
глобальный 333
локальный 333
многостолбцовый 269
полнотекстовый 272
секционирование 333
создание 264
уникальный 268
B-tree 269
История SQL 26
Итоговые данные 190

К

Каскадное обновление 278
Кластеризация 344
Ключ
внешний 25, 26
естественный 24
первичный 24, 26
составной 24
суррогатный 24
Комментарий 31

Л

Летнее время 166

М

Масштаб 46
Метаданные 28, 295
Механизм хранения 259
Множество 129
объединение 130
пересечение 130
теория 129

Н

Набор символов 42
Нормализация 25, 113

О

Обобщенное табличное выражение 217
Ограничение 52, 275
внешнего ключа 55
первичного ключа 52
проверочное 52
типы 275

Окна данных 312
рамки 324
Оптимизатор 29, 68

П

Подзапрос 76, 195
коррелированный 197, 206
некоррелированный 197
скалярный 197, 218
Представление 78, 281
обновляемое 288
Псевдоним столбца 73

Р

Ранжирование 315
Регулярные выражения 106
Результирующий набор 27, 68
Реляционная модель 22

С

Самосоединение 126
Секционирование 331
вертикальное 333
горизонтальное 333
индекса 333
ключ 333
композитное 341
отсечение разделов 343
по диапазону 334
по списку 337
по хешу 339
функция 333
Системный каталог 296
Сканирование таблицы 264
Словарь данных 28, 296
Соединение 25, 113, 114
внешнее 224
левое 225, 226
правое 226
трехстороннее 227
естественное 235
перекрестное 115, 205, 229
Скрытие таблиц 284
Сортировка 85, 87
Столбец 23, 26
псевдоним 73
Страница 253
Строка 23, 26
СУБД 20
иерархическая 20

Т

Таблица 26, 76
ведущая 122
виртуальная 76, 78
временная 77
постоянная 76
производная 76
псевдоним 80
связь 79
секционирование 332
соединение 80
Тип данных 40
временной 46, 166
символьный 41
строковый 145
текстовый 44
числовой 45, 160
Точность 46, 163
Транзакция 254
точка сохранения 259

У

Управление версиями 252
Условная логика 239

Ф

Фильтрация 91
групповая 192
условие 91
диапазона 97
неравенства 96
равенства 95
регулярное выражение 106
соответствия 104
членства 102

Функция

агрегатная 180, 183
аналитическая 311
математическая 161
отчетности 321
ранжирования 315
хеширования 339
avg() 183
cast() 170, 177
ceil() 163
ceiling() 163
charindex() 154
concat() 157
count() 181, 183
current_date() 172

current_time() 172
current_timestamp() 172
date_add() 173
datediff() 176
dense_rank() 315
extract() 175
floor() 163
getutcdate() 167
insert() 158
lag() 327
last_day() 174
lead() 327
len() 152
length() 152
locate() 153
max() 183
min() 183
mod() 161
now() 40
position() 153
pow() 162
power() 162
rank() 315
replace() 159
round() 164
row_number() 315
strcmp() 154
str_to_date() 171
stuff() 159
substr() 160
substring() 160
sum() 183
to_date() 172
trunc() 163
truncate() 163, 164
utc_timestamp() 167

Ч

Часовой пояс 166

Ш

Шардинг 344

Дана книга відрізняється широким охопленням як тем (від азів SQL до таких складних питань, як аналітичні функції та робота з великими базами даних), так і конкретних баз даних (MySQL, Oracle Database, SQL Server) і особливостей реалізації тих чи інших функціональних можливостей SQL на цих серверах. Книга ідеально підходить в якості підручника для початківця-розробника в області баз даних. У ній описані всі можливі застосування мови SQL і найбільш поширені сервери баз даних.

Науково-популярне видання

Больє, Алан

Вивчаємо SQL

Генерація, вибірка та обробка даних

3-е видання

(Рос. мовою)

Зав. редакцією С.М. Тригуб

Із загальних питань звертайтеся до видавництва “Діалектика” за адресою:
info@dialektika.com, <http://www.dialektika.com>

Підписано до друку 31.05.2021. Формат 60x90/16

Ум. друк. арк. 25,0. Обл.-вид. арк. 18,5

Зам. № 21-1753

Видавець ТОВ “Комп’ютерне видавництво “Діалектика”

03164, м. Київ, вул. Генерала Наумова, буд. 23-Б.

Свідоцтво суб’єкта видавничої справи ДК № 6758 від 16.05.2019.

Надруковано ТОВ “АЛЬФА ГРАФІК ”

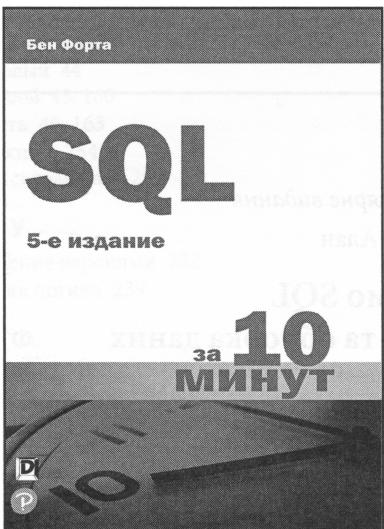
03067, м. Київ, вул. Машинобудівна, 42

Свідоцтво суб’єкта видавничої справи ДК № 6838 від 09.07.2019.

SQL ЗА 10 МИНУТ

5-е издание

Бен Форта



www.williamspublishing.com

Эксперт по базам данных Бен Форта расскажет обо всем, что касается основ SQL: от простых запросов на выборку данных до более сложных тем, таких как соединения, подзапросы, хранимые процедуры, курсоры, триггеры и табличные ограничения. Все темы последовательно излагаются в виде простых и коротких уроков, на каждый из которых уйдет не более 10 минут. Большинство уроков дополняется упражнениями, предназначенными для закрепления материала.

В книге:

- основные инструкции SQL;
- создание сложных запросов с множеством предложений и операторов;
- извлечение, сортировка и форматирование данных;
- фильтрация результатов запроса;
- применение итоговых функций для получения сводных данных;
- соединение таблиц;
- добавление, обновление и удаление данных;
- создание и изменение таблиц;
- работа с представлениями, хранимыми процедурами и триггерами.

ISBN 978-5-907365-67-4 в продаже

Изучаем SQL

Как только ваша компания сталкивается с большими потоками данных, их необходимо сразу же заставить работать на пользу компании (и всего человечества), и лучший инструмент для этого — язык SQL. В третьем издании данного руководства его автор, Алан Болье, помогает разработчикам освоить основы SQL для написания приложений, работающих с базами данных, выполнения административных задач и создания отчетов. В этом издании вы найдете новые главы, посвященные аналитическим функциям, стратегиям работы с большими базами данных и связям SQL с большими данными.

Каждая глава представляет собой самостоятельный урок по той или иной ключевой концепции или методике SQL, в котором используются многочисленные иллюстрации и примеры с комментариями. Приводимые в конце глав упражнения позволяют применить изученный материал на практике. С помощью этой книги вы приобретете необходимые для эффективной работы с данными знания и сможете быстро применить всю мощь и гибкость языка SQL для практической работы. В книге, в частности, раскрыты следующие темы.

- Изучение основ SQL и некоторых его расширенных возможностей
- Использование инструкций SQL для генерации, выборки и обработки данных
- Создание объектов баз данных, таких как таблицы, индексы и ограничения, с помощью инструкций схемы SQL
- Взаимодействие наборов данных с запросами; применение подзапросов и их важность для построения сложных запросов
- Преобразование и манипулирование данными с помощью встроенных функций SQL и применения условной логики в инструкциях данных

“От азов SQL до таких сложных тем, как аналитические функции и работа с большими базами данных, — третье издание этой книги обеспечивает читателя всем необходимым материалом, который нужно знать для эффективного использования SQL в современном мире баз данных”.

— Марк Ричардс,
автор книги *Fundamentals of Software Architecture*

Алан Болье более 25 лет проектировал и создавал приложения для работы с базами данных. Он — один из авторов книги *Mastering Oracle SQL* и автор онлайн-курса по SQL Калифорнийского университета. Алан создал собственную консалтинговую компанию, которая специализируется на проектировании и разработке баз данных в области финансов и телекоммуникаций.

ISBN 978-617-7987-01-6



9 786177 987016

Ком'ютерне видавництво
“ДІАЛЕКТИКА”
www.dialektika.com

<http://oreilly.com>