ИЗУЧАЕМ SQL

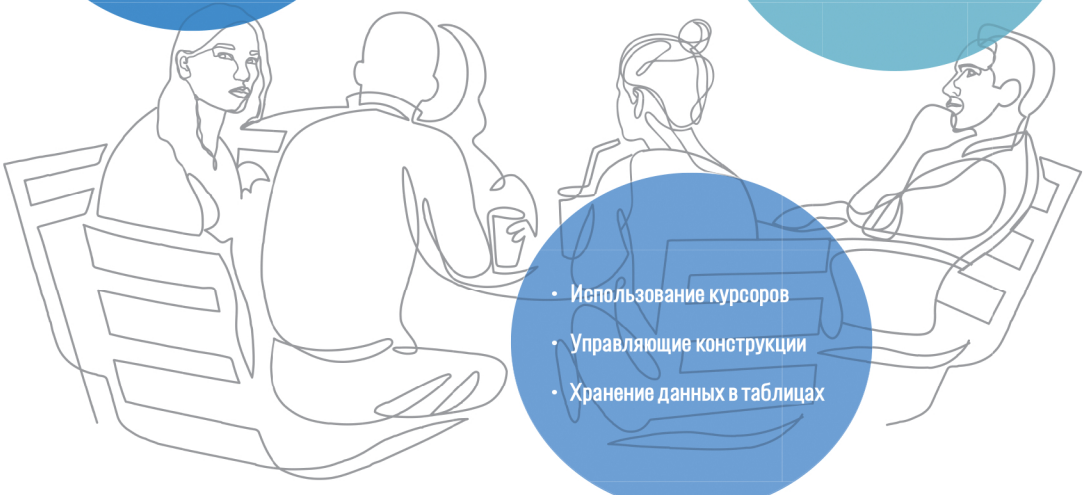
ЗА МЕСЯЦ, ЗАНИМАЯСЬ ОДИН ЧАС В ДЕНЬ

24 коротких понятных урока

- Подстановочные знаки и NULL-значения
- Запросы к таблицам
- Фильтрация и сортировка

- Логические операторы
- Группировка данных
- Работа с переменными

- Операции с данными
- Применение функций
- Создание ограничений



- Использование курсоров
- Управляющие конструкции
- Хранение данных в таблицах

ДЖЕФФ ЯНУЧЧИ

MANNING

Learn SQL in a Month of Lunches

Jeff Iannucci



MANNING
Shelter Island

Изучаем SQL за месяц, занимаясь один час в день

Джефф Януччи

ПИТЕР®

Санкт-Петербург • Москва • Минск

2026

Джефф Януччи

Изучаем SQL за месяц, занимаясь один час в день

Серия «Библиотека программиста»

Перевел с английского С. Гусев

Научный редактор И. Кадыков

ББК 32.973.2-018.1
УДК 004.434

Януччи Джефф

Я63 Изучаем SQL за месяц, занимаясь один час в день. — СПб.: Питер, 2026. — 352 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-4494-5

SQL, «структурированный язык запросов» (Structured Query Language) — это универсальное средство создания, управления и составления запросов к реляционным базам данных, таким как SQL Server, PostgreSQL и Oracle. Для аналитиков данных SQL — суперинструмент, позволяющий выйти за пределы обычных табличных редакторов и систем бизнес-аналитики. При этом язык SQL интуитивен и на удивление прост: немного практики во время чтения книги — и вы уже с легкостью извлекаете данные, перестраиваете таблицы и создаете яркие, наглядные отчеты и презентации!

Книга предназначена для аналитиков данных и начинающих специалистов в сфере информационных технологий, включая тех, кто не имеет ни малейшего опыта работы с реляционными базами данных. В ней вас ждут 24 компактных урока, посвященных ключевым аспектам владения языком SQL, таким как извлечение, фильтрация и обработка данных. Каждый урок завершается практической работой, рассчитанной примерно на 15 минут. Вы и не заметите, как научитесь составлять эффективные запросы, возвращающие именно те данные, что вам нужны. Постепенно у вас сформируется ценное интуитивное понимание того, как базы данных функционируют в реальных бизнес-сценариях.

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

ISBN 978-1633438576 англ.
ISBN 978-5-4461-4494-5

Authorized translation of the English edition © 2024 Manning Publications. This translation is published and sold by permission of Manning Publications, the owner of all rights to publish and sell the same.
© Перевод на русский язык ООО «Прогресс книга», 2026
© Издание на русском языке, оформление ООО «Прогресс книга», 2026
© Серия «Библиотека программиста», 2026

Права на издание получены по соглашению с Manning Publications. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. В книге возможны упоминания организаций, деятельность которых запрещена на территории Российской Федерации, таких как Meta Platforms Inc., Facebook, Instagram и др. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 06.2026. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 28.04.26. Формат 70×100/16. Бумага офсетная. Усл. п. л. 28,380. Тираж 1500. Заказ 0000.

Краткое содержание

Глава 1. Прежде чем начать 30

Глава 2. Ваш первый SQL-запрос 36

Глава 3. Извлечение данных 50

Глава 4. Сортировка, выборочное исключение и комментирование данных 62

Глава 5. Выборка по заданным условиям 75

Глава 6. Фильтрация с учетом нескольких значений, диапазонов и исключений 88

Глава 7. Выборка по подстановочным знакам и NULL-значениям 101

Глава 8. Запросы к нескольким таблицам 114

Глава 9. Применение различных видов соединений 131

Глава 10. Объединение запросов с помощью операций над множествами 144

Глава 11. Применение подзапросов и логических операторов 158

Глава 12. Группировка данных 170

Глава 13. Работа с переменными 184

Глава 14. Работа с функциями в запросах 199

Глава 15. Объединение и вычисление значений при помощи функций 211

Глава 16. Вставка данных 225

Глава 17. Обновление и удаление данных 239

Глава 18. Организация и хранение данных в таблицах 252

Глава 19. Создание ограничений и индексов 270

Глава 20. Повторное использование запросов: представления и хранимые процедуры 286

Глава 21. Средства принятия решений в запросах 302

Глава 22. Работа с курсорами 318

Глава 23. Работа со сторонними сценариями 334

Глава 24. Все только начинается 348

Оглавление

От издательства ... 20
О научном редакторе книги ... 20

Предисловие ... 21

Благодарности ... 23

Об этой книге ... 25
Для кого эта книга ... 25
Структура книги ... 26
О программном коде ... 27
Форум liveBook ... 28

Об авторе ... 29

Глава 1. Прежде чем начать ... 30
1.1. Актуальность SQL ... 30
1.2. Подходит ли вам эта книга? ... 31
1.2.1. Спектр применения SQL ... 31
1.2.2. Диалекты SQL ... 32
1.2.3. ИИ и SQL ... 33

1.3. Как построена книга ... 33
1.3.1. Основные главы ... 33
1.3.2. Практические занятия ... 34
1.3.3. Дальнейшее изучение ... 34
1.4. Подготовка рабочего места ... 34
1.4.1. Установка MySQL и MySQL Workbench ... 34
1.4.2. Выполнение практических работ ... 35
1.5. Полезные онлайн-ресурсы ... 35
1.6. SQL в действии — с первого же дня ... 35

Глава 2. Ваш первый SQL-запрос ... 36
2.1. Кто работал с Excel — уже знаком с таблицами БД ... 36
2.2. Изучение SQL похоже на урок английского ... 40
2.3. Ваш первый SQL-запрос ... 41
2.4. Ключевые слова и термины ... 47
2.5. Практическое занятие ... 48
2.6. Ответы ... 49

Глава 3. Извлечение данных ... 50
3.1. Правила работы с инструкцией SELECT ... 50
3.1.1. Требования к SELECT ... 51
3.1.2. Ключевые и зарезервированные слова ... 51
3.1.3. Нечувствительность к регистру ... 52
3.1.4. Форматирование и пробельные символы ... 53
3.2. Извлечение данных из таблицы ... 54
3.2.1. Извлечение отдельного столбца ... 55
3.2.2. Извлечение нескольких столбцов ... 56
3.2.3. Переименование столбцов в выборке при помощи псевдонимов ... 57
3.2.4. Извлечение всех столбцов ... 58

3.3. Практическое занятие ... 61
3.4. Ответы ... 61

Глава 4. Сортировка, выборочное исключение и комментирование данных ... 62
4.1. Сортировка данных ... 62
4.1.1. Сортировка по одному столбцу ... 63
4.1.2. Сортировка по нескольким столбцам ... 65
4.1.3. Указание направления сортировки ... 66
4.1.4. Сортировка по скрытым столбцам ... 67
4.1.5. Сортировка по позиции ... 68
4.2. Ограничение выборки ... 68
4.2.1. Ограничение выборки при помощи LIMIT ... 68
4.2.2. Применение OFFSET к ограниченной выборке данных ... 70
4.2.3. Ограничение выборки в других СУБД ... 71
4.3. Комментарии к данным ... 71
4.4. Практическое занятие ... 73
4.5. Ответы ... 74

Глава 5. Выборка по заданным условиям ... 75
5.1. Фильтрация по одному условию ... 75
5.1.1. Фильтрация по числовым значениям ... 76
5.1.2. Фильтрация по строковым значениям ... 77
5.1.3. Фильтрация по значениям даты и времени ... 79
5.2. Фильтрация по нескольким условиям ... 80
5.2.1. Фильтрация с соблюдением всех условий ... 80
5.2.2. Фильтрация с соблюдением одного из многих условий ... 81
5.2.3. Управление порядком применения нескольких фильтров ... 83
5.2.4. Фильтрация в сочетании с сортировкой ... 85
5.3. Практическое занятие ... 86
5.4. Ответы ... 86

Глава 6. Фильтрация с учетом нескольких значений, диапазонов и исключений ... 88
6.1. Фильтрация по конкретным значениям ... 88
6.2. Фильтрация по диапазону значений ... 90
6.2.1. Фильтрация по открытому диапазону ... 91
6.2.2. Фильтрация по закрытому диапазону ... 92
6.3. Исключение из выборки ... 94
6.3.1. Исключение конкретного значения ... 94
6.3.2. Исключение всего условия фильтрации ... 95
6.4. Сочетание различных типов условий фильтрации ... 97
6.5. Сводный обзор операторов сравнения ... 98
6.6. Практическое занятие ... 99
6.7. Ответы ... 100

Глава 7. Выборка по подстановочным знакам и NULL-значениям ... 101
7.1. Фильтрация при помощи подстановочных знаков ... 101
7.1.1. Фильтрация при помощи знака процента ... 102
7.1.2. Фильтрация с помощью символа подчеркивания ... 105
7.2. Фильтрация по значениям null ... 106
7.2.1. Как не следует выполнять поиск NULL-значений ... 107
7.2.2. Как правильно выполнять поиск NULL-значений ... 108
7.2.3. Как выполнять поиск значений, отличных от NULL ... 110
7.3. Практическое занятие ... 111
7.4. Ответы ... 112

Глава 8. Запросы к нескольким таблицам ... 114
8.1. Принципы построения связей между данными ... 115
8.1.1. Данные без реляционных связей ... 116
8.1.2. Данные с реляционными связями ... 117
8.2. Как соединять данные ... 121

8.2.1. Соединение двух таблиц ... 121
8.2.2. Соединение нескольких таблиц ... 123
8.3. Псевдонимы таблиц ... 125
8.4. Альтернативный способ присоединения данных ... 126
8.5. Практическое занятие ... 128
8.6. Ответы ... 129

Глава 9. Применение различных видов соединений ... 131
9.1. Внутренние соединения ... 131
9.2. Внешние соединения ... 134
9.2.1. Левые внешние соединения ... 134
9.2.2. Правые внешние соединения ... 136
9.2.3. Используем внешние соединения для поиска строк без совпадений ... 137
9.2.4. Смена левых и правых соединений ... 138
9.2.5. Ключевое слово USING ... 139
9.2.6. Естественные соединения ... 140
9.3. Перекрестные (декартовы) соединения ... 141
9.4. Практическое занятие ... 142
9.5. Ответы ... 143

Глава 10. Объединение запросов с помощью операций над множествами ... 144
10.1. Применение операторов над множествами ... 144
10.2. Оператор UNION ... 146
10.3. Оператор UNION ALL ... 149
10.4. Эмуляция операции FULL OUTER JOIN в MySQL ... 150
10.5. Оператор INTERSECT ... 153
10.6. Оператор EXCEPT ... 154
10.7. Практическое занятие ... 155
10.8. Ответы ... 156

Глава 11. Применение подзапросов и логических операторов 158
11.1. Простой подзапрос 158
11.2. Логические операторы и подзапросы 161
11.2.1. Операторы ANY и IN 162
11.2.2. Операторы ALL и NOT IN 163
11.2.3. Операторы EXISTS и NOT EXISTS 165
11.3. Подзапросы в других частях запроса 166
11.3.1. Подзапросы в предложении FROM 166
11.3.2. Подзапросы в предложении SELECT 168
11.4. Практическое занятие 168
11.5. Ответы 169

Глава 12. Группировка данных 170
12.1. Агрегатные функции 170
12.1.1. Функция SUM 171
12.1.2. Функция COUNT 172
12.1.3. Функция MIN 173
12.1.4. Функция MAX 173
12.1.5. Функция AVG 174
12.1.6. Сочетание фильтрации и агрегации 174
12.2. Агрегирование данных посредством GROUP BY 175
12.2.1. Сочетание фильтрации и агрегации 176
12.2.2. Предложение GROUP BY и NULL-значения 177
12.3. Фильтрация при помощи HAVING 178
12.4. Логический порядок обработки SQL-запроса 180
12.5. Ключевое слово DISTINCT 181
12.6. Практическое занятие 182
12.7. Ответы 183

Глава 13. Работа с переменными ... 184
13.1. Пользовательские переменные ... 185
13.1.1. Объявляем нашу первую пользовательскую переменную ... 185
13.1.2. Правила работы с пользовательскими переменными ... 186
13.1.3. Задействуем нашу первую пользовательскую переменную ... 187
13.2. Фильтрация с помощью переменных в предложениях FROM и HAVING ... 188
13.3. Присвоение переменной неизвестного значения ... 190
13.3.1. Разберем, как работает запрос ... 191
13.3.2. Присвоение неизвестного значения переменной посредством SELECT ... 191
13.3.3. Производительность при работе с переменными ... 193
13.3.4. Диагностика и отладка при работе с переменными ... 194
13.4. Еще несколько слов о переменных ... 195
13.4.1. Присвоение литерального значения с помощью SELECT ... 195
13.4.2. Присвоение переменной значения NULL ... 196
13.4.3. Изменение типа данных переменной ... 196
13.5. Практическое занятие ... 197
13.6. Ответы ... 197

Глава 14. Работа с функциями в запросах ... 199
14.1. Минусы применения функций ... 199
14.1.1. Набор команд функций варьируется в каждой СУБД ... 199
14.1.2. Применение функций может быть неэффективным ... 200
14.2. Строковые функции ... 200
14.2.1. Функции для работы с регистром ... 200
14.2.2. Функции удаления пробельных символов (TRIM-функции) ... 202
14.2.3. Прочие строковые функции ... 204

14.3. Функции для работы с датой и временем 205
14.3.1. Функции даты, возвращающие числовые значения 205
14.3.2. Функции даты, возвращающие строковые значения 206
14.3.3. Прочие функции для работы с датой и временем 207
14.4. Информационные функции 207
14.4.1. Сведения о дате и времени 207
14.4.2. Сведения о подключении 209
14.5. Практическое занятие 210
14.6. Ответы 210

Глава 15. Объединение и вычисление значений при помощи функций 211
15.1. Объединение строковых значений 211
15.1.1. Функция CONCAT 212
15.1.2. Функция CONCAT_WS 215
15.1.3. Функция COALESCE 217
15.2. Преобразование значений 218
15.2.1. Функция REPLACE 218
15.2.2. Функции CONVERT и CAST 219
15.3. Функции для математических вычислений 221
15.4. Практическое занятие 223
15.5. Ответы 223

Глава 16. Вставка данных 225
16.1. Вставка конкретных значений 225
16.1.1. Вставка новой строки 226
16.1.2. Вставка нескольких строк 228
16.1.3. Вставка неполной строки 229
16.1.4. Предостережение относительно пропуска столбцов 232
16.2. Вставка строки посредством подзапроса 233
16.3. Вставка строки при помощи переменных 235

16.4. Практическое занятие ... 236
16.5. Ответы ... 237

Глава 17. Обновление и удаление данных ... 239
17.1. Обновление значений ... 239
17.1.1. Выполняем операции с данными в реальном времени ... 240
17.1.2. Обязательные компоненты операции обновления ... 241
17.1.3. Обновляем значения в одном или нескольких столбцах ... 242
17.1.4. Обновляем значения при помощи многотабличного запроса ... 244
17.2. Удаление строк ... 247
17.2.1. Удаляем одну или несколько строк ... 247
17.2.2. Удаляем строки при помощи многотабличного запроса ... 248
17.2.3. Удаляем все строки таблицы ... 249
17.3. Один важный совет по манипулированию данными ... 250
17.4. Практическое занятие ... 250
17.5. Ответы ... 251

Глава 18. Организация и хранение данных в таблицах ... 252
18.1. Создание таблицы ... 252
18.1.1. Что следует учесть ... 252
18.1.2. Создаем пустую таблицу ... 254
18.1.3. Вносим значения в пустую таблицу ... 255
18.2. Модификация таблицы ... 256
18.2.1. Добавляем столбец в таблицу ... 256
18.2.2. Что следует учесть при добавлении нового столбца ... 259
18.3. Первичные ключи ... 260
18.3.1. Что важно знать о первичных ключах ... 260
18.3.2. Добавляем первичный ключ (PRIMARY KEY) ... 261
18.4. Внешние ключи и ограничения ... 263

18.4.1. Диаграммы данных ... 263
18.4.2. Добавляем ограничение внешнего ключа (FOREIGN KEY) ... 265
18.5. Удаление таблицы, столбца или ограничения ... 265
18.5.1. Удаляем ограничение ... 266
18.5.2. Удаляем столбец ... 266
18.5.3. Удаляем таблицу ... 267
18.6. Практическое занятие ... 267
18.7. Ответы ... 268

Глава 19. Создание ограничений и индексов ... 270
19.1. Ограничения ... 270
19.1.1. Ограничения NOT NULL ... 271
19.1.2. Ограничения DEFAULT ... 273
19.1.3. Ограничения UNIQUE ... 275
19.1.4. Ограничения CHECK ... 275
19.2. Автоматическое приращение значений в столбце ... 276
19.3. Индексы ... 277
19.3.1. Кластеризованные индексы ... 278
19.3.2. Некластеризованные индексы ... 280
19.4. Практическое занятие ... 284
19.5. Ответы ... 284

Глава 20. Повторное использование запросов: представления и хранимые процедуры ... 286
20.1. Представления ... 286
20.1.1. Создание представлений ... 287
20.1.2. Фильтрация представлений ... 288
20.1.3. Соединение представлений ... 289
20.1.4. Что важно знать о представлениях ... 291

20.2. Хранимые процедуры ... 292
20.2.1. Создание хранимых процедур ... 292
20.2.2. Применение переменных в хранимых процедурах ... 294
20.2.3. Что важно знать о хранимых процедурах ... 298
20.3. Различия между представлениями и хранимыми процедурами ... 299
20.4. Практическое занятие ... 299
20.5. Ответы ... 300

Глава 21. Средства принятия решений в запросах ... 302
21.1. Условные функции и выражения ... 302
21.1.1. Функция COALESCE ... 302
21.1.2. Функция IFNULL ... 303
21.1.3. Выражение CASE ... 305
21.2. Управляющие конструкции ... 309
21.2.1. IF и THEN ... 309
21.2.2. ELSE ... 312
21.2.3. Набор условий ... 314
21.3. Практическое занятие ... 316
21.4. Ответы ... 317

Глава 22. Работа с курсорами ... 318
22.1. Подробнее о переменных и параметрах ... 318
22.1.1. Переменные в теле хранимых процедур ... 318
22.1.2. Выходные параметры ... 319
22.2. Курсоры ... 320
22.2.1. Устройство курсора ... 320
22.2.2. Создание курсора ... 321
22.3. Альтернативы курсорам ... 326
22.3.1. Цикл WHILE ... 326
22.3.2. Временные таблицы ... 328

22.4. Что важно учитывать при работе с курсорами 330
22.4.1. Мышление в терминах множеств 330
22.4.2. Стоит ли вообще использовать курсоры? 331
22.5. Практическое занятие 331
22.6. Ответы 332

Глава 23. Работа со сторонними сценариями 334
23.1. Сценарий стороннего разработчика: создание таблицы 335
23.1.1. Запрос CREATE TABLE 335
23.1.2. Обзор сценария CREATE TABLE 335
23.1.3. Доработка запроса CREATE TABLE 337
23.2. Сценарий стороннего разработчика: вставка данных 337
23.2.1. Хранимая процедура INSERT 337
23.2.2. Обзор хранимой процедуры INSERT 339
23.2.3. Доработка хранимой процедуры INSERT 343
23.2.4. Дальнейшая доработка хранимой процедуры INSERT 345

Глава 24. Все только начинается 348
24.1. Больше SQL 348
24.2. Прочие SQL-ресурсы 349
24.3. Счастливого пути! 350

Посвящается тебе, дорогой читатель.

И пусть эта книга станет подспорьем
в твоем путешествии по миру данных.

От издательства

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

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

Мы будем рады узнать ваше мнение!

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

О научном редакторе книги

Иван Кадыков — системный аналитик, работает в компании «ОРИОН» на проекте защищенной виртуализации zVirt. Имеет профильное образование в области системного анализа и прикладной информатики. В профессиональной деятельности специализируется на проектировании инфраструктурных решений в сфере виртуализации.

Глубоко изучал внутреннюю архитектуру Microsoft SQL Server: структуры хранения данных (кучи, B-деревья), оптимизацию запросов, уровни изоляции транзакций, занимался проектированием и анализом OLAP- и ROLAP-кубов. В магистерской диссертации разрабатывал автоматизированную обучающую систему для поддержки принятия решений (Python, React, MySQL). По ее результатам опубликованы две научные статьи.

Также работал в сфере Data Science, где получил практический опыт обработки и анализа реальных данных, включая временные ряды и биомедицинские данные (ЭЭГ). Интерес к методологии работы с данными и глубокое понимание SQL стали основой для участия в научном редактировании книги «Изучаем SQL за месяц, занимаясь один час в день».

Предисловие

Мое знакомство с SQL началось еще в конце прошлого века, когда я впервые столкнулся с реляционными базами данных. Я не был программистом — максимум, что я тогда умел, это ввести несколько DOS-команд на домашнем компьютере. Однако, оказавшись перед необходимостью освоить SQL для выполнения рабочих задач, я с удивлением обнаружил, насколько легко писать и запускать сценарии на этом интуитивно понятном языке.

С тех пор я успел поделиться своими знаниями SQL с сотнями коллег. И знаете, что меня всегда удивляло? Большинство из них вовсе не программисты. Это люди из сферы финансов, маркетинга, продаж — специалисты, которым SQL нужен преимущественно для того, чтобы получать оперативный доступ к важной корпоративной информации без долгих ожиданий и посредников. Им не до тонкостей вроде третьей нормальной формы или кортежей. Они просто хотят освоить набор основных команд, чтобы тут же использовать их в работе.

Если вдуматься, многие вещи в жизни мы узнаём именно так. Большинство из нас не училось собирать автомобиль, прежде чем сесть за руль. Мы не посещали кулинарную школу, до того как начали готовить еду. Мы не разбирались, как жесткие диски хранят данные и как процессоры управляют потоками, перед тем как стали пользоваться компьютерами. Вместо этого мы осваивали ряд базовых понятий и приемов, чтобы приступить к делу и в дальнейшем совершенствовать свои знания и навыки на практике.

Вот почему я с таким энтузиазмом взялся за написание этой книги. Как человек, знакомый с несколькими изданиями из серии «За 24 часа», я особенно ценю, что одна из главных ее целей — научить читателя с первых же шагов задействовать полученные знания в решении реальных задач. Эта книга не исключение — она задумана и написана так, чтобы вы могли как можно скорее приступить к работе с SQL. Я надеюсь, что уже к концу второй главы вы будете с увлечением составлять первые запросы, открывать для себя новые возможности языка и пытаться применять его с различными реляционными базами данных.

И хотя ни одна книга не в состоянии охватить все богатство языка SQL, я уверен, что к последним страницам этого руководства любая задача, связанная с выборкой и обработкой данных, будет вам по плечу. Впрочем, это лишь начало вашего путешествия в мир SQL, и я призываю вас не останавливаться на достигнутом. Ищите другие книги, онлайн-публикации и видеоматериалы, чтобы постоянно углублять и подпитывать ваши знания, посещайте региональные мероприятия и местные сообщества пользователей, перенимая опыт тех, кто уже не первый день работает с этим популярным языком.

А сейчас просто наслаждайтесь чтением и воодушевляйтесь тем, что вскоре перед вами распахнется целая вселенная данных.

Благодарности

Работа над *«Изучаем SQL за месяц, занимаясь один час в день»* стала, пожалуй, самым неожиданным и увлекательным приключением в моей жизни. Никогда прежде я не думал, что займусь написанием книги. Но стоило издательству Manning Publications предоставить мне такую возможность, как я ощутил невероятный заряд вдохновения — вокруг оказалось столько людей, готовых помочь и поддержать! На обложке указано мое имя, но к тому, чтобы эта книга вышла в свет, приложили руку многие.

Прежде всего, я бесконечно признателен моим редакторам-консультантам, Карен Миллер (Karen Miller) и Бекки Уитни (Becky Whitney), за их терпение и помощь в создании книги, а также Майку Шепарду (Mike Shepard) за его выдающиеся технические замечания и рекомендации. Майк — архитектор программных решений в компании Jack Henry and Associates, выпускник Университета штата Миссури, где он изучал математику и информатику. За свои 27 лет в ИТ-сфере он успел поработать разработчиком, администратором баз данных, системным администратором и архитектором ПО. Его специализация — оптимизация процессов с использованием PowerShell и SQL Server. Неоценимую помощь в создании, организации и продвижении издания оказали такие замечательные специалисты, как Майк Стивенс (Mike Stephens), Элеонор Гарднер (Eleonor Gardner), Малена Селич (Malena Selic), Ария Дучич (Aria Ducic), Пол Спрэтли (Paul Spratley), Матко Хрватин (Matko Hrvatin), Адриано Сабо (Adriano Sabo), Ана Ромак (Ana Romac), Сьюзен Ханиуэлл (Susan Honeywell) и Стьепан Юрекович (Stjepan Jurekovic). Отдельная благодарность редактору проекта Дидре Бланчфилд-Хайэм (Deirdre Blanchfield-Hiam), литературному редактору Киру Симпсону (Keir Simpson) и корректору Мелоди Долаб (Melody Dolab). Мне невероятно повезло работать с такой потрясающей командой профессионалов!

Отдельное спасибо моему другу Майку Уолшу (Mike Walsh) и всем моим коллегам по Straight Path Solutions за их поддержку и вдохновение во время работы над книгой, а также наставникам, чьи мудрые советы помогали мне на протяжении

многих лет, — среди них Мартин Грант (Martin Grant), Курт Джонсон (Curt Johnson), Крис Роуз (Chris Rose), Крис Хинсон (Chris Hinson), Энди Юн (Andy Yun), Джинджер Грант (Ginger Grant), Бак Вуди (Buck Woody), Грант Фритчи (Grant Fritchey) и Кевин Кляйн (Kevin Kline).

Благодарю всех рецензентов: Али Шакибу (Ali Shakiba), Андреса Сакко (Andres Sacco), Кристиана Антониоли (Cristian Antonioli), Дейва Коруна (Dave Corun), Эдера Андреса Авила Ниньо (Eder Andres Avila Niño), Фостера Хейнса (Foster Haines), Джампьеро Гранателлу (Giampiero Granatella), Гранта Колли (Grant Colley), Грега Граймса (Greg Grimes), Харлана Брюэра (Harlan Brewer), Хелен Мэри Баррамеда (Helen Mary Barrameda), Иябо Синдику (Iyabo Sindiku), Джейн Нусгорд Ларсен (Jane Noesgaard Larsen), Хосе Альберто Рейеса Кеведо (José Alberto Reyes Quevedo), Мализу Мидлбрукс (Malisa Middlebrooks), Мэри Энн Тигесен (Mary Anne Thygesen), Маттиаса Лайна (Matthias Lein), Майка Бэрана (Mike Baran), Оливера Кортена (Oliver Korten), Паоло Брунасти (Paolo Brunasti), Пола Лава (Paul Love), Питера Шотта (Peter Schott), Раджешкумара Мутайю (Rajeshkumar Muthaiah), Равичандрана Раджу (Ravichandran Raja), Рохини Уппулури (Rohini Uppuluri), Саймона Чеке (Simon Tschoeke), Слеймана Саламе (Sleiman Salameh), Стивена Джозефа Эрреру Корралеса (Steven Joseph Herrera Corrales) и Свету Нату (Sveta Natu). Ваши отзывы и советы помогли значительно улучшить издание.

И наконец, мне не хватит слов, чтобы выразить всю признательность моей прекрасной жене Эми. Она не только помогала с большей частью форматирования текста, но и была моим первым редактором, вычитывая каждую главу перед отправкой в издательство. Эми проявила безграничное терпение во время моей уединенной работы над книгой и посвятила бесконечные часы проверке моих неаккуратных черновиков. Без нее этот труд так и остался бы неоконченным. Спасибо тебе, Эми. Всегда люблю тебя.

Об этой книге

Хотя язык программирования SQL был создан еще в 1970-х годах, применение реляционных баз данных с тех пор выросло в геометрической прогрессии. Сегодня компании по всему миру ищут профессионалов, умеющих осуществлять содержательную интерпретацию и анализ данных — и речь идет не только о разработчиках ПО.

Поразительно, но владение всего одним языком может открыть карьерные возможности как для программистов, так и для специалистов иных направлений. Ведь, по существу, единственным предварительным условием для изучения SQL является знание английского языка. SQL был спроектирован так, чтобы быть максимально близким к естественному английскому, поэтому освоить его под силу практически каждому.

Примеры в этом руководстве основаны на реализации SQL в системе MySQL. Однако не стоит думать, что данный материал окажется для вас бесполезным, если вы работаете с SQL Server, PostgreSQL или другой реляционной базой данных. Большинство приведенных здесь SQL-запросов подходят и для других систем, при этом я приложил немало усилий, чтобы пометить существующие исключения для наиболее распространенных СУБД.

Для кого эта книга

В отличие от большинства книг о языках программирования, *«Изучаем SQL за месяц, занимаясь один час в день»* написана так, чтобы сделать процесс изучения простым и доступным для всех начинающих пользователей SQL, вне зависимости от их опыта в программировании. Основной упор делается на прикладные аспекты языка, что позволяет в кратчайшие сроки освоить необходимые навыки для работы с информацией в реляционных базах данных.

Если вы не специалист в сфере IT, но вам необходимо собирать сведения для заказчиков, руководителей или тех, кто принимает решения на основе данных, эта книга для вас. Мне доводилось сотрудничать с представителями разных профессий, которым требовалось иметь дело с данными, и я не встречал ни одного человека, который не смог бы научиться составлять рабочие SQL-запросы. Я часто говорю, что если вы знакомы с электронными таблицами, такими как Microsoft Excel или Google Sheets, то вам не составит труда разобраться с реляционными базами данных и языком SQL.

Но даже если вы программист или специалист в области информационных технологий и решили освоить SQL, книга окажется хорошим подспорьем — вы сможете быстро разобраться с основами и начать применять этот язык для ваших задач. Хотя она и рассчитана на новичков, вы не почувствуете, что к вам относятся снисходительно. Убежден, что полезные замечания и предупреждения, встречающиеся в каждой главе, помогут вам создавать SQL-запросы более грамотно и эффективно по сравнению с большинством пользователей этого языка.

Структура книги

Учебное руководство *«Изучаем SQL за месяц, занимаясь один час в день»* состоит из 24 глав, каждая из которых опирается на ключевые слова и понятийный аппарат предыдущих. Мы начинаем с элементарных способов извлечения данных, чтобы затем перейти к отбору конкретных данных, различным способам соединения нескольких наборов данных, модификации данных, вплоть до создания объектов реляционной базы, предназначенных для хранения как самих данных, так и пользовательских SQL-сценариев.

Глава 1 знакомит читателя с языком SQL, объясняя, почему он так важен для доступа к данным в реляционных базах. Здесь же вы настроите вашу первую базу данных в среде MySQL.

Глава 2 без лишних преамбул переходит к работе с SQL: вы напишете первый запрос и увидите, насколько его структура уподоблена естественному языку.

В главе 3 мы приступим к составлению запросов, начиная с самых базовых (и популярных) способов.

Глава 4 показывает, как сортировать или ограничивать результаты запросов, а также знакомит с добавлением полезных комментариев в SQL-код.

Главы с 5-й по 7-ю посвящены углубленному изучению самых распространенных механизмов фильтрации данных при выполнении запросов. В них рассматриваются полезные ключевые слова, а также разбираются расхожие ошибочные представления относительно пустых NULL-значений.

В главах с 8-й по 11-ю описывается, как объединять наборы данных, связывая их различными способами через отношения, заданные в структуре базы данных.

Глава 12 содержит последовательное руководство по группировке данных для нахождения базовых арифметических значений, таких как максимум, среднее арифметическое и суммарное число значений в наборе данных.

Глава 13 знакомит с переменными — неотъемлемым элементом практически всех языков программирования, которые позволяют сохранять значения для последующего применения в рамках исполняемого сценария.

В главах 14 и 15 рассматриваются функции — специализированные языковые конструкции, предназначенные для выполнения типовых операций, в число которых входит получение значений даты и времени, а также изменение способа отображения данных.

Главы 16 и 17 посвящены операциям с данными. В них вы научитесь добавлять, редактировать и удалять значения в базе данных.

В главах 18 и 19 объясняется, как хранить данные в создаваемых вами таблицах, а также как применять ограничения и индексы для повышения целостности данных.

Глава 20 показывает, как многократно использовать SQL-код, сохраняя его в таких объектах базы данных, как хранимые процедуры и представления.

Глава 21 описывает механизмы реализации условной логики внутри запросов на основе различных критериев.

Глава 22 посвящена курсорам — инструменту для пошаговой обработки данных. Вы узнаете, как они работают, и познакомитесь с более эффективными альтернативами.

Глава 23 дает возможность применить все, чему вы научились, анализируя сторонние SQL-сценарии и изыскивая способы их доработки.

Глава 24 содержит заключительные рекомендации, определяющие перспективные направления для последующего самостоятельного изучения и совершенствования навыков работы с SQL.

О программном коде

Книга содержит множество примеров исходного кода как в нумерованных листингах, так и в тексте. В обоих случаях исходный код форматируется `моноширинным шрифтом`, в отличие от обычного текста. Иногда для кода также применяется **`жирный шрифт`**, чтобы выделить фрагменты, изменившиеся по сравнению

с предыдущими шагами, — например, при добавлении новой функциональности в существующую строку кода.

Во многих случаях оригинальная версия исходного кода переформатируется; добавляются разрывы строк и измененные отступы, чтобы код помещался на странице. Иногда даже этого оказывается недостаточно и в листинги включаются маркеры продолжения строк (➥). Также из исходного кода часто удаляются комментарии, если код описывается в тексте.

Исполняемые фрагменты кода можно загрузить из версии liveBook (электронной) по адресу https://livebook.manning.com/book/learn-sql-in-a-month-of-lunches. Полный код примеров книги доступен для загрузки на сайте издательства Manning по адресу https://www.manning.com/books/learn-sql-in-a-month-of-lunches, а также из репозитория GitHub по ссылке https://mng.bz/PNl8.

Форум liveBook

Приобретя печатную версию книги *«Изучаем SQL за месяц, занимаясь один час в день»*, вы также получаете бесплатный доступ к платформе для онлайн-чтения liveBook издательства Manning (на английском языке). Эксклюзивные возможности liveBook позволяют оставлять комментарии как к книге в целом, так и к отдельным ее разделам или абзацам. Можно легко делать заметки для себя, задавать технические вопросы и отвечать на них, а также получать помощь от авторов и других пользователей. Чтобы получить доступ к форуму, посетите страницу https://livebook.manning.com/book/learn-sql-in-a-month-of-lunches/discussion. Подробнее о форумах Manning и правилах поведения на них можно узнать по ссылке https://livebook.manning.com/discussion.

В рамках своих обязательств перед читателями издательство Manning предоставляет ресурс для содержательного общения читателей и авторов. Эти обязательства не подразумевают конкретную степень участия автора, которое остается добровольным (и неоплачиваемым). Задавайте автору хорошие вопросы, чтобы ему было интересно участвовать в диалоге! Форум и архивы обсуждений доступны на сайте Manning, пока книга продолжает издаваться.

Об авторе



Джефф Януччи (Jeff Iannucci) — старший консультант компании Straight Path Solutions, специализирующийся на администрировании реляционных баз данных и программировании на языке SQL. Более 20 лет он применяет SQL для решения головоломных задач, связанных с проектированием и оптимизацией баз данных. Джефф активно делится своим профессиональным опытом в качестве докладчика на отраслевых конференциях и мероприятиях пользовательских групп, публикуется на различных специализированных интернет-ресурсах и выступает в роли автора образовательного контента на платформе Pluralsight.

1 Прежде чем начать

Практически каждое наше действие оставляет цифровой след. Каждая покупка, каждая пройденная миля и каждый переход по ссылке в интернете заносятся в колоссальный, все разрастающийся массив данных, превратившийся для многих компаний в их самый ценный актив. Эти данные, как правило, хранятся в реляционной базе данных (БД), обеспечивающей их безопасность, масштабируемость и доступность для непрерывного чтения и правки бесчисленным количеством пользователей.

Однако как именно пользователи взаимодействуют с данными в реляционной БД? И, что еще важнее, каким образом осуществляется чтение критически важной для вашей компании информации? Ответ на этот вопрос, а также предмет рассмотрения в книге, — *язык структурированных запросов* (Structured Query Language), широко известный под аббревиатурой *SQL*.

1.1. Актуальность SQL

Возможно, вы задаетесь вопросом: а стоит ли SQL того, чтобы потратить на его изучение целых 24 часа? Можете не сомневаться — владение этим языком по праву считается одним из ценнейших навыков для любого специалиста, работающего с информацией. Огромное количество данных в современном мире хранится в системах управления базами данных (СУБД) от разных производителей, таких как Oracle, Microsoft и IBM, и почти все они применяют SQL, причем на протяжении достаточно долгого времени.

В то время как многие языки программирования живут всего несколько лет, SQL остается стандартом для работы с реляционными базами данных на протяжении

уже нескольких десятилетий — и в ближайшем будущем его позиции ничто не угрожает. Таким образом, мастерство, которое вы приобретете и отточите, изучая это руководство и выполняя рекомендуемые упражнения, — это долгосрочная инвестиция в ваше профессиональное будущее.

Возможно, одно из ключевых преимуществ SQL — исключительная простота его освоения, обусловленная тем обстоятельством, что он максимально приближен к естественному английскому языку. Если вы знакомы с английским, команды и синтаксис, применяемые в SQL, покажутся вам интуитивно понятными. К примеру, для выборки имен и фамилий всех покупателей из Канады достаточно составить такой SQL-запрос:

```
SELECT FirstName, LastName FROM Customers WHERE Country = 'Canada';
```

Видите, как все просто? Пусть вам пока и не ясна каждая часть этого выражения, уверяю вас, что по прочтении первых нескольких глав этой книги вы научитесь свободно составлять такие конструкции.

1.2. Подходит ли вам эта книга?

Руководств, учебных пособий, видеокурсов и сайтов, предлагающих обучить вас SQL, существует великое множество. Однако большинство из них рассчитаны на аудиторию с опытом разработки программ. Зачастую они начинаются с изложения истории языка, затем переходят к обсуждению его широкого понятийного аппарата, после чего следуют главы, сгруппированные по принципу демонстрации работы различных команд. Возможно, это и неплохой подход, но он не особо учитывает интересы нетехнических специалистов, таких как я сам, которым необходимо знать SQL.

Хотя я пользуюсь SQL уже свыше 20 лет, начинал я отнюдь не программистом. Впервые я столкнулся с базами данных на должности администратора данных, где я отвечал за импорт данных из различных источников в реляционную БД. Мне нужно было считывать поступившие данные, чтобы проверять, все ли прошло успешно, — и единственным инструментом для этого был SQL, который мне и пришлось освоить.

И хотя тогда я почти ничего не смыслил в программировании, я довольно быстро разобрался с SQL. Если вы умеете писать по-английски, будьте уверены — у вас все получится. Вы убедитесь в этом сами, читая книгу: большинство команд и операторов SQL интуитивно понятны и звучат точь-в-точь как их английские эквиваленты.

1.2.1. Спектр применения SQL

Разумеется, данные нужны не только ИТ-отделу. Взять, к примеру, бизнес-аналитиков: при помощи SQL они могут оперативно извлекать и анализировать

информацию об операционных тенденциях для принятия обоснованных бизнес-решений. Маркетологи могут воспользоваться SQL для выявления полезных инсайтов о недавних рекламных кампаниях и поиска новых точек роста для бизнеса, а финансисты — для получения критически важной бизнес-отчетности, необходимой для обеспечения соответствия нормативным требованиям.

Все эти данные являются основой существования и развития любой современной компании, а успех напрямую зависит от того, способны ли сотрудники самых разных отделов работать с реляционными БД и на основе извлекаемых данных принимать судьбоносные для бизнеса решения. Цель этой книги — помочь вам, как одному из таких специалистов, овладеть SQL и развить соответствующие навыки. Если пока ваш главный инструмент — электронные таблицы, это уже неплохо: вы идеально подготовлены к старту.

Если же вы программист, администратор БД или специалист по анализу данных, эта книга предназначена и для вас — просто метод подачи материала здесь несколько иной. В отличие от большинства учебников, которые с первых страниц обрушивают на вас поток терминологии, вы сразу приступите к решению реальных задач на SQL, постигая концепции и значения терминов постепенно, в ходе работы.

В то же время моя цель — не просто научить вас набору SQL-команд. Продвигаясь шаг за шагом, я покажу вам, как применять элементы языка SQL для решения рабочих задач, вне зависимости от вашего уровня подготовки в области программирования.

1.2.2. Диалекты SQL

Несмотря на то что изучать SQL мы будем на примере MySQL, практически все рассматриваемые нами концепции и приемы работы с языком универсальны и подойдут для любой реляционной базы данных. Таким образом, обретенные вами знания будут применимы к любой из следующих СУБД:

- IBM DB2;
- MariaDB;
- Microsoft SQL Server;
- MySQL;
- Oracle;
- PostgreSQL.

Сформировав прочный фундамент знаний в области применения SQL, вы сможете работать с данными в любой из перечисленных систем. Правда, стоит иметь в виду: в каждой из них встречаются свои особенности. Я буду отмечать их по ходу книги, чтобы вы чувствовали себя уверенно с любым диалектом SQL.

1.2.3. ИИ и SQL

С появлением генеративного искусственного интеллекта (ИИ) резонно задаться вопросом: а стоит ли вообще изучать SQL, когда можно просто попросить ИИ-ассистента, скажем, ChatGPT, написать любой необходимый SQL-код? Безусловно, ИИ выглядит соблазнительной альтернативой изучению языка. Но подвох вот в чем — без прочного знания SQL вы даже не сможете проверить, будет ли код, предоставленный таким инструментом, давать надлежащие результаты. К тому же, чтобы нейросеть сгенерировала конкретный SQL-запрос, вам нужно предоставить сведения о вашей базе данных, что может быть запрещено корпоративными правилами.

Впрочем, у генеративного ИИ есть и своя ниша — он может стать отличным помощником для тех, кто уже уверенно владеет SQL. Когда вы достигнете уровня, на котором будете точно понимать, как работает ваш код, эти инструменты можно будет задействовать для быстрого поиска узких мест в производительности или для анализа логики выполнения запросов. Однако для интерпретации предоставляемых рекомендаций и объяснений все равно необходимы знания SQL, сформировать которые и призвано это руководство.

1.3. Как построена книга

Концепция работы с книгой проста: по одной главе в день. Разумеется, читать можно не только в обеденный перерыв, хоть это и идеальное время: на главу уйдет минут 40, и у вас еще останется минут 20 на то, чтобы закрепить материал на практике, пока вы допиваете кофе.

1.3.1. Основные главы

Главы 1 и 2 помогут вам быстро войти в курс дела. Они не только объяснят, что такое таблица и как правильно формулировать запросы, но и познакомят с инструментарием, который пригодится вам в дальнейшем. В некотором смысле это самые важные главы во всей книге.

Главы с 3-й по 22-ю содержат основной учебный материал, на освоение которого уйдет около месяца, даже если месяц такой короткий, как февраль. Не на каждую главу потребуется целый час, но очень важно идти по порядку: каждая следующая глава опирается на сведения и команды из предыдущих. Можно, конечно, читать и по нескольку глав в день, но я настоятельно рекомендую не спешить: осваивайте по одной главе в день и не жалейте времени на практику. Умение сосредоточиться на небольшом наборе понятий и примеров обеспечивает оптимальный режим для быстрого закрепления знаний. Как говаривал великий баскетбольный тренер Джон Вуден: «Поспешай не торопясь».

1.3.2. Практические занятия

Почти в каждой главе вас ждет небольшой практикум — специальное задание, позволяющее незамедлительно опробовать новые команды и концепции в деле. Относитесь к этим упражнениям не как к контрольной проверке ваших знаний, а как к возможности применить и закрепить ваши новые навыки работы с SQL. Разумеется, ответы на практические задачи приведены в конце каждой главы, но поверьте: именно их самостоятельное прохождение — ключ к эффективному усвоению пройденного материала.

1.3.3. Дальнейшее изучение

Книга призвана стать отправной точкой для тех, кто делает первые шаги в SQL и лишь приоткрывает дверь в мир возможностей работы с реляционными базами данных. Поэтому в конце некоторых глав вы найдете рекомендации по дальнейшему изучению рассмотренных аспектов языка. Если у вас есть время и желание, ознакомьтесь с этими ресурсами, чтобы обогатить ваш растущий набор профессиональных навыков в области SQL.

1.4. Подготовка рабочего места

Не теряя времени, приступим к настройке всего, что необходимо для практики. Вся процедура займет буквально несколько минут и не потребует мощного компьютера. Нам нужно установить две бесплатные программы, а затем выполнить готовый SQL-сценарий, чтобы сгенерировать данные для практикума.

1.4.1. Установка MySQL и MySQL Workbench

Первый шаг — загрузка MySQL и ее установка на выбранный вами компьютер. MySQL не только совершенно бесплатна, но и входит в число самых популярных реляционных СУБД в мире.

Мы также установим MySQL Workbench — именно в этой среде мы и будем выполнять все запросы из книги. Она потребляет минимальный объем ресурсов, так что можете смело ставить MySQL Workbench даже на ноутбук.

Инструкции по скачиванию и установке этих двух приложений доступны в моем GitHub-репозитории по адресу https://mng.bz/PNl8. Поскольку программный пакет MySQL часто обновляется, номера версий, которые вы увидите, могут быть новее, чем указанные в документации. Не беспокойтесь об этом — все примеры из книги будут работать и на свежих сборках.

1.4.2. Выполнение практических работ

На протяжении всей книги мы будем работать с одним и тем же набором данных, размещенным в базе данных `sqlnovel`. Это заказы вымышленного издательства, которое специализируется на книгах про SQL. Мы рассмотрим эти данные немного позднее, а пока создадим базу данных и заполним ее тестовым содержимым, запустив подготовленный SQL-сценарий.

Инструкции по настройке базы данных `sqlnovel` также размещены по адресу https://mng.bz/PNl8, и они еще проще, чем процедура установки MySQL и MySQL Workbench. Сейчас вы просто запустите готовый сценарий, а ближе к концу книги мы вернемся к нему и разберем, как он устроен. К тому моменту вы уже сможете создавать собственные наборы данных!

1.5. Полезные онлайн-ресурсы

В книге вас ждет множество примеров и заданий для самостоятельной работы. Я призываю вас набирать все сценарии самостоятельно и при желании оформлять SQL-код в стиле, отличном от представленного в этом руководстве. Так вы лучше почувствуете язык. При наборе SQL-инструкций вы можете столкнуться с непонятными вам ошибками. В связи с этим в онлайн-репозитории размещены все SQL-сценарии, фигурирующие в издании. Заглядывайте туда лишь для того, чтобы свериться или найти причину ошибки. Поверьте, когда вы пишете код сами, а не копируете его, вы учитесь в разы быстрее.

1.6. SQL в действии — с первого же дня

Подобно другим изданиям серии «За 24 часа», первостепенная задача книги — прямо в процессе освоения материала научить вас эффективно применять его на практике. Так, каждая глава посвящена конкретному компоненту языка SQL и дает его краткий обзор, при этом основное внимание уделяется сценариям из реальной жизни. Дополнительно в конце каждой главы предусмотрено закрепление обретенных навыков: выполнение упражнений в форме практической работы.

Как я отмечал ранее, если ваша цель — углубленное изучение теории и истории реляционных баз данных, существует масса других изданий, способных стать хорошим помощником на этом пути. Несмотря на то что во многих частях руководства обсуждаются технические детали и тонкости, каждая глава подчинена одной цели — обеспечить вашу оперативную результативность при решении прикладных задач.

Итак, достаточно о книге. Приступаем к работе с SQL!

2 Ваш первый SQL-запрос

Первую главу мы закончили словами об оперативной результативности — и сейчас безотлагательно перейдем к делу! Начнем с рассмотрения того, как данные хранятся в реляционной базе, и разберемся с их структурой. Это поможет вам увереннее оперировать терминами, которые встретятся в книге. Не волнуйтесь: терминов будет совсем немного, да и слова эти хорошо вам известны и применяются в обычной жизни. Просто теперь мы уточним их в контексте хранения данных в реляционных БД.

Кроме того, вы построите ваш первый запрос. На всякий случай уточним: под *запросом* (query) подразумевается выполнение SQL-команды с целью извлечения данных из базы. Чем дальше вы углубитесь в книгу, тем больше запросов напишете — и тем увереннее будете себя чувствовать в SQL. Если вы еще не установили MySQL и MySQL Workbench (см. главу 1) и не запустили сценарий `Create_SQLNovel_database.sql` для создания учебной БД, — самое время это сделать, чтобы можно было сразу приступить к работе. Однако, прежде чем начать выполнять запросы, давайте сначала взглянем на сами данные.

2.1. Кто работал с Excel — уже знаком с таблицами БД

Хотя для изучения SQL это и не обязательно, опыт работы с Excel или другими табличными редакторами очень пригодится. Быть может, вы об этом и не задумывались, но структура электронных таблиц весьма схожа с устройством ключевых объектов БД. В этой главе мы познакомимся с рядом терминов, которые помогут разобраться, как хранятся данные в реляционной базе и как с ними работает система управления базами данных (СУБД).

Мы не просто собираем данные и загружаем их в реляционную БД как попало; мы организуем и сохраняем их в объектах, соответствующих характеру самих данных. Эти объекты называются *таблицами*. Как правило, мы группируем данные в таблицах, соотнося их с элементами предметной области, такими как заказы, клиенты или платежи. Таблицы являются составными компонентами любой реляционной базы данных и по своей структуре во многом похожи на электронные таблицы, поэтому сравнение с таблицами Excel поможет понять термины, применяемые в этой главе и далее в книге.

Если вы не знакомы с основными терминами, служащими для описания электронной таблицы, взгляните на типичную электронную таблицу на рис. 2.1, содержащую сведения о нескольких книгах про SQL (разумеется, вымышленных)[1]. Считайте такого рода информацию массивом данных, или, как часто говорят, *набором данных* (dataset). Все довольно просто, не так ли? Этот набор представлен в электронной таблице, однако если бы он хранился в таблице базы данных, его структура осталась бы практически идентичной.

	A	B	C	D	E
1	Название	Цена	Аванс	Гонорар	Дата публикации
2	Pride and Predicates	$ 9.95	$ 5,000.00	15%	4/30/2015
3	The Join Luck Club	$ 9.95	$ 6,000.00	12%	2/6/2016
4	The Catcher in the Try	$ 8.95	$ 5,000.00	10%	4/3/2017
5	Anne of Fact Table	$ 12.95	$ 10,000.00	15%	1/12/2018
6	The DateTime Machine	$ 7.95	$ 5,500.00	15%	2/4/2019
7	The Great GroupBy	$ 10.95	$ -	20%	12/23/2019
8	The Call of the While	$ 8.95	$ 2,500.00	15%	3/14/2020
9	The Sum Also Rises	$ 7.95	$ 5,000.00	12%	11/12/2021

Books +

Рис. 2.1. Пример электронной таблицы, содержащей пять столбцов (A–E) с информацией о книгах. Ее структура очень похожа на таблицу базы данных

Я обещал не сыпать сложными терминами, но три основных понятия, связанных с таблицами, следует усвоить обязательно, прежде чем вы начнете применять

[1] В заглавиях вымышленных «книг про SQL» обыгрываются названия следующих классических литературных произведений: «Гордость и предубеждение» Джейн Остин (*Pride and Prejudice*), «Клуб радости и удачи» Эми Тан (*The Joy Luck Club*), «Над пропастью во ржи» Джерома Сэлинджера (*The Catcher in the Rye*), «Энн из Зеленых Мезонинов» Люси Мод Монтгомери (*Anne of Green Gables*), «Машина времени» Герберта Джорджа Уэллса (*The Time Machine*), «Великий Гэтсби» Фрэнсиса Скотта Фицджеральда (*The Great Gatsby*), «Зов предков» Джека Лондона (*The Call of the Wild*), «Фиеста» Эрнеста Хемингуэя (*The Sun Also Rises*). — *Примеч. пер.*

SQL для чтения и обработки данных. В начале главы я уже говорил, что вы наверняка уже слышали эти слова раньше:

- столбец;
- строка;
- значение.

Следуя самому простому определению, таблица — это структура из одного или нескольких *столбцов* (column) данных. Столбцы располагаются вертикально, подобно архитектурным колоннам. На рис. 2.2 мы видим столбцы «Название», «Цена», «Аванс», «Гонорар» и «Дата публикации», причем столбец «Название» выделен. Иногда столбец называют *полем* (field), однако в формальной спецификации языка SQL такой термин не применяется.

Следующий ключевой термин — это *строка* (row). Строками являются горизонтальные записи данных в таблице. Каждая такая запись представляет собой отдельный элемент из множества тех объектов, которым посвящена приводимая таблица, — в нашем примере одну конкретную книгу. Посмотрите на рис. 2.3: все строки имеют одинаковую структуру и повторяют один и тот же порядок столбцов. Это необходимо, поскольку в каждой строке должны быть представлены все столбцы таблицы.

	A	B	C	D	E
1	Название	Цена	Аванс	Гонорар	Дата публикации
2	Pride and Predicates	$ 9.95	$ 5,000.00	15%	4/30/2015
3	The Join Luck Club	$ 9.95	$ 6,000.00	12%	2/6/2016
4	The Catcher in the Try	$ 8.95	$ 5,000.00	10%	4/3/2017
5	Anne of Fact Table	$ 12.95	$ 10,000.00	15%	1/12/2018
6	The DateTime Machine	$ 7.95	$ 5,500.00	15%	2/4/2019
7	The Great GroupBy	$ 10.95	$ -	20%	12/23/2019
8	The Call of the While	$ 8.95	$ 2,500.00	15%	3/14/2020
9	The Sum Also Rises	$ 7.95	$ 5,000.00	12%	11/12/2021

Рис. 2.2. Электронная таблица с названиями книг. Выделенный столбец «Название» подчеркивает вертикальное расположение столбцов

Эти строки также пронумерованы в левой боковой панели, впрочем, в таблицах не всегда присутствует явная нумерация. Следует отметить, что слова *«строка»* (row) и *«запись»* (record) часто означают одно и то же — в некоторых программах строки называются записями. Тем не менее в большинстве реляционных СУБД корректным термином для обозначения горизонтальных записей в таблицах является именно «строка».

	A	B	C	D	E
1	Название	Цена	Аванс	Гонорар	Дата публикации
2	Pride and Predicates	$ 9.95	$ 5,000.00	15%	4/30/2015
3	The Join Luck Club	$ 9.95	$ 6,000.00	12%	2/6/2016
4	The Catcher in the Try	$ 8.95	$ 5,000.00	10%	4/3/2017
5	Anne of Fact Table	$ 12.95	$ 10,000.00	15%	1/12/2018
6	The DateTime Machine	$ 7.95	$ 5,500.00	15%	2/4/2019
7	The Great GroupBy	$ 10.95	$ -	20%	12/23/2019
8	The Call of the While	$ 8.95	$ 2,500.00	15%	3/14/2020
9	The Sum Also Rises	$ 7.95	$ 5,000.00	12%	11/12/2021

Рис. 2.3. Электронная таблица с названиями книг. Выделенная первая строка подчеркивает горизонтальное расположение строк

И наконец, последний (по крайней мере, на текущий момент) термин — это *значение* (value), которое представляет собой отдельный фрагмент данных на пересечении строки и столбца таблицы. Каждая строка содержит по одному значению для каждого столбца. Так, на рис. 2.4 значение столбца «Название» в последней строке нашего набора данных — «The Sum Also Rises», а значение столбца «Цена» в этой строке — «$7.95». Стоит отметить, что хотя в каждой строке должны быть представлены все столбцы, сами значения могут оставаться пустыми. К примеру, в строке с названием «The Great GroupBy» значение «Аванс» не заполнено.

	A	B	C	D	E
1	Название	Цена	Аванс	Гонорар	Дата публикации
2	Pride and Predicates	$ 9.95	$ 5,000.00	15%	4/30/2015
3	The Join Luck Club	$ 9.95	$ 6,000.00	12%	2/6/2016
4	The Catcher in the Try	$ 8.95	$ 5,000.00	10%	4/3/2017
5	Anne of Fact Table	$ 12.95	$ 10,000.00	15%	1/12/2018
6	The DateTime Machine	$ 7.95	$ 5,500.00	15%	2/4/2019
7	The Great GroupBy	$ 10.95	$ -	20%	12/23/2019
8	The Call of the While	$ 8.95	$ 2,500.00	15%	3/14/2020
9	The Sum Also Rises	$ 7.95	$ 5,000.00	12%	11/12/2021

Рис. 2.4. Выделено значение столбца «Цена» в строке с названием «The Sum Also Rises» — одно из множества других значений в таблице

Ну что ж, с табличной терминологией пока закончим. Теперь самое интересное — давайте узнаем, как с помощью всего этого делать запросы к таблицам!

2.2. Изучение SQL похоже на урок английского

Многие спорят, как правильно произносить *SQL*: по буквам «эс-кью-эл» или как слово sequel («сиквел»). Учитывая, что первый прототип языка назывался Structured English Query Language («язык структурированных запросов на английском языке»), или сокращенно SEQUEL, становится понятно, отчего прижился второй вариант произношения. И хотя это дело прошлое, из-за существования торговой марки SEQUEL авторам пришлось убрать из названия слово *English* и сократить аббревиатуру до SQL.

Здесь кроется еще одна причина популярности SQL: в отличие от большинства языков программирования, он максимально приближен к естественному английскому. Дело в том, что SQL — *декларативный* язык, в котором программист формулирует саму задачу, а не алгоритм ее решения. Вы просто запрашиваете необходимые вам данные. А как именно их получить — уже забота реляционной СУБД, с которой вы работаете.

VegetableID	Name
1	Артишок
2	Бок-чой
3	Брокколи
4	Свекла
5	Спаржа

Рис. 2.5. Таблица vegetables, состоящая из двух столбцов и пяти строк. Сейчас мы узнаем, как создать запрос, выводящий все названия овощей

Следуя логике декларативного языка, можно пойти еще дальше: представьте, что вы описываете, что нужно сделать с данными, пользуясь... обычными словами! А почему бы и нет? Как скоро вы убедитесь сами, многие SQL-запросы напоминают фразы из речевого обихода. Рассмотрим конкретный пример. Допустим, имеется таблица `vegetables` (овощи), как показано на рис. 2.5, и вы хотите выбрать из нее названия всех овощей. Пользуясь простым языком, вы могли бы произнести нечто вроде: «Мне нужны все названия овощей». SQL не настолько интуитивен, чтобы понимать разговорную речь, но достаточно к тому близок. Чтобы выполнить ваш гипотетический запрос, необходимо включить в инструкцию два основных ключевых слова SQL. Что такое *ключевое слово* (keyword)? Это специальное слово языка SQL, при помощи которого можно сделать... да практически все что угодно. Первое и главное из них — `SELECT`. Оно станет вашим лучшим другом в мире баз данных! Поверьте — вы и `SELECT` будете много работать вместе. И это неудивительно. Ведь именно ключевое слово `SELECT` определяет, что именно вы хотите увидеть и в какой форме: “I would like to SELECT all the names of the vegetables” («Хочу ВЫБРАТЬ все названия овощей»).

Итак, мы на пути к формированию настоящего SQL-запроса, однако необходимо добавить еще кое-что: ключевое слово `FROM`. Оно указывает, откуда именно

брать данные. В нашем случае — из таблицы `vegetables`: “I would like to SELECT all the names FROM the vegetables table” («Хочу ВЫБРАТЬ все названия ИЗ таблицы овощей»).

Уже лучше. Правда, когда мы указываем источник данных при помощи `FROM`, то не уточняем явным образом, что это таблица. Пусть таблицы — лишь один из нескольких видов наборов данных, подлежащих запросу, СУБД в состоянии сама определить нужный набор по его имени: “I would like to SELECT all the names FROM vegetables” («Хочу ВЫБРАТЬ все названия ИЗ овощей».

Мы все ближе и ближе. Теперь поразмыслим над фрагментом «SELECT all the names» (ВЫБРАТЬ все названия). Если мы хотим извлечь все названия, то задача упрощается, поскольку это поведение — по умолчанию для такого типа запроса. Раз здесь нет указания на то, что нам нужны какие-то конкретные названия, то не нужно явным образом указывать, что мы хотим их все: “I would like to SELECT names FROM vegetables” («Хочу ВЫБРАТЬ названия ИЗ овощей»).

Если вы внимательно изучили таблицу на рис. 2.5, то должны были заметить еще один нюанс. Оказывается, нужный нам столбец называется `Name` (Название), а не `Names` (Названия). Со временем вы заметите, что в базах данных столбцы почти всегда озаглавливаются в единственном числе, так как в одном столбце редко содержится больше одного значения для одной строки. Подкорректируем наш запрос: “I would like to SELECT Name FROM vegetables” («Хочу ВЫБРАТЬ Название ИЗ овощей»).

Мы почти у цели. Осталось убрать вводную часть “I would like to” («Хочу...») — ведь любой запрос начинается сразу с ключевого слова, в нашем случае со слова `SELECT`. Согласитесь, так и правда лаконичнее?

И завершающий штрих: в конце нужно поставить точку с запятой. Она работает как точка в предложении, сообщая реляционной СУБД о завершении текущего запроса и отделяя его от последующих команд:

```
SELECT Name FROM vegetables;
```

Вот, собственно, и все! Перед вами абсолютно корректный запрос названия всех овощей из БД. Далее мы перейдем от проектирования запроса на основе гипотетических данных к составлению и выполнению запроса к реальным данным.

2.3. Ваш первый SQL-запрос

Ваш первый SQL-запрос будет очень простым: вы лишь попросите вывести результат вашего первого запроса. Как мы договорились, начнем с построения предложения на английском языке, в котором сформулируем наше пожелание:

"I would like the outcome of my first query" («Хочу получить результат моего первого запроса»).

Все необходимые данные уже подготовлены в столбце `Outcome` («Результат») таблицы `MyFirstQuery` («Мой первый запрос»). Удобно, не так ли? Применив знания о синтаксисе SQL, что мы почерпнули в разделе 2.2, вы сможете без особого труда составить нужный запрос:

```
SELECT Outcome FROM MyFirstQuery;
```

Обратите внимание — запрос завершается точкой с запятой. Добавлю к сказанному выше: в SQL точка с запятой обозначает *конец SQL-команды* (statement terminator). Чтобы не углубляться в технические подробности, просто запомните, что точка с запятой говорит системе: «Мы закончили с предыдущей SQL-командой, и все, что следует после, является уже другой командой». Это помогает базе данных избежать путаницы (особенно при обработке более сложных инструкций, к которым мы перейдем позднее), поэтому мы будем ставить точки с запятой в конце каждого SQL-запроса.

Возможно, вы задаетесь вопросом: а есть ли разница между «командой» и «запросом»? Или же это синонимы? На самом деле, это не тождественные понятия, однако они взаимосвязаны. Считайте, что запрос — это особый вид инструкции, предназначенный для извлечения данных. По мере развития ваших навыков SQL вы узнаете и другие виды инструкций. А пока что запросы — это единственный вид SQL-команд, с которым мы будем работать.

> **НА ЗАМЕТКУ** В разных реляционных СУБД правила могут отличаться: где-то точка с запятой в конце запроса обязательна, а где-то нет. Однако лучше сразу выработать привычку завершать все SQL-инструкции точкой с запятой — даже в таких простых случаях, как ваш первый запрос. Это хороший тон в мире SQL!

Не будем пока углубляться в дебри синтаксических правил и вернемся к нашему запросу. Итак, он готов, теперь нужно открыть MySQL Workbench и выполнить его. Представьте, что выполнение — это как нажать кнопку «Отправить»: вы посылаете задание СУБД, она сама находит оптимальный способ его выполнить и отображает результат в специальном окне MySQL Workbench.

К практике!

Запустите MySQL Workbench и откройте подключение Month of Lunches, которое мы настроили ранее. Можно просто щелкнуть на нем мышью или вызвать правой кнопкой контекстное меню и выбрать Open Connection («Открыть подключение»). Должно появиться окно, приблизительно как на рис. 2.6, с активной вкладкой Query 1. Это верхняя часть панели запросов; слева в редакторе вы увидите число 1 — это номер первой строки любого запроса, который мы здесь введем.

Обратите внимание на интерфейс MySQL Workbench. Сверху вы видите вкладку с подписью Month of Lunches — она показывает, в каком контексте вы сейчас подключены. Этот контекст мы настраивали ранее для пользователя lunch. В упражнениях книги менять его не придется. Но если позже вы будете работать с MySQL самостоятельно, всегда следите, к какому подключению обращаются ваши запросы.

Следующий элемент, на котором следует заострить ваше внимание, размещен слева. По соседству со вкладкой Administration (Администрирование) находится более важная для нас вкладка с названием Schemas (Схемы). Нажмите на слово Schemas и в открывшемся списке найдите базу данных sqlnovel. Она создана сценарием, который мы выполнили в первой главе, и мы хотим установить ее в качестве базы данных по умолчанию для всех наших запросов. Щелкните правой кнопкой мыши на sqlnovel и выберите пункт Set as Default Schema (Установить как схему по умолчанию) во всплывающем меню. После этого ваш интерфейс MySQL Workbench должен выглядеть так, как продемонстрировано на рис. 2.7.

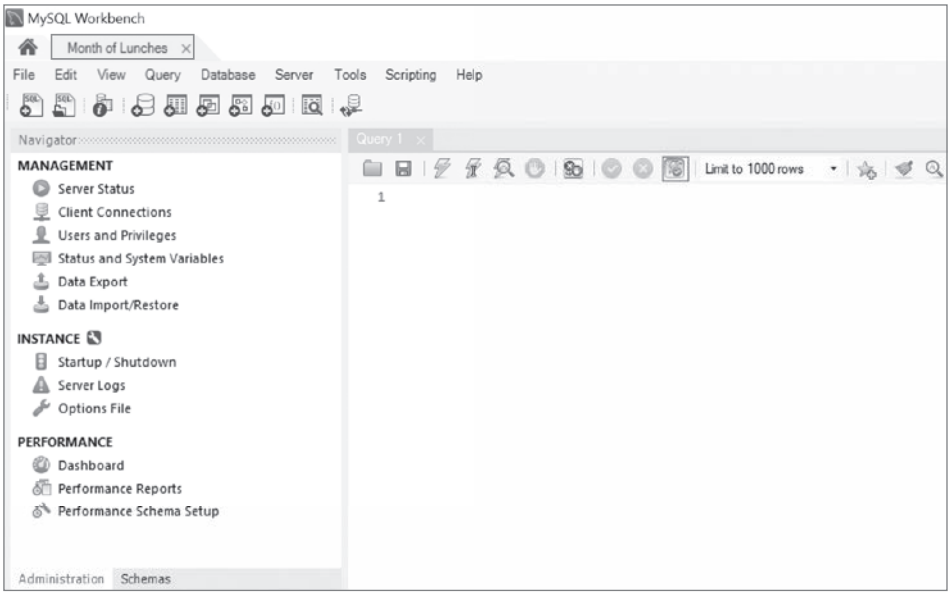


Рис. 2.6. Приложение MySQL Workbench открыто с активным подключением Month of Lunches, в панели навигации отображается служебная информация (Administration). Запросы, подобные только что написанному, мы вводим в панели запросов

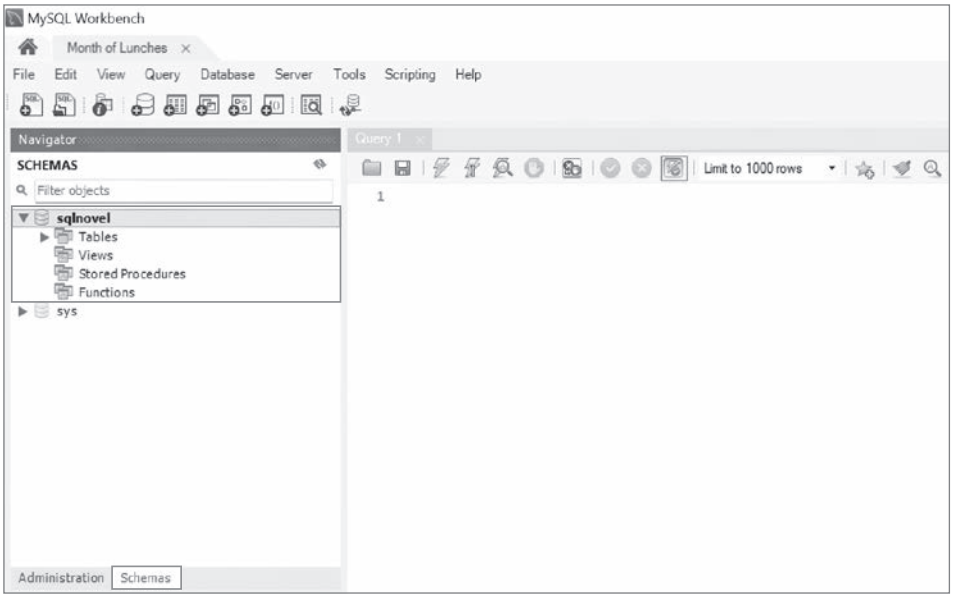


Рис. 2.7. Интерфейс с тем же активным подключением Month of Lunches. На этот раз в панели навигации открыта вкладка Schemas, где база данных sqlnovel отображается полужирным шрифтом, поскольку теперь она выбрана по умолчанию для всех запросов

По мере изложения материала я буду уделять время разбору возможностей MySQL Workbench. Пока же нам достаточно проверить подключение и базу данных, с которой мы будем работать. Итак, вернемся к запросу:

```
SELECT Outcome FROM MyFirstQuery;
```

К практике!

Переместите курсор в панель запросов и щелкните справа от цифры 1 с синей точкой. Введите ваш первый SQL-запрос — тот самый, что мы только что составили. Результат должен выглядеть, как показано на рис. 2.8.

Знаю, нетерпение ваше растет — осталось совсем чуть-чуть! Запрос можно выполнить несколькими способами. Во-первых, можно выбрать Execute (All or Selection) (Выполнить (все или выбранное)) или Execute Current Statement (Выполнить текущую инструкцию) в разделе Query (Запрос) главного меню программы. Для нашего случая разницы нет — ведь в панели запросов у нас всего одна команда.

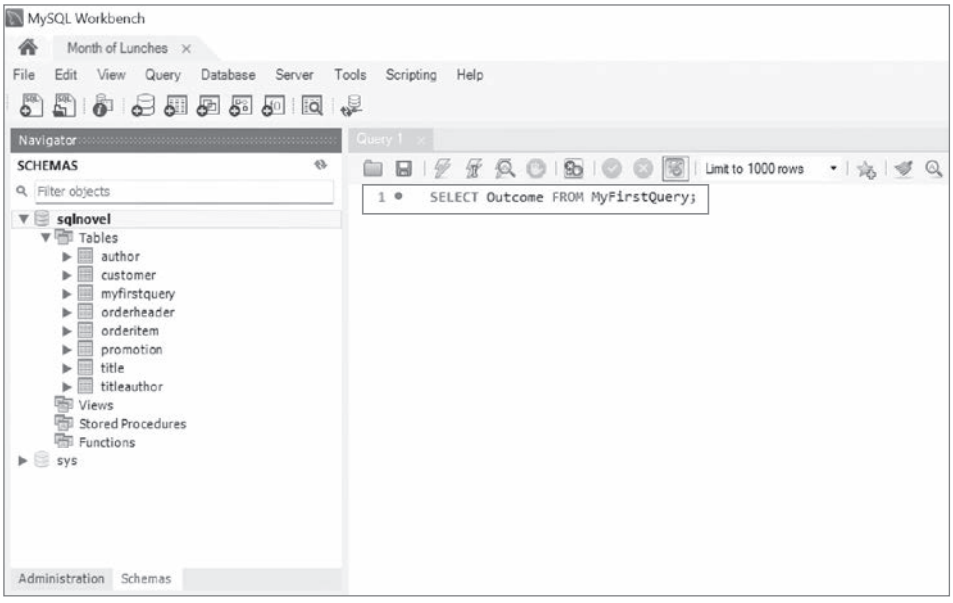


Рис. 2.8. Панель запросов с введенным и готовым к выполнению запросом

Наверняка вы заметили, что пунктам меню Query присвоены сочетания клавиш, которые можно задействовать для выполнения запроса. Так, Ctrl+Enter выполнит текущую команду под курсором, а Ctrl+Shift+Enter — все (или выделенное) содержимое панели. Опять же, поскольку наш запрос умещается в одну строку, горячие клавиши сработают одинаково.

И наконец, обратите внимание на кнопки над панелью запросов — некоторые со значками молний. Самая первая из них слева (выделенная на рис. 2.9) действует так же, как и Ctrl+Shift+Enter: выполняет выделенную часть сценария в панели запросов или весь сценарий при отсутствии выделения.

К практике!

Прежде чем выбрать способ выполнения, удостоверьтесь, что вы установили курсор в конец SQL-команды. А теперь вперед — сделайте это!

После запуска запроса ниже появятся две новые панели с результатами. Должно получиться, как на рис. 2.10 — проверьте!

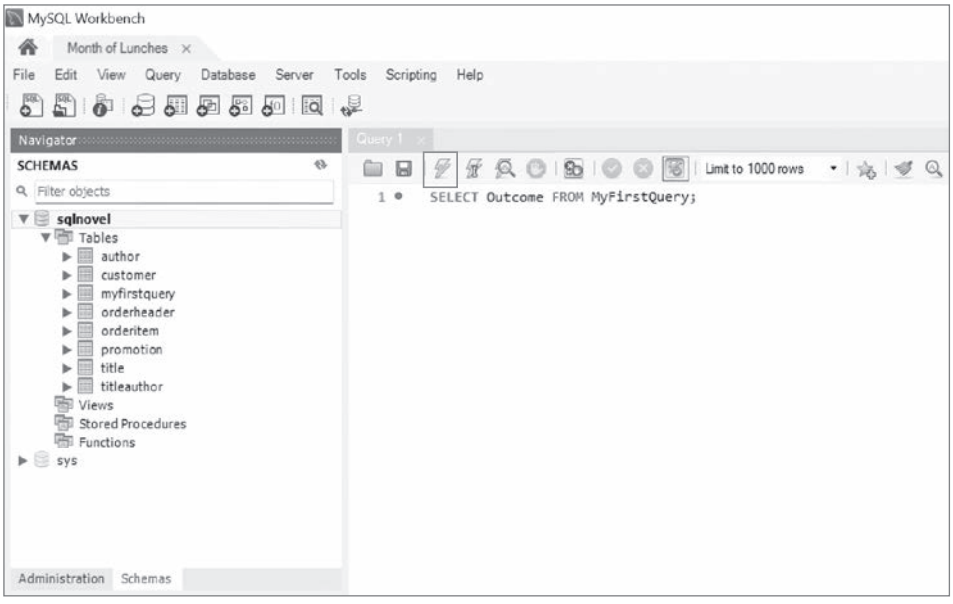


Рис. 2.9. В панели запросов подсвечена кнопка со значком простой молнии. Нажатие этой кнопки выполняет выделенный фрагмент в панели запросов, что соответствует нажатию Ctrl+Shift+Enter на клавиатуре

Сразу под панелью запроса вы видите панель результатов — и вот он, наш первый вывод: `"Hello, World!"`. Возможно, вы не в курсе, но у программистов есть негласное правило: начинать изучение любого языка с вывода этой фразы. SQL, может, и не похож на большинство языков программирования, но это не помешает нам отдать дань традиции.

А теперь взгляните прямо над `"Hello, World!"` — там есть слово Outcome (Вывод). Это название столбца, который мы запросили. Даже такой крохотный результат уже считается набором данных: один столбец, одна строка — но не таблица. Хотя мы и запрашивали таблицу с именем MyFirstQuery, результат запроса сам по себе является отдельным набором данных.

И еще одна вещь, на которую стоит обратить внимание при выполнении, — это панель вывода (Output panel), расположенная под панелью результатов (Result panel). В панели вывода отображается множество полезных сведений: время, когда был выполнен запрос, текст самого запроса, количество возвращенных строк и продолжительность выполнения всей операции. Наиболее важным является значок с зеленой галочкой в круге, указывающий на успешное выполнение запроса. Если бы что-то пошло не так, на его месте мы бы увидели круг с красным крестиком, сигнализирующим об ошибке. Надеюсь, что при работе над книгой мы с ним ни разу не столкнемся!

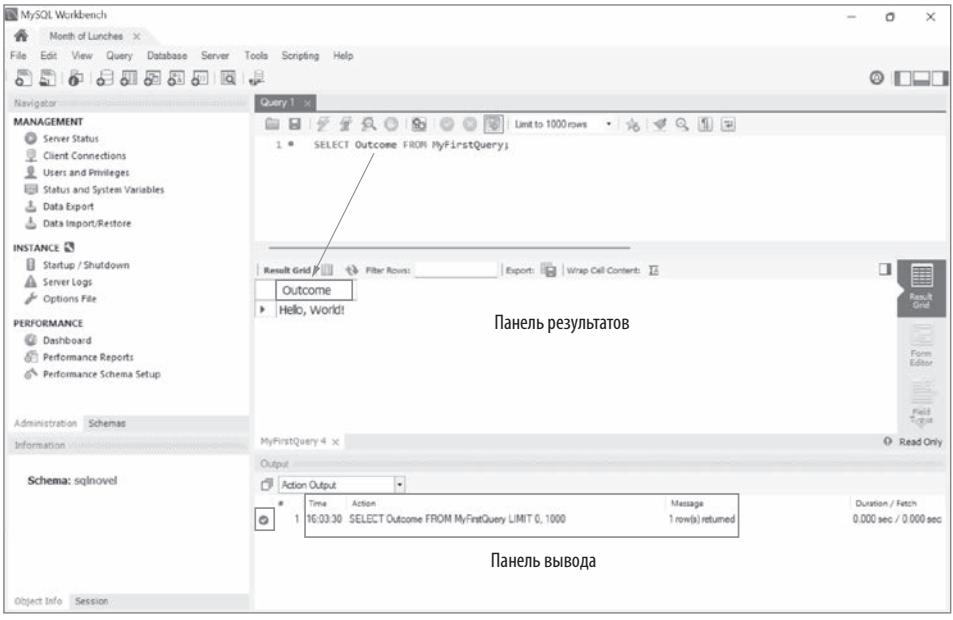


Рис. 2.10. Результат выполнения запроса. На панели результатов (Result panel) мы видим наш результат — «Hello, World!». На панели вывода (Output panel) круг с галочкой (которая отображается зеленым цветом) указывает на успешное выполнение запроса. Здесь же можно увидеть время выполнения, текст запроса, количество возвращенных строк и длительность выполнения операции

2.4. Ключевые слова и термины

В этой главе мы рассмотрели несколько основных терминов и ключевых слов, позволяющих вам приступить к освоению запросов к данным. Это тот самый фундамент, на котором строится весь SQL, поэтому давайте кратко повторим самое важное:

- *Набор данных* (Dataset) — логически сгруппированная информация, которая может храниться в базе данных, электронной таблице или иных системах, хранилищах или форматах. В контексте настоящего руководства мы будем применять SQL для работы с данными в реляционной СУБД.
- *Таблица* (Table) — логическая структура, состоящая из столбцов и содержащая набор данных. *Таблица* является основным способом организации наборов данных в реляционной СУБД.
- *Столбец* (Column) — вертикальный набор атрибутов, присутствующих в каждой строке таблицы.

- *Строка* (Row) — горизонтальная запись, объединяющая все данные об отдельном объекте в таблице.
- *Значение* (Value) — фрагмент данных на пересечении строки и столбца таблицы (в ячейке).
- Команда/оператор (Statement) — способ объявить СУБД, что мы хотим выполнить какое-то действие.
- *Запрос* (Query) — особый вид команды, предназначенный для извлечения данных.
- `SELECT` — ключевое слово, с которого начинаются запросы. В следующих нескольких главах мы подробно рассмотрим способы применения ключевого слова `SELECT`.
- `FROM` — ключевое слово, указывающее на набор данных, к которому мы хотим обратиться для выполнения запроса.
- *Точка с запятой* — не забывайте завершать ваши запросы *точкой с запятой*!

2.5. Практическое занятие

В последующих разделах издания каждая глава будет завершаться одним или несколькими практическими заданиями, нацеленными на закрепление полученных знаний. Эти упражнения максимально приближены к реальным сценариям применения SQL и моделируют работу с данными из внешних источников, с которыми вам предстоит сталкиваться. Учитывая, что это занятие — первое, оно содержит ряд чисто умозрительных задач для понимания таблиц и их структуры.

Опираясь на то, что мы узнали о работе с `SELECT`-запросами (и допустив, что на рис. 2.1 представлена настоящая таблица), давайте поразмыслим над следующими важными вопросами:

1. Допустим, что таблица на рис. 2.1 — это настоящая таблица базы данных с несколькими столбцами. Мы также видели, что таблица, к которой обращается `SELECT`-команда, имеет как минимум один столбец. А может ли таблица не иметь ни одного столбца?
2. Теперь представьте, что в каком-либо из этих наборов данных нет ни одной строки с данными. Может ли таблица иметь ноль строк?
3. На рис. 2.1 в одной из ячеек столбца «Аванс» отсутствует значение. Как вы считаете, требуется ли заполнить это значение, имей мы дело с настоящей таблицей БД?
4. Предположим, имеется таблица `vegetable` со столбцом `Name` в базе данных `sqlnovel`. Что, по вашему мнению, произойдет, если объединить две команды

из этой главы и выполнить их одновременно? Каким будет результат такого запроса:

```
SELECT Name FROM vegetable;
SELECT Outcome FROM MyFirstQuery;
```

Будет ли такой SQL-сценарий успешно выполнен?

2.6. Ответы

1. Нет, таблица без столбцов невозможна. Именно поэтому мы определяем таблицу как структуру из одного или нескольких столбцов. Обязательно должен присутствовать хотя бы один столбец; иначе негде будет размещать значения данных.
2. Да, таблицы без строк вполне возможны. В момент создания любая таблица пуста — в ней нет ни одной строки.
3. Вопрос с подвохом! Ответ зависит от настроек таблицы. В наборе данных могут содержаться вполне корректные строки, в которых отсутствуют значения для определенных столбцов. Взять, к примеру, столбец «Второе имя» в таблице с персональными данными. Не у всех оно есть, и система должна это учитывать. При этом разработчик может установить обязательность заполнения для критически важных столбцов — в частности, сделать так, чтобы в той же таблице с персональными данными фамилия указывалась для всех без исключения.
4. Оба запроса будут выполнены, и в результате будут возвращены два отдельных набора данных. Именно для этого мы и ставим точку с запятой в конце команды — чтобы СУБД понимала, где заканчивается один запрос и начинается другой.

Что ж, отлично! Перейдем к главе 3, где откроем для себя новые интересные возможности составления запросов при помощи ключевого слова `SELECT`!

3 Извлечение данных

В главе 2 мы рассматривали электронную таблицу с книгами про SQL, стремясь разобраться в основах работы с таблицами реляционных баз данных (БД). Теперь мы возьмем очень похожую таблицу и посмотрим, как можно извлекать из нее данные при помощи команды `SELECT`.

Однако сначала следует внимательно изучить наш первый запрос. Несмотря на его простоту, он содержал все минимально необходимые компоненты. Давайте кратко разберем эти компоненты, а также ряд потенциальных проблем, связанных с оформлением кода и употреблением в нем некоторых слов.

3.1. Правила работы с инструкцией SELECT

В первых двух главах мы учились составлять запросы на основе естественного языка. Теперь пришло время обратить внимание на технические аспекты и формальные требования. Помните наш первый запрос? Он выглядел так:

```
SELECT Outcome FROM MyFirstQuery;
```

Код запроса состоит из четырех компонентов, каждый из которых представлен одним словом. Формально точка с запятой тоже является компонентом, выступая в качестве завершения команды, но так как она не всегда обязательна, мы не будем ее учитывать.

3.1.1. Требования к SELECT

Слова `Outcome` и `MyFirstQuery` указывают на то, какие данные мы выбираем. Они крайне важны — это тот минимум, который необходимо сообщить базе данных, чтобы получить данные из таблицы. А именно:

- какие данные нужно выбрать;
- откуда нужно выбрать данные.

В таком случае данные для выбора — это столбец `Outcome`, а место, откуда выбираются данные, — таблица `MyFirstQuery`. Оба этих слова следуют за определенными ключевыми словами SQL — `SELECT` и `FROM`, — при этом каждое из них формирует отдельное *предложение* (clause) в составе SQL-команды. Все SQL-команды состоят из набора предложений, а для извлечения данных из таблицы необходимо задействовать как минимум эти два. Предложения принято обозначать по ключевым словам, которые в них используются, поэтому эти два называются «предложение SELECT» (SELECT clause) и «предложение FROM» (FROM clause).

> **НА ЗАМЕТКУ** Сразу после `SELECT` мы указываем, какие именно данные хотим выбрать, а после `FROM` — откуда именно их брать. Это железное правило!

К примеру, мы не можем просто так поменять местами команды в нашем первом запросе:

```
FROM MyFirstQuery SELECT Outcome;
```

Попытка выполнить такую инструкцию приведет к синтаксической ошибке, поскольку предложение SELECT всегда должно предшествовать предложению FROM.

3.1.2. Ключевые и зарезервированные слова

Ключевые слова наподобие `SELECT` и `FROM` представляют собой подмножество служебных, *зарезервированных слов* (reserved words) языка SQL, понятных любой СУБД. Когда система видит их в запросе, она интерпретирует их как указание на выполнение определенного действия, связанного с соответствующим зарезервированным словом. Многие зарезервированные слова являются универсальными, однако некоторые из них специфичны для той или иной СУБД.

По мере освоения SQL запоминайте зарезервированные слова, служащие командами, чтобы случайно не использовать их в качестве имен таблиц или столбцов.

СОВЕТ Все зарезервированные слова можно посмотреть в официальной документации на веб-сайте MySQL, где вы скачали MySQL и MySQL Workbench. Каждая крупная система управления базами данных ведет свой перечень таких слов. И если вы работаете в средах разработки наподобие MySQL Workbench, эти служебные слова обычно подсвечиваются особым цветом.

Если задействовать зарезервированные слова в качестве имен таблиц или столбцов, неизбежны ошибки синтаксиса — СУБД просто не поймет, что вы от нее хотите. Допустим, в таблице MyFirstQuery есть столбец с именем Select, и вы пытаетесь выполнить следующий запрос:

```
SELECT Select FROM MyFirstQuery;
```

К практике!

Введите этот SQL-код в вашем подключении «Month of Lunches» в MySQL Workbench. Вы сразу заметите, что `Select` выделено цветом, отличным от `MyFirstQuery`, — это первый сигнал, что `Select` является зарезервированным словом. Помните: каждая СУБД имеет десятки или даже сотни зарезервированных слов, поэтому цветовая подсветка — ваш верный помощник в предотвращении ошибок.

При выполнении указанного запроса будет сгенерировано сообщение об ошибке[1]: You have an error in your SQL syntax (Ошибка в синтаксисе SQL). Система увидела два служебных слова `Select` подряд, что недопустимо — после первого `SELECT` нужно указать, что именно выбирать, а не повторять ту же команду.

3.1.3. Нечувствительность к регистру

Как видно из того же примера, `SELECT` и `Select` подсвечиваются как зарезервированные слова, несмотря на разницу в регистре. Ключевые слова являются независимыми от регистра, в силу чего любой из следующих запросов будет успешно выполнен и вернет одинаковый результат:

```
SELECT Outcome FROM MyFirstQuery;
Select Outcome From MyFirstQuery;
select Outcome from MyFirstQuery;
SeLeCt Outcome fRoM MyFirstQuery;
```

[1] Технически можно использовать зарезервированные ключевые слова (такие, как SELECT) в качестве имен таблиц или столбцов, заключив их в кавычки или скобки, — например, SELECT * FROM «SELECT» или SELECT * FROM [SELECT]. Однако это крайне не рекомендуется: такой код сложнее читать, легко допустить ошибку, и синтаксис сильно различается в зависимости от СУБД. — *Примеч. науч. ред.*

Однако возможность использования произвольного регистра для ключевых слов не означает, что так следует поступать. Для создания многократно применяемого SQL-кода большинство разработчиков пишут ключевые слова заглавными буквами, чтобы сделать код удобочитаемым и упростить поиск ошибок. В книге мы тоже будем придерживаться верхнего регистра для ключевых слов SQL.

> **ВНИМАНИЕ** Хотя ключевые слова SQL являются независимыми от регистра, информация, относящаяся к данным, может быть чувствительна к регистру в зависимости от настроек вашей СУБД. Будьте внимательны при указании имен таблиц, столбцов или значений в ваших запросах — они могут оказаться регистрозависимыми!

3.1.4. Форматирование и пробельные символы

Следует также отметить гибкость использования пробельных символов в запросах. Реляционная СУБД не придает значения их количеству, что позволяет форматировать запросы практически неограниченным количеством способов с применением пробелов, табуляций и переводов строк. Ваш первый запрос уместился в одну строку, но он выполнялся бы точно так же, будь он разделен на несколько строк. Так, все три приведенных ниже варианта запроса будут выполнены и возвратят идентичный результат.

Запрос 1

```
SELECT Outcome
FROM MyFirstQuery;
```

Запрос 2

```
SELECT
Outcome
FROM
MyFirstQuery;
```

Запрос 3

```
SELECT
       Outcome
FROM
       MyFirstQuery;
```

За отсутствием универсальных стандартов форматирования лучший совет, что я могу дать: будьте последовательны. Форматирование нужно прежде всего для удобства чтения, и, если вы найдете подходящий для вас способ оформления кода, применяйте и придерживайтесь его.

Полагаю, мы исчерпали весь учебный потенциал нашего первого запроса. Пришло время перейти к данным, приближенным к реальной практике, — таким, с которыми вам предстоит работать.

3.2. Извлечение данных из таблицы

В оставшейся части этой главы мы поработаем с таблицей `title` из базы данных `sqlnovel`. В отличие от нашего учебного примера `MyFirstQuery`, таблица `title` содержит несколько столбцов и множество строк данных.

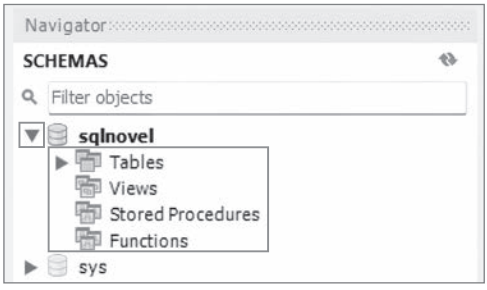


Рис. 3.1. Имя базы данных развернуто, что позволяет увидеть вложенные разделы Tables, Views, Stored Procedures и Functions

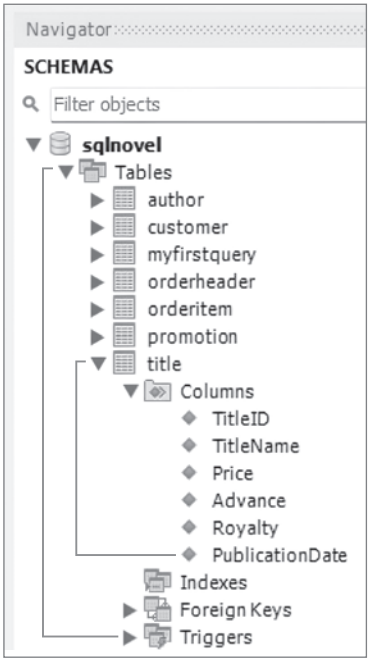


Рис. 3.2. Панель навигации (Navigator) с развернутым пунктом Tables, где отображены имена отдельных таблиц, и с развернутой таблицей title, где отображены все столбцы этой таблицы

Если вы не заглядывали в запросы, используемые для создания этой базы данных, то названия столбцов в таблице `title` вам неизвестны. К счастью, мы можем легко найти эту информацию в MySQL Workbench. Посмотрите в верхний левый угол панели навигации и обратите внимание на стрелки рядом с пунктами sqlnovel и Tables. Стрелка рядом с sqlnovel направлена вниз, указывая на то, что этот пункт развернут, благодаря чему мы видим все его содержимое: Tables (таблицы), Views (представления), Stored Procedures (хранимые процедуры) и Functions (функции), как демонстрирует рис. 3.1.

Стрелка рядом с Tables смотрит вправо — значит, список таблиц свернут. Чтобы увидеть столбцы в таблице `title` или любой другой таблице, нужно раскрыть список таблиц, нажав на стрелку. Затем необходимо найти таблицу `title`, нажать на стрелку рядом с ней, затем на стрелку рядом с Columns, чтобы развернуть и его. Выполнив все эти действия, мы увидим названия всех столбцов в таблице `title`, как показано на рис. 3.2.

3.2.1. Извлечение отдельного столбца

Теперь, когда известны имена столбцов, можно приступать к формированию запросов к таблице. Начнем с простого запроса к столбцу `TitleName` таблицы `title`. Учитывая, что в дальнейшем объем и сложность запросов будут расти, целесообразно уже сейчас форматировать их с размещением предложения `FROM` на отдельной строке, чтобы повысить удобочитаемость кода:

```
SELECT TitleName
FROM title;
```

К практике!

Наберите и выполните приведенный выше запрос. Считайте, что это уже второй ваш запрос (не то чтобы кто-то вел им счет).

Выполнение запроса возвращает восемь строк, приведенных на рис. 3.3. Ничего страшного, если строки у вас отображаются в другой последовательности. По умолчанию SQL не гарантирует порядок вывода результатов.



Рис. 3.3. Значения столбца TitleName в таблице title выводятся в произвольном порядке

ВНИМАНИЕ Подчеркну еще раз, поскольку это весьма распространенное заблуждение: SQL не гарантирует порядок выдачи результатов запроса по умолчанию! Не удивляйтесь, если один и тот же запрос возвращает одинаковые данные, но в разном порядке. Причин на то может быть масса — от изменений в значениях запрашиваемых таблиц до настроек сервера вашей базы данных. Помните: SQL — это декларативный язык, и, если вы явно не указали системе, каким образом следует упорядочить результаты, строки могут выводиться в случайном порядке. Впрочем, если вы уже заглянули вперед, то знаете, что в главе 4 мы как раз научимся управлять порядком выдачи результатов.

Следует обратить внимание и на то, что имя столбца в запросе — `TitleName`. Возникает закономерный вопрос: почему не назвать его просто `Name`? Основная

причина кроется в том, что многие столбцы в базах данных содержат данные со значениями названий (names) предметов или людей, в то время как `TitleName` указывает конкретно на заглавия произведений. Другая причина связана со сказанным выше о зарезервированных словах: в их число входит и слово `Name`.

К практике!

Если вы выполнили предыдущий запрос, попробуйте удалить из названия столбца `TitleName` префикс `Title`, оставив только `Name`. Вы заметили, что `Name` выделено теперь тем же цветом, что и `SELECT` и `FROM`? Все потому, что `Name` — зарезервированное слово в MySQL!

`Name` не является ключевым словом, таким как `SELECT` или `FROM`, и тем не менее, это зарезервированное слово, связанное с определенным видом операций в этой реляционной СУБД. В связи с этим не рекомендуется применять *Name* в качестве идентификатора столбца.

СОВЕТ Существует удобный прием, который сэкономит вам массу времени и сил: удалите Name из текста запроса, оставив пару пробелов между `SELECT` и `FROM`. Теперь переместите курсор в панель навигации. Нажмите и удерживайте TitleName в списке столбцов (Columns); затем перетащите его в панель запросов, поместив курсор между `SELECT` и `FROM`. Отпустив кнопку мыши, вы увидите, что *TitleName* автоматически вставилось в текст. Данный метод особенно полезен при работе с длинными именами столбцов; он также применим к именам таблиц.

3.2.2. Извлечение нескольких столбцов

До сих пор мы выбирали лишь один столбец данных, однако в реальности вам рано или поздно придется писать SQL-запросы, извлекающие данные из нескольких столбцов. Давайте снова сформулируем инструкцию обычным языком, как мы делали это в главе 2: “I would like all the TitleNames and Prices of the titles” («Хочу получить все названия книг (TitleNames) и цены (Prices) из таблицы title»).

Мы уже знаем, как преобразовать большую часть этой инструкции в запрос, и теперь нам нужно лишь перечислить столбцы через запятую. А чтобы несколько названий столбцов смотрелись элегантнее и лучше читались, изменим форматирование кода. В итоге наш запрос будет иметь следующий вид:

```
SELECT
    TitleName,
    Price
FROM title;
```

Запятая в SQL-запросе служит разделителем при перечислении элементов — в данном случае столбцов. Запятая ставится между элементами списка (TitleName, Price), но не после последнего элемента. Если поставить запятую после Price, это приведет к синтаксической ошибке — система будет ожидать следующий столбец, которого нет. Это работает так же, как и при обычном перечислении в письменной речи.

Также обратите внимание: порядок вывода столбцов определяете вы сами! Не важно, как они расположены в исходной таблице, — вы можете менять их порядок в выборке, как вам заблагорассудится:

```
SELECT
    Price,
    TitleName
FROM title;
```

При желании можно даже вывести один столбец несколько раз, вот так:

```
SELECT
    TitleName,
    TitleName,
    Price
FROM title;
```

Впрочем, наличие двух столбцов с одинаковыми именами и правда вносит изрядную путаницу.

Есть ли более эффективный способ управления несколькими столбцами с одинаковыми именами? Безусловно, такой способ существует!

3.2.3. Переименование столбцов в выборке при помощи псевдонимов

Хотя `SELECT` и не может осуществлять правку имен столбцов в исходных таблицах, можно легко поменять имя выходного столбца на любое желаемое. Давайте снова сформулируем инструкцию: “I would like all the TitleNames as BookNames of the titles” («Мне нужно получить все названия книг (TitleNames) как BookNames из таблицы title»).

Подобно тому как в речи мы употребляем союз *as* («как») в значении «в качестве, в роли, под именем», в SQL-команде мы задействуем слово `AS`, чтобы задать новое имя (псевдоним) для столбца в результатах выборки:

```
SELECT
    TitleName AS BookName
FROM title;
```

Теперь название столбца в результатах будет отображаться как `BookName` вместо `TitleName` (рис. 3.4).

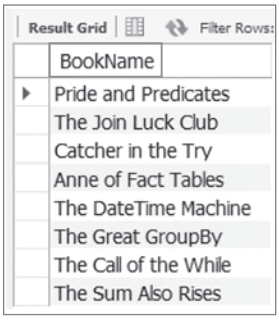


Рис. 3.4. Значения столбца TitleName теперь выводятся в столбце под именем BookName

Здесь мы применили *псевдоним* (alias) — простой способ изменить в SQL-запросе исходное имя столбца в результатах выборки. Присваивая выходным столбцам уникальные имена, мы можем избежать путаницы в том случае, когда в разных таблицах имеются столбцы с одинаковыми именами:

```
SELECT
    TitleName AS BookName,
    TitleName AS AlsoBookName,
    Price
FROM title;
```

Кстати сказать, для применения псевдонима слово `AS` можно и пропустить, указав *имя псевдонима* непосредственно после исходного имени столбца, хотя такой формат делает ваши псевдонимы столбцов чуть менее очевидными:

```
SELECT
    TitleName BookName,
    TitleName AlsoBookName,
    Price
FROM title;
```

Псевдонимы — прекрасный способ сделать названия столбцов более понятными! Помните лишь, что следует избегать употребления зарезервированных слов в качестве псевдонимов.

3.2.4. Извлечение всех столбцов

Итак, мы уже умеем выбирать и один, и несколько столбцов данных из таблицы. Однако на практике нередко возникает необходимость извлечения всех столбцов таблицы для комплексного анализа — скажем, когда требуется составить детализированный отчет с максимально полными данными о продажах или

предоставить аудиторам исчерпывающие сведения о клиентах. Как бы то ни было, существует три метода для решения задачи.

Первый способ — просто перечислить все названия столбцов через запятую. Но он самый трудоемкий, особенно если таких имен много, они длинные и вы не очень быстро печатаете.

Второй способ еще проще: выделите первый столбец таблицы в панели навигации, щелкнув на нем мышью, а затем, удерживая нажатую клавишу Shift, щелкните на последнем столбце. При этом будет выделен весь диапазон столбцов, как отмечено на рис. 3.5.

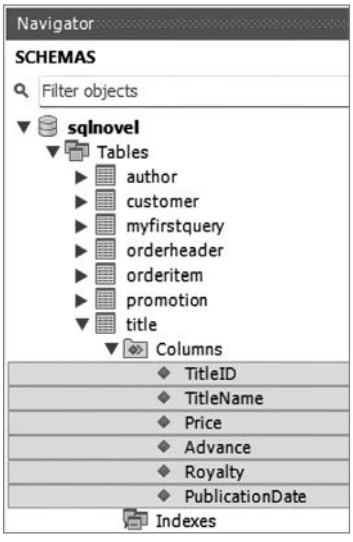


Рис. 3.5. Названия столбцов таблицы title выделены после щелчка мышью на имени первого столбца, нажатия клавиши Shift и щелчка по имени последнего столбца

Далее нажмите и удерживайте любой из выделенных столбцов, переместите курсор между командами `SELECT` и `FROM` в панели запросов, а затем отпустите клавишу мыши. Теперь ваш запрос содержит полный перечень названий столбцов, как показано на рис. 3.6.

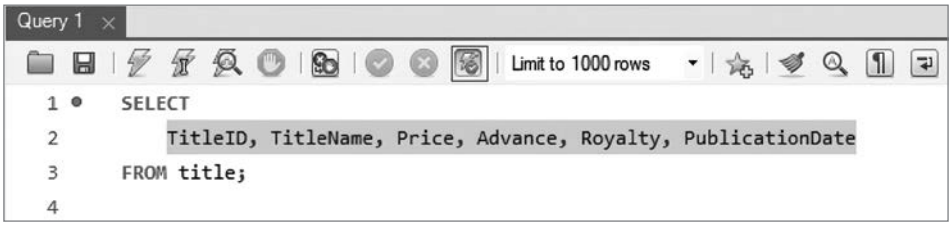


Рис. 3.6. Столбцы таблицы title после их вставки в панели запросов

К практике!

Выполните этот запрос. В панели результатов отобразятся значения всех столбцов таблицы. Если правая часть запроса перекрыта другой панелью, можете скрыть ее, выбрав в меню: View ▸ Panel ▸ Hide Secondary Sidebar (Вид ▸ Панель ▸ Скрыть дополнительную боковую панель).

Третий метод, вероятно, является самым распространенным, поскольку он требует минимальных усилий. Можно извлечь все столбцы таблицы, заменив имена столбцов символом «звездочка» (*), который на профессиональном жаргоне обычно называют select star («звездочкой»), реже — select all («выбрать все»):

```
SELECT
    *
FROM title;
```

Результат точно такой же, как у предыдущего запроса, а хлопот — минимум! Уверен, вы понимаете, отчего он так популярен. Одна звездочка заменяет десятки нажатий клавиш!

Помимо экономии времени и сил, дополнительное преимущество подобного метода заключается в возможности оперативного просмотра всех названий столбцов таблицы без обращения к панели навигации. Однако пользоваться `SELECT *` следует с осторожностью по двум веским причинам:

- Полная выборка означает, что СУБД должна считывать больше данных и передать их через сеть в качестве результата. Может показаться, что ресурсы бесконечны, но по моему опыту, использование `SELECT *` на очень больших таблицах потребляет такой объем системных ресурсов, что вызывает проблемы с производительностью для других запросов. Поэтому рекомендую прибегать к этому методу с большой осмотрительностью.
- Вторая опасность применения `SELECT *` обусловлена отсутствием явного указания порядка столбцов, что делает его непригодным для повторяющихся запросов. Предположим, SQL-запрос для отчета ожидает ровно пять выходных столбцов из таблицы. Если впоследствии структура таблицы изменится (будут добавлены или удалены столбцы), в отчете может произойти сбой — ведь количество столбцов станет другим!

ВНИМАНИЕ Учитывая вышеупомянутые риски, конструкцию `SELECT *` следует применять лишь при необходимости, в разовых запросах, и то с крайней осторожностью. В конце концов, можно всегда прибегнуть ко второму способу — ведь щелкнуть и перетащить нужные имена столбцов в текст запроса совсем не сложно!

3.3. Практическое занятие

1. В базе данных `sqlnovel` имеется таблица с названием `author`. Какими двумя способами можно воспользоваться, чтобы узнать имена столбцов в этой таблице?
2. Вам нужно написать запрос, выводящий все имена и фамилии из таблицы `author`. Как бы вы составили такой запрос?
3. Что, на ваш взгляд, произойдет, если вы забудете поставить запятую между именами столбцов? Как вы считаете, следующий запрос будет работать, и если да, то каков будет результат?

```
SELECT
    TitleName
    Price
FROM title;
```

4. Что произойдет, если вы попытаетесь применить метод `SELECT *` с псевдонимом, как в следующем запросе?

```
SELECT
    * AS Everything
FROM title;
```

3.4. Ответы

1. Проще всего раскрыть список столбцов под таблицей `author` в панели навигации. При работе с интерфейсом, не предоставляющим такую возможность, можно задействовать метод `SELECT *` в форме следующего запроса:

```
SELECT
    *
FROM author;
```

2. Ответ:

```
SELECT
    FirstName,
    LastName
FROM author;
```

3. Запрос выполнится, но с результатом, отличным от ожидаемого! Без запятой между `TitleName` и `Price` система решит, что слово «Price» — это псевдоним столбца для `TitleName`. Будет выведен один столбец с псевдонимом `Price`, а значения будут взяты из столбца `TitleName`.
4. Данный запрос не выполнится, поскольку нельзя использовать псевдонимы для имен столбцов с `SELECT *`. При попытке составить такой запрос напротив строки с псевдонимом появится красный квадрат с белым крестиком. Это сигнал, означающий, что приложение Workbench нашло синтаксическую ошибку.

4 Сортировка, выборочное исключение и комментирование данных

Как мы отметили в главе 3, реляционная СУБД не обеспечивает предсказуемого порядка возврата результатов. Такое поведение является архитектурной особенностью, поскольку различные запросы могут требовать или не требовать сортировки данных, и СУБД оптимизирует процесс извлечения данных наиболее эффективным для себя способом. И хотя результаты могут отображаться в определенном порядке, к примеру в порядке добавления строк в таблицу, в самой системе не существует никаких внутренних указаний относительно того, как следует упорядочить данные в результатах запроса. Поэтому, если мы хотим быть уверенными в порядке выдачи результатов, нам нужно явно обозначить такой порядок в нашем SQL-запросе.

Ряд интересных функциональных возможностей, связанных с упорядочиванием данных, позволяет управлять количеством строк, возвращаемых запросом. Это особенно полезно при работе с огромными таблицами, содержащими миллионы или миллиарды строк, когда нужно увидеть лишь самые последние записи или записи с наименьшими или наибольшими значениями.

И поскольку мы только приступили к освоению SQL, здесь мы также поговорим о комментариях в этом языке программирования. Комментарии являются незаменимым средством как для вас самих, так и для тех, кто будет читать ваш код. Если вы хотите стать профессионалом в составлении запросов, с самого начала привыкайте грамотно их комментировать.

4.1. Сортировка данных

Куда ни глянь, повсюду мы обнаруживаем предметы и явления, организованные в определенном порядке. Книги в библиотеке сортируются по имени автора,

этажи в здании — по номеру, а события в ежедневнике — по дате и времени. Все вокруг упорядочено для удобства, доступности и практического применения, и с нашими рабочими данными должно быть точно так же!

4.1.1. Сортировка по одному столбцу

Посмотрим, как это делается в SQL! Вспомним нашу словесную формулировку из третьей главы: “I would like all the TitleNames and Prices of the titles” («Мне нужны все названия книг (TitleNames) и их цены (Prices) из таблицы title»). Вот как выглядит соответствующий запрос:

```
SELECT
    TitleName,
    Price
FROM title;
```

Как видно на рис. 4.1, данный запрос возвращает неупорядоченную выборку.

TitleName	Price
Pride and Predicates	9.95
The Join Luck Club	9.95
Catcher in the Try	8.95
Anne of Fact Tables	12.95
The DateTime Machine	7.95
The Great GroupBy	10.95
The Call of the While	8.95
The Sum Also Rises	7.95

Рис. 4.1. Выдача по запросу столбцов TitleName и Price из таблицы title. Результаты не отсортированы ни по одному из этих столбцов

Теперь давайте скажем, что хотим получить те же результаты, но отсортированные по названию книги: “I would like all the TitleNames and Prices of the titles, and I would like the results ordered by TitleName” («Мне нужны все названия книг (TitleNames) и их цены (Prices) из таблицы title, чтобы результаты при этом были упорядочены по TitleName»). Как и прежде, SQL-запрос будет почти таким же, как эта фраза. Чтобы составить его, добавим новое предложение, применив ключевое слово `ORDER BY` («упорядочить по»):

```
SELECT
    TitleName,
    Price
FROM title
ORDER BY
    TitleName;
```

При выполнении этой команды мы видим, что возвращаемые строки те же, что и ранее, но теперь они отсортированы в соответствии с запросом, как продемонстрировано на рис. 4.2.

TitleName	Price
Anne of Fact Tables	12.95
Catcher in the Try	8.95
Pride and Predicates	9.95
The Call of the While	8.95
The DateTime Machine	7.95
The Great GroupBy	10.95
The Join Luck Club	9.95
The Sum Also Rises	7.95

Рис. 4.2. Та же выборка, теперь отсортированная в алфавитном порядке по столбцу TitleName

Предложение ORDER BY в структуре SQL-команды, как правило, должно размещаться последним. За исключением одного особого случая (к которому мы вернемся позже в этой главе), любое предложение, добавленное после ORDER BY в тексте запроса, приведет к возникновению синтаксической ошибки.

Это правило легко запомнить, если учесть, что упорядочивание данных является заключительной операцией, выполняемой СУБД. Этот аспект часто истолковывается неверно; так, многие пользователи SQL ошибочно полагают, что ORDER BY определяет порядок считывания данных. Тогда как в реальности система должна сначала завершить операции для остальной части вашей SQL-команды, и лишь после формирования итоговой выборки она компонует данные в соответствии с предложением ORDER BY.

> **ВНИМАНИЕ** Дополнительная вычислительная нагрузка, обусловленная применением ORDER BY, является незначительной для запросов, приведенных в книге, поскольку итоговые выборки содержат всего несколько строк. Однако при работе с наборами данных, содержащими миллионы или миллиарды записей, добавление ключевого слова ORDER BY может катастрофически снизить производительность. Более того, сортировка крупных массивов данных требует значительных ресурсов ЦП и оперативной памяти, что может привести к падению производительности одновременно выполняемых запросов других пользователей. Всегда проявляйте осмотрительность, когда задействуете ORDER BY.

Учитывая, что реляционная СУБД сначала формирует всю выборку и лишь потом компонует ее, в предложении ORDER BY можно спокойно задействовать псевдонимы столбцов. Псевдонимы применяются после создания результирующего набора, но до того, как он упорядочен, что позволяет воспользоваться следующим SQL-алгоритмом для сортировки данных (см. рис. 4.3):

```
SELECT
    TitleName AS NameOfTheBook,
    Price
FROM title
ORDER BY
    NameOfTheBook;
```

NameOfTheBook	Price
Anne of Fact Tables	12.95
Catcher in the Try	8.95
Pride and Predicates	9.95
The Call of the While	8.95
The DateTime Machine	7.95
The Great GroupBy	10.95
The Join Luck Club	9.95
The Sum Also Rises	7.95

Рис. 4.3. После применения ORDER BY выборка по предыдущему запросу отсортирована по столбцу TitleName, которому присвоен псевдоним NameOfTheBook

4.1.2. Сортировка по нескольким столбцам

Разумеется, `ORDER BY` позволяет выполнять сортировку не только по одному столбцу. Чтобы упорядочить набор данных сразу по нескольким столбцам, в предложении `ORDER BY` допускается перечисление нескольких столбцов через запятую — точно так же, как мы делаем это в предложении `SELECT`. Рассмотрим следующий пример запроса:

```
SELECT
    TitleName,
    Advance,
    Royalty
FROM title
ORDER BY
    Advance,
    Royalty;
```

Как видно на рис. 4.4, результаты нашего запроса отсортированы в первую очередь по столбцу `Advance` («Аванс»), от меньшего к большему. Но когда значения в `Advance` одинаковы — как в третьей, четвертой и пятой строках, где оно равно 5000.00, — для упорядочивания этих трех строк служит столбец `Royalty` («Гонорар»).

TitleName	Advance	Royalty
The Great GroupBy	0.00	20.00
The Call of the While	2500.00	15.00
Catcher in the Try	5000.00	10.00
The Sum Also Rises	5000.00	12.00
Pride and Predicates	5000.00	15.00
The DateTime Machine	5500.00	15.00
The Join Luck Club	6000.00	12.00
Anne of Fact Tables	10000.00	15.00

Рис. 4.4. Результаты запроса, отсортированные по столбцу Advance, а затем по столбцу Royalty, в обоих случаях от меньшего к большему

К практике!

Теперь, когда вы ознакомились с несколькими способами сортировки данных в таблице заглавий, попробуйте применить ORDER BY, как продемонстрировано в примерах выше, или выполните сортировку по другим столбцам, таким как `Price` (Цена) или `PublicationDate` (Дата выхода издания).

4.1.3. Указание направления сортировки

При упорядочивании данных посредством `ORDER BY` подразумевается неявное направление сортировки в порядке *возрастания* (ascending order) — либо по алфавиту от A до Z, либо в численном выражении от меньшего к большему значению. Также можно выстроить результаты в обратном направлении, в порядке *убывания* (descending order) — от Z до A или от наибольшего к наименьшему числовым значениям. Сортировку в убывающем порядке необходимо указывать явно, путем добавления модификатора `DESC` после имени столбца в предложении `ORDER BY`. Вот пример предыдущего запроса, данные которого сортируются по столбцу `Advance` в убывающем порядке:

```
SELECT
    TitleName,
    Advance,
    Royalty
FROM title
ORDER BY
    Advance DESC,
    Royalty;
```

Вывод на рис. 4.5 показывает, что результаты теперь упорядочены по значениям столбца `Advance` от наибольшего к наименьшему. Однако внимательно изучите четвертую, пятую и шестую строки: они все еще выстроены по значению `Royalty` от наименьшего к наибольшему. Столбец `Royalty` отсортирован в порядке возрастания, поскольку для этого столбца не было задано иного порядка.

TitleName	Advance	Royalty
Anne of Fact Tables	10000.00	15.00
The Join Luck Club	6000.00	12.00
The DateTime Machine	5500.00	15.00
Catcher in the Try	5000.00	10.00
The Sum Also Rises	5000.00	12.00
Pride and Predicates	5000.00	15.00
The Call of the While	2500.00	15.00
The Great GroupBy	0.00	20.00

Рис. 4.5. Теперь результаты отсортированы по столбцу Advance в порядке убывания и по столбцу Royalty в порядке возрастания

Для большей ясности можно прямо указать порядок сортировки по столбцу `Royalty`, добавив к предложению `ORDER BY` модификатор `ASC`, обозначающий возрастающий порядок данных. Выглядит это так:

```
SELECT
    TitleName,
    Advance,
Royalty
FROM title
ORDER BY
    Advance DESC,
    Royalty ASC;
```

СОВЕТ Неявный (заданный по умолчанию) порядок по возрастанию столбца, указанного в предложении `ORDER BY`, может сбивать с толку при чтении запросов, поэтому при написании SQL-кода, который будут читать другие, рекомендуется выработать привычку явно обозначать направление сортировки, даже если для порядка по возрастанию этого не требуется. Следует всегда стремиться сделать ваш программный код предельно понятным и легким для восприятия.

4.1.4. Сортировка по скрытым столбцам

Возникают случаи, когда необходимо скомпоновать результаты по столбцу, который не требуется возвращать в наборе результатов. Так, можно смело упорядочить результат запроса по одному или нескольким столбцам, которые не отображаются в выводе, оставаясь скрытыми. Допустим, вы решили изменить предыдущий запрос, чтобы он возвращал только столбец `TitleName`, но при этом сортировал результаты по столбцу `Advance` в порядке убывания и по столбцу `Royalty` в порядке возрастания:

```
SELECT
    TitleName
FROM title
ORDER BY
    Advance DESC,
    Royalty ASC;
```

Данные, представленные на рис. 4.6, идентичны результатам предыдущего запроса, за исключением того, что в выдаче отсутствуют столбцы `Advance` и `Royalty`.

TitleName
Anne of Fact Tables
The Join Luck Club
The DateTime Machine
Catcher in the Try
The Sum Also Rises
Pride and Predicates
The Call of the While
The Great GroupBy

Рис. 4.6. В результатах представлены только значения столбца TitleName, хотя строки по-прежнему отсортированы по столбцу Advance в порядке убывания и по столбцу Royalty в порядке возрастания

Как вообще удается осуществить сортировку по данным, которые не включены в предложение `SELECT`? СУБД выполняет этот маленький фокус, добавляя столбцы `Advance` и `Royalty` к результирующему набору перед его возвратом, затем компонует данные в соответствии с условиями запроса и, наконец, возвращает только столбцы, указанные в предложении `SELECT`. Безусловно, процесс этот требует дополнительных вычислительных затрат, о чем следует помнить при работе с большими массивами данных.

4.1.5. Сортировка по позиции

Если названия столбцов кажутся слишком длинными, чтобы набирать их вручную, существует более быстрый способ указать порядок сортировки: вместо имени столбца укажите его *порядковый номер* (numerical column position) в предложении `SELECT`. К примеру, если в запросе столбцы перечислены в последовательности `TitleName` (1), `Advance` (2) и `Royalty` (3), то допустимо ссылаться на них просто по их номерам:

```
SELECT
    TitleName,
    Advance,
    Royalty
FROM title
ORDER BY
    2 DESC,
    3 ASC;
```

Порядок сортировки указан как убывающий для столбца `Advance` и возрастающий для столбца `Royalty`. Мы знаем это, поскольку `Advance` является вторым, а `Royalty` третьим столбцом в предложении `SELECT`, обозначенными соответственно цифрами 2 и 3 в предложении `ORDER BY`.

> **ВНИМАНИЕ** Сокращенная форма записи в `ORDER BY` может пригодиться при составлении разовых запросов на скорую руку, однако из-за худшей читаемости по сравнению с явным указанием имен сортируемых столбцов следует избегать такого приема при написании SQL-кода, предназначенного для многократного применения. Представьте себе, если столбцы в предложении `SELECT` изменятся, сортировка по позиции будет осуществляться по другим столбцам.

4.2. Ограничение выборки

До сих пор наши запросы включали все данные из таблицы. А если полная выборка вам не нужна? Иногда требуется просмотреть лишь несколько строк, а бывает, что достаточно и одной, тогда как остальную часть результатов можно спокойно пропустить, чтобы, к примеру, ознакомиться с неизвестной таблицей и понять, как в ней отформатированы данные. В SQL предусмотрена и такая возможность!

4.2.1. Ограничение выборки при помощи LIMIT

Сначала по традиции сформулируем на простом английском языке, что хотим найти только три опубликованные книги: “I would like all the TitleNames and PublicationDates, but limit the results to the first three rows” («Мне нужны все названия книг (TitleNames) и даты публикации (PublicationDates), но ограничь результаты первыми тремя строками»). Новое ключевое слово — `LIMIT`

(«ограничивать») — позволяет сократить выборку до указанного количества строк. Вот как это работает:

```
SELECT
    TitleName,
    PublicationDate
FROM title
LIMIT 3;
```

СУБД захватывает первые три найденные строки, и ваш результирующий набор должен походить на представленный на рис. 4.7.

TitleName	PublicationDate
Pride and Predicates	2015-04-30 00:00:00
The Join Luck Club	2016-02-06 00:00:00
Catcher in the Try	2017-04-03 00:00:00

Рис. 4.7. Результаты применения модификатора LIMIT для извлечения только трех строк

Благодаря `LIMIT` выборка ограничивается тремя строками из восьми имеющихся в таблице. Несмотря на то что подобного рода модификатор может на первый взгляд показаться бесполезным, его значение трудно переоценить в случаях, когда необходимо оперативно получить образцы имен столбцов и типов содержащихся в них данных.

К практике!

При помощи `SELECT *` и `LIMIT` напишите запрос, позволяющий быстро получить выборку нескольких строк из таблицы title. Возможно, что среди выбранных строк окажутся три заглавия, представленные на рис. 4.7, а возможно, нет, коль скоро вы не указываете порядок сортировки.

В связи с предназначением предыдущего запроса для пробной выборки данных результаты могут быть неточными. Давайте сначала сформулируем наши намерения на естественном английском языке более точно, то есть запросим сведения о трех самых последних опубликованных книгах: “I would like all the TitleNames and PublicationDates, but limit the results to the three most recent PublicationDates” («Мне нужны все названия книг (TitleNames) и даты публикации (PublicationDates), но ограничь результаты тремя самыми свежими публикациями»).

Для выполнения требуемого запроса мы добавим сортировку по убыванию для столбца `PublicationDate` посредством ключевого слова `ORDER BY`:

```
SELECT
    TitleName,
```

```
    PublicationDate
FROM title
ORDER BY PublicationDate DESC
LIMIT 3;
```

Обратите внимание на порядок предложений в команде: предложение `LIMIT` следует за предложением `ORDER BY`. Предложение `LIMIT` является единственным, которое должно располагаться после предложения `ORDER BY`, и при наличии всегда завершает SQL-запрос. Размещение предложения `LIMIT` в любой другой части команды приведет к синтаксической ошибке. На рис. 4.8 приведены результаты запроса, возвращающие три самых свежих издания (`TitleName`) вместе со значениями даты их выхода в печать (`PublicationDate`).

TitleName	PublicationDate
The Sum Also Rises	2021-11-12 00:00:00
The Call of the While	2020-03-14 00:00:00
The Great GroupBy	2019-12-23 00:00:00

Рис. 4.8. Три последних опубликованных произведения (TitleNames) и соответствующие даты публикации (PublicationDates) при использовании сортировки ORDER BY по столбцу PublicationDate в сочетании с ограничением LIMIT до трех строк

СОВЕТ Несмотря на отсутствие строгой необходимости, предложение `LIMIT` почти всегда используется в сочетании с предложением `ORDER BY`. Почему? Просто потому, что вам наверняка захочется считать пробный набор строк, отсортированных по какому-либо условию, будь то возраст данных или их объем. Применение `LIMIT` без указания порядка сортировки может привести к выдаче случайных и непредсказуемых результатов.

4.2.2. Применение OFFSET к ограниченной выборке данных

Сценарий написания SQL-запроса для поиска самых свежих данных не является редким, но иной раз требуется осуществить пробную выборку, минуя крайние значения. В таких случаях можно воспользоваться еще одной функцией предложения `LIMIT`, а именно прибегнуть к его полезному дополнению — `OFFSET` («смещение»). Ключевое слово `OFFSET` нельзя задействовать вне предложения `LIMIT`; оно сообщает системе, что нужно пропустить указанное число строк перед возвратом ограниченного набора данных, заданного в предложении `LIMIT`. Повторно запустим предыдущий запрос, однако на этот раз применим ключевое слово `OFFSET`, чтобы пропустить первую возвращаемую строку:

```
SELECT
    TitleName,
    PublicationDate
FROM title
ORDER BY PublicationDate DESC
LIMIT 3 OFFSET 1;
```

TitleName	PublicationDate
The Call of the While	2020-03-14 00:00:00
The Great GroupBy	2019-12-23 00:00:00
The DateTime Machine	2019-02-04 00:00:00

Рис. 4.9. Три самых поздних по дате пары значений столбцов TitleName и PublicationDate после применения OFFSET для пропуска первой строки

На рис. 4.9 показано, что строка с заглавием книги «The Sum Also Rises» пропущена, и теперь результаты включают еще одну строку, «The DateTime Machine», имеющую более раннюю дату выхода в свет.

4.2.3. Ограничение выборки в других СУБД

В первой главе мы обсуждали то обстоятельство, что у каждой реляционной СУБД встречаются собственные реализации некоторых команд. К сожалению, к числу таких команд относится и предложение `LIMIT`.

> **ВНИМАНИЕ** Предложение `LIMIT` поддерживается многими популярными СУБД, включая MySQL, MariaDB, PostgreSQL и SQLite, однако оно не совместимо с DB2, Oracle или SQL Server. В этих СУБД используются собственные специальные команды.

4.3. Комментарии к данным

На протяжении всей главы рассматривались методы сортировки и пропуска строк в запросах. Я попытался снабдить каждый запрос пояснением относительно его назначения и особенностей выполнения, но если бы кто-нибудь прочел только написанный нами SQL-код, то разве бы он понял, почему данный запрос составлен именно так? Вероятно, нет, поэтому сейчас самое время поговорить о комментариях. *Комментарии* (comments) позволяют включать в запрос текст, который не учитывается при выполнении программы. Обычно такой текст содержит заметки автора, описывающие функционал запроса, а также его идентификатор и дату написания или изменения. По существу, комментарий может включать любую необходимую информацию в дополнение к самому SQL-коду.

Зачем нужны комментарии? Ваши запросы будут читать не только СУБД, но и другие пользователи. Это могут быть коллеги, работающие с вашим сценарием, или кто-то сменивший вас на вашей позиции, когда вы продолжите карьерный рост благодаря углубленным знаниям SQL. Комментарии могут быть как простыми, содержащими лишь ваше имя и дату создания SQL-сценария, так и подробными, включающими построчные описания того, что должен выполнять каждый конкретный фрагмент программного кода.

Отсутствие комментариев в SQL-коде порождает неясность. Другим придется часами ломать голову над вашим запросом в попытке разгадать ваши намерения. Хуже того, и вы сами, взглянув на собственный SQL-код спустя недели, месяцы

или даже годы, поневоле задумаетесь, что именно ваш «прежний я» пытался в нем реализовать.

Умение создавать полезные содержательные комментарии — отличительная черта любого профессионального SQL-разработчика. Коллеги будут выше ценить вашу работу, поскольку дополнительные секунды или минуты, затраченные на написание понятных комментариев, сэкономят им (и вашему «будущему я») часы неразберихи.

Существует несколько способов написания комментариев в SQL. Во-первых, можно закомментировать отдельную строку, предварив ее двумя последовательными дефисами:

```
-- Данный запрос выводит три произвольно взятые строки
SELECT
    TitleName,
    PublicationDate
FROM title
LIMIT 3;
```

Применение двух дефисов позволяет закомментировать одну строку кода до следующего перевода строки. Этот тип комментария называется *однострочным* (inline comment). Во-вторых, в MySQL также доступно создание однострочного комментария при помощи символа решетки (#):

```
# Данный запрос выводит первые 3 строки по убыванию значения PublicationDate
SELECT
    TitleName,
    PublicationDate
FROM title
ORDER BY PublicationDate DESC
LIMIT 3;
```

Это менее распространенный тип комментариев, поэтому имейте в виду, что другие СУБД могут его не понять.

И наконец, в-третьих, можно заключить комментарий между символами /* и */, что позволяет создавать многострочные комментарии, как в следующем примере:

```
/* Данный запрос выводит первые 3 книги (TitleName)
... по убыванию даты публикации (PublicationDate),
...исключая одно, самое свежее издание (TitleName) */
SELECT
    TitleName,
    PublicationDate
FROM title
ORDER BY PublicationDate DESC
LIMIT 3 OFFSET 1;
```

Кроме того, можно заключать однострочные комментарии внутри многострочных. Скажем, вы пометили однострочным комментарием определенную SQL-команду,

но позже решили полностью ее закомментировать при помощи многострочного комментария, заменив ее другим SQL-кодом. Вот как это выглядит:

```
/*
# Данный запрос выводит три произвольно взятые книги, но он нас не устраивает
SELECT
    TitleName,
    PublicationDate
FROM title
LIMIT 3;
*/

-- Обновленный запрос выводит теперь позднейшие издания по дате публикации
SELECT
    TitleName,
    PublicationDate
FROM title
ORDER BY PublicationDate DESC
LIMIT 3;
```

> **СОВЕТ** Благодаря большей функциональности многострочные комментарии (с использованием `/*` и `*/`) являются предпочтительными для многократно применяемого кода. Они особенно полезны, когда требуется закомментировать целые разделы SQL-сценария — к примеру, старую версию запроса, которая может еще пригодиться, или участок кода с ошибками, который необходимо доработать.

Возвращаясь спустя недели, месяцы и даже годы к составленному вами программному коду и пытаясь в нем разобраться, вы обнаружите, насколько бесценны ваши комментарии. За десятилетия работы с SQL мне постоянно приходилось просматривать комментарии в старых запросах, чтобы разобраться, как и в каких целях эти запросы были составлены. Не раз бывало и наоборот: открываешь чужой сценарий, не снабженный комментариями, и часами ломаешь голову, пытаясь разгадать замысел разработчика.

Сделайте себе одолжение: заведите привычку кропотливо комментировать любой SQL-код, который вы пишете, вне зависимости от того, насколько он прост. Процесс составления комментария займет всего несколько секунд, зато, как я уже говорил, это сэкономит вам или кому-то еще, кто будет работать с вашей программой, гораздо больше времени. Именно по этой причине весь SQL-код в сценариях, прилагаемых к этому изданию, снабжен комментариями, помогающими уяснить назначение каждого запроса.

4.4. Практическое занятие

1. Необходимо получить список всех авторов в алфавитном порядке. Напишите запрос, который выдаст столбцы `FirstName` и `LastName` для всех авторов, отсортировав результат сначала по `LastName`, а затем по `FirstName`.

2. Требуется написать запрос, который выведет все столбцы из таблицы `title` только для самого дорогого издания. Как будет выглядеть такой запрос?
3. Допустим, имеется следующий SQL-запрос, из которого выполняющее его приложение удаляет все символы новой строки, в результате чего весь его код размещается в одной строке. Каков будет результат выполнения такого запроса?

```
-- Retrieve the book titles
SELECT TitleName
FROM title;
```

4.5. Ответы

1.

```
SELECT
    FirstName,
    LastName
FROM author
ORDER BY
    LastName,
    FirstName;
```

2.

```
SELECT
    TitleID,
    TitleName,
    Price,
    Advance,
    Royalty,
    PublicationDate
FROM title
ORDER By
    Price DESC
LIMIT 1;
```

   В качестве альтернативного подхода вместо перечисления всех имен столбцов можно прибегнуть к конструкции `SELECT *`.

3. Поскольку теперь запрос размещается в одной строке, перед которой стоят два дефиса, весь его код интерпретируется как комментарий. К синтаксической ошибке это не приведет, но и не выведет ожидаемые результаты. Вот одна из причин, по которой в повторно применяемом коде предпочтительны многострочные комментарии (`/*` и `*/`) — они четко ограничивают начало и конец закомментированной строки (или строк).

5 Выборка по заданным условиям

До сих пор мы в основном писали запросы, которые возвращают полный набор данных, однако по мере написания более содержательного SQL-кода с применением больших наборов данных вы обнаружите, что на практике чаще всего необходимо выбрать лишь некое подмножество записей вместо всего их массива. В главе 4 мы освоили ключевые слова `LIMIT` и `OFFSET` для сокращения числа возвращаемых данных, однако эти команды не предназначены для поиска конкретных строк.

Так, вам может потребоваться вывести отчет о продажах за последний месяц, перечень заказов со статусом «в ожидании» или список клиентов из Нью-Гэмпшира. В каждом из этих сценариев имеются *условия* для извлечения конкретных данных, и мы задаем эти условия при помощи различных фильтров. *Фильтрация* (filtering), или *выборка*, означает применение одного или нескольких условий к более широкому набору данных для ограничения возвращаемых результатов. В этих целях в основном используется другой вид предложения — с ключевым словом `WHERE`.

Весьма вероятно, что большинство SQL-запросов, составленных вами на протяжении вашей карьеры, будет включать предложение `WHERE` — ведь практически всегда нужно искать данные по каким-то условиям. Предложение `WHERE` обладает исключительной мощностью и предоставляет столько методов фильтрации данных, что их обзор займет несколько глав. Итак, приступим!

5.1. Фильтрация по одному условию

Самые простые способы фильтрации данных интуитивно понятны и легко осваиваются. Их основные разновидности связаны с типом данных, который вы запрашиваете. Как вы могли заметить, существуют различные типы

данных — такие, как имена, числа и даты, — и правила фильтрации для каждого типа имеют небольшие отличия. В этом разделе мы рассмотрим их все, начиная с фильтрации по условию с числовым значением.

5.1.1. Фильтрация по числовым значениям

Допустим, мы хотим узнать названия книг (`TitleName`), для которых авторский аванс (`Advance`) составил 10 000 долларов. Начнем с формулировки предложения: "I would like the title name of the title where the advance is 10,000 dollars" («Мне нужно название книги из таблицы title, где аванс составляет 10 000 долларов»).

Заметьте, что в обычной речи для указания условия выбора мы не только употребляем слово *where* («где»), но и ставим его в конец высказывания. В SQL мы действуем точно так же! Вот как будет выглядеть соответствующий запрос:

```
SELECT TitleName
FROM title
WHERE Advance = 10000.00;
```

Давайте пристально рассмотрим предложение `WHERE` и изучим правила, которым оно подчиняется:

- Как и в предыдущем SQL-запросе, предложение `WHERE` следует после предложения `FROM`, что соответствует естественному порядку в английском языке.
- Обратите внимание, что вместо слова *is* («является») в качестве связки выступает знак равенства (`=`). Это означает строгое равенство — мы ищем строки, значение которых целиком и полностью совпадает с заданным. В этом случае использование знака равенства более чем логично.
- И еще один важный момент: в числе `10000.00` нет ни знака доллара, ни разделителя групп разрядов, таких как запятая, пробел или точка. Хотя мы привыкли к такому формату, база данных хранит просто числа. Компьютеру, на котором работает ваша реляционная СУБД, не важны конкретный тип валюты или удобочитаемость чисел. Если добавить знак доллара, пробел или точку — получите ошибку!

  ВНИМАНИЕ При выборке больших числовых значений, таких как заказы стоимостью свыше 1 миллиона долларов, может возникнуть соблазн поставить в них разделители разрядов, чтобы повысить удобочитаемость данных. В конце концов, легко ошибиться в количестве нулей и напечатать 1000000 как 100000 или 10000000. Но увы: включение в числовые значения любых символов, кроме цифр, вызовет синтаксические ошибки в запросе.

Несмотря на то что с числовыми значениями нельзя задействовать символы валют и разделители групп разрядов, добавление либо исключение дробной части

зачастую не влияет на результат выборки. Это объясняется тем обстоятельством, что числовые значения могут рассматриваться как эквивалентные, даже если они различаются степенью точности. Под *точностью* (precision) понимается количество значащих цифр в числе (например, число 3,14 имеет точность 3 (три значащие цифры), а число 3,1415 — точность 5 (пять значащих цифр)); уровень точности зависит как от способа хранения данных в СУБД, так и от того, каким образом вы формулируете запрос.

Для примера: 1,00 точнее, чем просто 1. Запись 1,00 указывает на точность до сотых долей, тогда как 1 — только до целых чисел. Однако в SQL-запросах, несмотря на разную точность, эти значения считаются эквивалентными: 1,00 равно 1.

Что касается значения столбца `Advance`, задействованного в предыдущем условии фильтра, давайте бегло взглянем на данные, возвращаемые следующим запросом (результаты показаны на рис. 5.1):

```
SELECT
      TitleName,
      Advance
FROM title
WHERE Advance = 10000.00;
```

TitleName	Advance
Anne of Fact Tables	10000.00

Рис. 5.1. Только одна строка соответствует критериям фильтрации, равным 10000.00 для столбца Advance

По степени точности сумма в 10 000,00 доллара точнее в сравнении с суммой в 10 000 долларов, хотя численно обе суммы представляют одно и то же значение. По этой причине мы можем сформулировать запрос без десятичных значений для представления центов, сохраняя при этом результаты, приведенные на рис. 5.1:

```
SELECT TitleName, Advance
FROM title
WHERE Advance = 10000;
```

К практике!

Протестируйте предложение `WHERE` для двух предыдущих запросов и убедитесь, что результаты будут идентичными! Попытайтесь теперь указать еще более точное значение в условии фильтра, скажем, `10000.0000`. Результат должен остаться прежним.

5.1.2. Фильтрация по строковым значениям

До настоящего момента условия фильтрации в наших запросах основывались на числовых критериях, однако фильтрация по нечисловым значениям несколько

отличается. Вместо поиска названия произведения (`TitleName`) для указанного значения аванса (`Advance`) сделаем наоборот: запросим значение аванса для конкретного названия произведения (результаты отображены на рис. 5.2):

```
SELECT Advance
FROM title
WHERE TitleName = 'Anne of Fact Tables';
```

Advance
10000.00

Рис. 5.2. Результат запроса для значения столбца Advance из таблицы title, где значение TitleName равно 'Anne of Fact Tables'

Теперь условие фильтра не является числовым значением, а представляет собой группу слов. Для СУБД — это *строковое значение* (string value), которое в запросе называется *литералом*. При фильтрации по строке необходимо всегда заключать такое значение в одинарные кавычки. Если этого не сделать, запрос приведет к синтаксической ошибке.

ВНИМАНИЕ Не всеми одинарными кавычками можно пользоваться! Чтобы запрос сработал, следует ставить одинарные кавычки, расположенные на той же клавише, что и двойные кавычки. При использовании символа обратной кавычки (или «гравис»), соседствующего с клавишей «1/!» на большинстве клавиатур, вы получите синтаксическую ошибку. Кроме того, при копировании кода из документа, отличного от SQL-сценария, можно вставить неправильно отформатированные одинарные кавычки наподобие тех, что показаны на рис. 5.3.

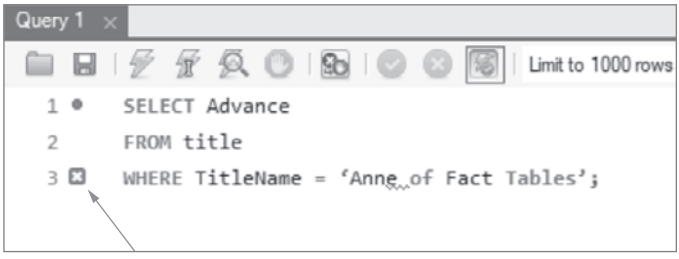


Рис. 5.3. Недопустимые одинарные кавычки, скопированные из документа Microsoft Word. Приложение Workbench сообщает об этом при помощи красного квадрата с крестиком слева от строки кода, содержащей ошибку

Значение строки условия, применяемой в предложении `WHERE`, должно точно совпадать со значением столбца — малейшее отклонение помешает выдаче ожидаемых результатов. Если мы забудем букву *s* в слове `Tables`, как это показано в следующем запросе, то выборка окажется пустой:

```
SELECT Advance
FROM title
WHERE TitleName = 'Anne of Fact Table';
```

НА ЗАМЕТКУ Пропуск символа, скажем, последней буквы *s*, может показаться слишком очевидным примером, но существуют куда менее явные ошибки, связанные со «скрытыми» символами, такими как лишние пробелы, знаки табуляции и символы новой строки, которые нередко приводят к неверным результатам. Хотя СУБД игнорирует такие символы в структуре SQL-запросов, она воспринимает их как дополнительные символы внутри строковых значений.

5.1.3. Фильтрация по значениям даты и времени

Фильтрация по значениям типа *дата* (date values) обладает своей спецификой, поскольку даты представляют собой нечто среднее между числовыми и строковыми значениями. Как и литералы, значения даты и времени в SQL-запросах должны быть заключены в одинарные кавычки. Так, для выборки значения `TitleName`, соответствующего дате публикации (`PublicationDate`) «14 марта 2020 года», применяется следующая SQL-команда:

```
SELECT
      TitleName,
      PublicationDate
FROM title
WHERE PublicationDate = '2020-03-14 00:00:00';
```

Формат по умолчанию — год-месяц-день, затем часы:минуты:секунды. Применение одинарных кавычек здесь может показаться очевидным, поскольку значение содержит нечисловые символы — такие, как тире и двоеточия. Но что, если я скажу вам, что реляционные СУБД на самом деле хранят значения даты и времени в числовом виде? Это правда. Хранить даты в таком формате куда эффективнее, чем в виде строки символов.

Это означает, что правила точности, применимые к числовым значениям, также относятся к значениям даты и времени. Учтите, что `00:00:00` в значении фильтра представляет часы:минуты:секунды, а именно ровно секунду полуночи в начале дня. Если время не указано, эти нули включаются в значение даты по умолчанию, как продемонстрировано в результатах последнего запроса (рис. 5.4).

TitleName	PublicationDate
The Call of the While	2020-03-14 00:00:00

Рис. 5.4. Заглавие (TitleName) и дата публикации (PublicationDate) книги, выпущенной 14 марта 2020 года. Значения времени эквивалентны полуночи в начале указанного календарного дня

Учитывая, что теперь нам известно: 10 000,00 численно эквивалентно 10 000, логично заключить, что 2020-03-14 00:00:00 также равно 2020-03-14. Поскольку значения данных в таблице содержат нули для часов, минут и секунд, мы можем быть уверены, что получим результаты, показанные на рис. 5.4, написав запрос следующего вида без учета времени:

```
SELECT
    TitleName,
    PublicationDate
FROM title
WHERE PublicationDate = '2020-03-14';
```

> **СОВЕТ** По мере вашего продвижения в SQL полезно научиться понимать, какие именно данные содержатся в таблицах, к которым вы собираетесь обращаться. Легко заметить, что все значения даты публикации в таблице title не имеют точности до часов, минут или секунд, и мы можем спокойно формировать запросы, не принимая в расчет значение времени. Но если хотя бы одно значение имеет точность до секунды, целесообразнее составлять все запросы с учетом времени.

5.2. Фильтрация по нескольким условиям

До сих пор мы осуществляли выборку лишь по одному условию, однако в реальных запросах часто требуется применять фильтр сразу по нескольким условиям. Представьте, что вам нужно найти клиента с именем Джефф и фамилией Януччи или заказ с номером 1001 и товаром Product X. Для таких запросов предложение `WHERE` позволяет использовать интуитивно понятный метод фильтрации по нескольким условиям одновременно.

5.2.1. Фильтрация с соблюдением всех условий

Допустим, требуется запросить из таблицы title названия произведений (**`TitleNames`**), для которых аванс (**`Advance`**) составляет 5 тысяч долларов, а авторский гонорар (**`Royalty`**) — 15 %. Сформулируем по-английски простыми словами: “I would like the TitleNames from title where the Advance is 5,000 dollars and the Royalty is 15 percent” («Мне нужны названия книг из таблицы title, где аванс составляет 5000 долларов, а авторский гонорар — 15 %»).

Употребление `AND` («и») для указания дополнительных критериев выборки естественным образом переносится в наш SQL-запрос (результаты представлены на рис. 5.5):

```
SELECT
    TitleName,
    Advance,
    Royalty
FROM title
WHERE Advance = 5000
    AND Royalty = 15;
```

TitleName	Advance	Royalty
Pride and Predicates	5000.00	15.00

Рис. 5.5. Несколько строк в таблице title имеют значение Price, равное 5000.00, но добавление второго условия фильтрации для столбца Royalty в 15.00 (%) сокращает выборку до одной строки

В таком контексте ключевое слово `AND` рассматривается как *логический оператор* (operator) — то есть как служебное слово, выполняющее определенную операцию в SQL. В случае с `AND` эта операция заключается в объединении нескольких условий фильтрации внутри предложения `WHERE`. Ключевое слово `AND` — первый из серии логических операторов, с которыми мы познакомимся.

Применение `AND` позволяет нам добавлять в запрос столько условий фильтрации, сколько необходимо — даже больше, чем в предыдущем примере. Хотя результатом прошлого запроса была всего одна строка, теоретически мы могли бы еще точнее сузить выборку, добавив в предложение `WHERE` еще одно условие фильтра посредством оператора `AND`:

```
SELECT
    TitleName,
    Advance,
    Royalty,
    PublicationDate
FROM title
WHERE Advance = 5000
    AND Royalty = 15
    AND PublicationDate = '2015-04-30';
```

К практике!

Поэкспериментируйте с предложением `WHERE` на примере двух последних запросов — посмотрите, как меняются результаты с применением разных фильтров. Попробуйте сначала отфильтровать по условию, где значение `Royalty` равно `12`; затем добавьте еще одно условие фильтрации, где значение `Advance` равно `6000`. Количество строк в выборке должно сократиться с двух строк в первом запросе до одной строки во втором.

Хотя и не существует строгого ограничения на количество условий фильтра в предложении `WHERE`, важно понимать, что для включения строк в результирующее множество они должны соответствовать каждому из указанных вами условий. Несоответствие любому из условий исключит строки из выборки.

5.2.2. Фильтрация с соблюдением одного из многих условий

Оператор `AND` позволяет фильтровать данные по нескольким условиям, обязательным для всех строк. Однако нередко возникает необходимость применения

нескольких условий фильтра для получения результатов, удовлетворяющих *одному или нескольким* из этих условий.

Допустим, нам нужна любая книга, аванс по которой составляет 5000 долларов или гонорар — 15 %. Простыми словами мы могли бы сформулировать этот запрос так: "I would like the TitleNames from title where the Advance is 5000 dollars or the Royalty is 15 percent" («Мне нужны названия книг из таблицы title, где аванс равен 5000 долларов или авторский гонорар составляет 15 процентов»). Слово *or* («или») заменило здесь *and* — и в SQL оно будет выполнять ту же функцию.

```
SELECT
    TitleName,
    Advance,
    Royalty
FROM title
WHERE Advance = 5000
    OR Royalty = 15;
```

При выполнении запроса мы получим совершенно другие результаты по сравнению с предыдущим, где вместо оператора `OR` применялся оператор `AND`. Теперь мы извлекаем шесть строк вместо одной (рис. 5.6), поскольку строки в результатах должны удовлетворять любому из условий для включения, а не обоим сразу.

TitleName	Advance	Royalty
Pride and Predicates	5000.00	15.00
Catcher in the Try	5000.00	10.00
Anne of Fact Tables	10000.00	15.00
The DateTime Machine	5500.00	15.00
The Call of the While	2500.00	15.00
The Sum Also Rises	5000.00	12.00

Рис. 5.6. Выборка строк по таблице title со значением Advance, равным 5000.00, или значением Royalty, равным 15.00 (%)

Оператор `OR` применяется аналогично оператору `AND` — с возможностью последовательного наращивания количества условий фильтра в соответствии с требованиями запроса. Важно отметить, что каждое дополнительное условие, присоединенное оператором `AND`, потенциально сужает выборку, поскольку критерии отбора становятся все строже. И напротив, каждое условие, добавленное при помощи оператора `OR`, способно расширить результирующий набор, так как «критериев допуска» становится все больше.

TitleName	Advance	Royalty	Price
Pride and Predicates	5000.00	15.00	9.95
The Join Luck Club	6000.00	12.00	9.95
Catcher in the Try	5000.00	10.00	8.95
Anne of Fact Tables	10000.00	15.00	12.95
The DateTime Machine	5500.00	15.00	7.95
The Call of the While	2500.00	15.00	8.95
The Sum Also Rises	5000.00	12.00	7.95

Рис. 5.7. Семь строк выдачи, соответствующих одному из трех условий (значение Advance составляет 5000.00, Royalty — 15.00 (%) либо Price — 9.95)

Мы можем увеличить выборку с шести строк до семи, добавив оператор OR для цены (Price), а заодно и сам столбец Price к результирующему набору — ведь любая строка, включенная в результаты на рис. 5.7, должна удовлетворять лишь одному из трех условий фильтра:

```
SELECT
    TitleName,
    Advance,
    Royalty,
    Price
FROM title
WHERE Advance = 5000
    OR Royalty = 15
    OR Price = 9.95;
```

Можно продолжать наращивать цепочку условий при помощи операторов OR. И всякий раз мы получим столько же строк, сколько и в прошлый раз, или даже больше, поскольку критерии отбора становятся все более объемлющими.

5.2.3. Управление порядком применения нескольких фильтров

В ряде случаев требуется осуществлять выборку данных таким образом, чтобы в предложении WHERE одновременно использовались логические операторы AND и OR. При этом критически важно соблюдать надлежащий порядок вычисления условий.

Предположим, нам необходимо вывести перечень заглавий изданий (TitleNames) со значением цены (Price), составляющим 9 долларов и 95 центов, а также либо с датой выхода в печать (PublicationDate) 6 февраля 2016 года, либо с авансом (Advance) в размере 5 тысяч долларов. Для получения этих данных можно составить запрос следующего вида:

```
SELECT
    TitleName,
    Price,
    PublicationDate,
    Advance
FROM title
WHERE Price = 9.95
    AND PublicationDate = '2016-02-06'
    OR Advance = 5000;
```

Запрос выглядит корректным, однако при его выполнении результаты расходятся с предполагаемой нами логикой. СУБД обрабатывает этот запрос иначе, чем мы задумывали (см. рис. 5.8). Происходит так потому, что база данных независимо от наших намерений отдает приоритет условиям AND перед условиями OR.

TitleName	Price	PublicationDate	Advance
Pride and Predicates	9.95	2015-04-30 00:00:00	5000.00
The Join Luck Club	9.95	2016-02-06 00:00:00	6000.00
Catcher in the Try	8.95	2017-04-03 00:00:00	5000.00
The Sum Also Rises	7.95	2021-11-12 00:00:00	5000.00

Рис. 5.8. Строки, возвращенные запросом, не соответствуют ожидаемым условиям фильтра для книг (TitleNames) с ценой (Price) 9.95 доллара и датой публикации (PublicationDate) 2016-02-06 либо авансом (Advance) в размере 5000 долларов

Сформулируем логическую структуру предполагаемых условий фильтрации. Нам нужно, чтобы выполнялось хотя бы одно из заданных условий:

- цена 9 долларов 95 центов и дата публикации 2016-02-06;
- цена 9 долларов 95 центов и аванс в размере 5000 долларов.

Поскольку реляционная СУБД придает более высокий приоритет условиям `AND`, чем условиям `OR`, она интерпретирует написанные условия фильтрации не так, как мы задумывали. Система воспринимает данный SQL-запрос как команду выбрать строки, удовлетворяющие одному из следующих критериев:

- цена 9 долларов 95 центов и дата публикации 2016-02-06;
- аванс в размере 5000 долларов.

По результатам на рис. 5.8 лишь одна книга соответствует первому условию («The Join Luck Club») и еще три — второму.

Решение этой проблемы может оказаться камнем преткновения для новичков в SQL, однако, как ни парадоксально, оно лежит на поверхности: необходимо задействовать круглые скобки для явного приоритета вашей логики над логикой SQL по умолчанию. Все операции внутри скобок вычисляются до операций за их пределами. Для получения ожидаемых результатов достаточно добавить пару круглых скобок к предыдущему запросу:

```
SELECT
    TitleName,
    Price,
    PublicationDate,
    Advance
FROM title
WHERE Price = 9.95
    AND (PublicationDate = '2016-02-06'
    OR Advance = 5000);
```

Теперь запрос будет вычислять значения в соответствии с нашими ожиданиями, проверяя условие `OR` до условия `AND`. Результат выполнения запроса выглядит в точности так, как отображено на рис. 5.9.

TitleName	Price	PublicationDate	Advance
Pride and Predicates	9.95	2015-04-30 00:00:00	5000.00
The Join Luck Club	9.95	2016-02-06 00:00:00	6000.00

Рис. 5.9. В записи с круглыми скобками запрос возвращает данные, где значение Price равно 9.95 и либо PublicationDate содержит 2016-02-06, либо Advance — 5000.00

> **СОВЕТ** Всякий раз, когда вы пишете SQL-код, задействующий логические операторы `AND` и `OR` в предложении `WHERE`, непременно ставьте круглые скобки для явного управления порядком вычислений. Так и СУБД сможет лучше понять, что от нее требуется, и вашим коллегам не придется ломать голову над вашим сценарием.

5.2.4. Фильтрация в сочетании с сортировкой

При чтении этой главы у вас наверняка возникал вопрос: а куда подевалось предложение `ORDER BY` из главы 4 и как оно согласуется с предложением `WHERE`, которое мы здесь рассматриваем? Все просто: при включении обоих предложений в одну SQL-инструкцию раздел `ORDER BY` должен следовать *после* раздела `WHERE`.

Вернемся к запросу, фигурировавшему ранее в этой главе, который возвращал четыре строки. На рис. 5.8 эти строки представлены в произвольном порядке, однако если необходимо отсортировать результаты по столбцу `TitleName`, просто добавьте предложение `ORDER BY`, как показано ниже (результаты приведены на рис. 5.10):

```
SELECT
    TitleName,
    Price,
    PublicationDate,
    Advance
FROM title
WHERE Price = 9.95
    AND PublicationDate = '2016-02-06'
    OR Advance = 5000
ORDER BY TitleName;
```

TitleName	Price	PublicationDate	Advance
Catcher in the Try	8.95	2017-04-03 00:00:00	5000.00
Pride and Predicates	9.95	2015-04-30 00:00:00	5000.00
The Join Luck Club	9.95	2016-02-06 00:00:00	6000.00
The Sum Also Rises	7.95	2021-11-12 00:00:00	5000.00

Рис. 5.10. Четыре строки, удовлетворяющие одному из двух условий — значение Price, равное 9.95, и PublicationDate, равное 2016-02-06, либо Advance, равное 5000.00, — отсортированные в алфавитном порядке по столбцу TitleName

Сегодня мы рассмотрели массу примеров, демонстрирующих основные разновидности фильтров. Настало время применить ваши новые познания на практике!

5.3. Практическое занятие

1. Нам известно, что если не заключить в нашем условии фильтра нечисловое строковое значение в одинарные кавычки, то это вызовет синтаксическую ошибку. Попробуйте поставить одиночные кавычки вокруг числового значения, такого как `Price` в условии фильтра, и выполнить запрос. Что произойдет?
2. Почему следующий запрос возвращает пустой результат?

```
SELECT
    TitleName,
    Price
FROM Title
WHERE TitleName = 'Anne of Fact Tables ';
```

3. Каков будет результат следующего запроса?

```
SELECT TitleName
FROM Title
ORDER BY TitleName ASC
WHERE Price = 9.95;
```

4. Составьте запрос к таблице `author`, возвращающий строки либо со значением `Check` столбца `PaymentMethod` и значением `Jorge` столбца `FirstName`, либо со значением `Check` столбца `PaymentMethod` и значением `Miller` столбца `LastName`. Включите в результирующий набор столбцы `FirstName`, `LastName` и `PaymentMethod`.

5.4. Ответы

1. Запрос выполняется согласно ожидаемому результату, однако за кулисами СУБД производит сопоставление данных. В такой ситуации системе необходимо либо привести значения в кавычках к числовому типу, либо преобразовать все значения столбца `Price` в строковый формат, что может отрицательно сказаться на длительности запроса при работе с большим объемом данных. По этой причине старайтесь не заключать числовые значения в одинарные кавычки.
2. Запрос не возвращает значений из-за наличия лишнего пробела после слова `Tables` в предложении `WHERE`. Чтобы фильтр работал подобающим образом, строка должна точно совпадать со значениями таблицы, включая невидимые символы вроде пробелов, табуляций и символов новой строки.
3. Запрос вызовет ошибку синтаксиса, поскольку предложение `ORDER BY` не может предшествовать предложению `WHERE`.

4. Данный запрос аналогичен рассмотренному в разделе 5.2.3. Поскольку он включает логические операторы `AND` и `OR`, а также круглые скобки, ожидается, что выборка будет содержать две строки, а сам запрос будет выглядеть следующим образом:

```
SELECT
    FirstName,
    LastName,
    PaymentMethod
FROM author
WHERE PaymentMethod = 'Check'
AND (FirstName = 'Jorge'
    OR LastName = 'Miller');
```

6 Фильтрация с учетом нескольких значений, диапазонов и исключений

Как мы убедились в главе 5, предложение `WHERE` располагает богатейшим арсеналом средств для фильтрации результатов на основе определенных условий. Мы рассмотрели несколько примеров выборки по одному значению с применением логических операторов `AND` и `OR`. Теперь мы разовьем эту концепцию, чтобы производить отбор по еще большему числу значений, включая перечисления конкретных значений и диапазоны значений с неопределенными границами.

Все это примеры выборки данных, удовлетворяющих заданным критериям включения. Однако зачастую требуется обратное — показать все значения, кроме тех, что соответствуют определенным условиям фильтрации. Поэтому мы также рассмотрим, как исключать данные из уже изученных нами условий. А начнем мы с нового оператора для предложения `WHERE`.

6.1. Фильтрация по конкретным значениям

Ранее мы рассмотрели элементарные примеры поискового отбора, такие как извлечение названий (`TitleNames`) для изданий с фиксированным значением цены (`Price`). Если, к примеру, мы захотим найти книги, стоимость которых составляет 10 долларов 95 центов, то наш SQL-запрос будет таким:

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price = 10.95;
```

Но что, если мы захотим выбрать книги ценой 10 долларов 95 центов или 12 долларов 95 центов? Теперь мы знаем, что для этого можно воспользоваться оператором `OR`, и потому напишем следующий SQL-запрос (результат которого отображен на рис. 6.1):

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price = 10.95
  OR Price = 12.95;
```

TitleName	Price
Anne of Fact Tables	12.95
The Great GroupBy	10.95

Рис. 6.1. Результаты запроса с условиями фильтра для столбца Price, равными 10.95 или 12.95

Приведенный запрос вернет нам ожидаемые результаты для двух условий выборки, но может привести к громоздкому SQL-коду, если список таких условий значительно возрастет. А если их станет десять, двадцать? Повторять `OR` снова и снова — все равно что копать ложкой траншею: технически возможно, но мучительно и бессмысленно. Особенно когда речь идет об одном-единственном столбце.

К счастью, в SQL предусмотрен оператор `IN`, позволяющий объединить несколько условий фильтра в одно. Оператор `IN` имеет три обязательных компонента:

- список значений должен разделяться запятыми;
- список значений следует заключить в круглые скобки;
- правила применения одинарных кавычек те же, что и для всех условий фильтра.

В качестве примера перепишем предыдущий запрос, прибегнув к оператору `IN`:

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price IN (10.95, 12.95);
```

Выполнение запроса дает те же результаты, что представлены на рис. 6.1, при более компактной форме SQL-кода: все короче, чище, элегантнее. Я уже упоминал, что использование одинарных кавычек здесь аналогично тому, что мы применяли в запросах для фильтрации строк и дат.

К практике!

При помощи оператора `IN` составьте запрос, выводящий строки со столбцами `TitleName` и `Price` таблицы `title`, в которых значение `Price` равно 7.95, 8.95 или 9.95 доллара.

При поиске книг по конкретной дате публикации важно помнить о правилах применения в фильтрах одинарных кавычек, коль скоро мы имеем дело с датами, а не числами. При этом наш SQL-запрос выглядел бы так (результаты отображены на рис. 6.2):

```
SELECT
    TitleName,
    PublicationDate
FROM title
WHERE PublicationDate IN ('2015-04-30', '2016-02-06');
```

TitleName	PublicationDate
Pride and Predicates	2015-04-30 00:00:00
The Join Luck Club	2016-02-06 00:00:00

Рис. 6.2. Выборка заглавий со значением PublicationDate, равным 2015-04-30 или 2016-02-06

НА ЗАМЕТКУ Для реляционной СУБД порядок указания значений в операторе `IN` не важен. Как и в любом запросе без предложения `ORDER BY`, в результирующем наборе не гарантируется возврат строк в каком-либо определенном порядке. Тем не менее, как я уже неоднократно отмечал в этой книге, следует придерживаться передового опыта и делать SQL-код максимально удобочитаемым. Поэтому значения в условиях фильтра оператора `IN` лучше перечислять в числовом, алфавитном или хронологическом порядке.

6.2. Фильтрация по диапазону значений

Применение оператора `IN` для фильтрации по перечню заданных значений оправданно, лишь когда вам точно известны все значения, которые требуется найти. В противном случае он совершенно бесполезен. При этом в реальной практике зачастую возникает необходимость выборки данных, принадлежащих к определенному интервалу или диапазону — скажем, все значения больше указанной суммы или предшествуют установленной дате. Для этого в SQL применяются *операторы сравнения* (comparison operators), которые сопоставляют два значения, чтобы определить, удовлетворяют ли они заданным критериям, и таким образом получить нужные результаты.

6.2.1. Фильтрация по открытому диапазону

Если вы хоть раз держали в руках учебник математики, вам наверняка знакомы знаки «*меньше*» (<) и «*больше*» (>). (На стандартной клавиатуре с латинской раскладкой они расположены на тех же клавишах, что и запятая с точкой соответственно.) Как и знак равенства (=), эти символы являются широко распространенными операторами сравнения. В отличие от знака равенства, знаки «меньше» и «больше» позволяют задавать полуоткрытые или неограниченные интервалы значений, что делает их особенно полезными при фильтрации по диапазонам.

Ранее в этой главе мы извлекали все названия книг со значением `Price`, равным 10.95 и 12.95 доллара. В таблице `title` это самые дорогие книги. Мы можем вывести те же два заглавия, применив в запросе знак «больше» (>). Добавим также предложение `ORDER BY`, чтобы отобразить цены в упорядоченном виде — напомним, что по умолчанию сортировка выполняется по возрастанию. Выборка по этому запросу представлена на рис. 6.3:

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price > 9.95
ORDER BY Price ASC;
```

TitleName	Price
The Great GroupBy	10.95
Anne of Fact Tables	12.95

Рис. 6.3. Названия книг со значением Price больше 9.95, отсортированные по столбцу Price по возрастанию

Можно и наоборот — с помощью знака «меньше» (<) найти книги дешевле 9 долларов 95 центов. Для такого запроса мы отсортируем значения столбца `Price` по убыванию, чтобы скомпоновать цены, наиболее близкие к 9,95 доллара, в начале выборки (рис. 6.4):

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price < 9.95
ORDER BY Price DESC;
```

TitleName	Price
Catcher in the Try	8.95
The Call of the While	8.95
The DateTime Machine	7.95
The Sum Also Rises	7.95

Рис. 6.4. Названия книг со значением Price меньше 9.95, отсортированные по столбцу Price по убыванию

Вы, вероятно, заметили, что ни один из результирующих наборов не включает в себя книги с ценой ровно 9 долларов 95 центов. Безусловно, иногда требуется запросить диапазон значений, включая и те, что равны условию фильтра, поэтому в SQL также присутствуют операторы сравнения «*меньше или равно*» (`<=`) и «*больше или равно*» (`>=`). Посредством одного из них мы можем запросить любую книгу ценой, превышающей или равной условию фильтра `9.95`, и получить результаты, отображенные на рис. 6.5:

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price >= 9.95
ORDER BY Price ASC;
```

TitleName	Price
Pride and Predicates	9.95
The Join Luck Club	9.95
The Great GroupBy	10.95
Anne of Fact Tables	12.95

Рис. 6.5. Названия книг со значением Price больше или равным 9.95, отсортированные по столбцу Price по возрастанию

НА ЗАМЕТКУ Теперь мы знаем, каким образом можно извлекать значения больше или меньше заданного числа или даты. Впрочем, операторы эти применимы и для строк — хотя такие случаи встречаются реже. Значения при этом, как правило, выводятся в соответствии с алфавитным порядком. Однако будьте осторожны при использовании этих операторов со строками: способ фильтрации чисел, символов и регистра зависит от *параметров сортировки* (collation) вашей СУБД. Параметры сортировки определяют правила сортировки на основе свойств символов — таких, как набор символов, регистр букв и наличие диакритики; поэтому разные правила сортировки могут устанавливать для символов различный приоритет.

6.2.2. Фильтрация по закрытому диапазону

Операторы, описанные в разделе 6.2.1, являются открытыми, то есть они могут включать бесконечное множество значений больше или меньше заданного. Если же необходимо найти значения, находящиеся в промежутке между двумя установленными пределами, то SQL предоставляет два варианта решения этой задачи.

Вы уже сами могли догадаться о первом варианте — применении `>` и `<` в предложении `WHERE`. Чтобы выбрать названия произведений с ценой между 8,95 и 10,95 доллара, можно написать SQL-запрос следующего вида. Как и прежде, в запросе задействован оператор `ORDER BY` для компоновки результатов таким образом, чтобы на рис. 6.6 акцентировать максимальные и минимальные значения, полученные благодаря условиям фильтра:

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price > 8.95
    AND Price < 10.95
ORDER BY Price ASC;
```

TitleName	Price
Pride and Predicates	9.95
The Join Luck Club	9.95

Рис. 6.6. Названия книг со значениями Price в пределах 8.95 и 10.95

Очевидно, что единственное значение `Price`, соответствующее условиям нашего фильтра, равно 9.95. Чтобы изменить условия выборки для *включения* в нее пороговых значений, необходимо задействовать операторы `>=` и `<=` в предложении `WHERE` следующим образом:

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price >= 8.95
    AND Price <= 10.95
ORDER BY Price ASC;
```

Как показано на рис. 6.7, теперь результирующий набор включает заглавия книг, значения `Price` которых применены в качестве условий фильтрации.

TitleName	Price
Catcher in the Try	8.95
The Call of the While	8.95
Pride and Predicates	9.95
The Join Luck Club	9.95
The Great GroupBy	10.95

Рис. 6.7. Названия книг со значениями Price в пределах 8.95 и 10.95, пороговые значения включены в условия фильтра

Существует и другой способ поиска по диапазону значений: вместо `>=` и `<=` для выполнения той же функции можно воспользоваться ключевым словом `BETWEEN`. Правила применения `BETWEEN` следующие:

- столбец, по которому осуществляется поиск, упоминается лишь раз;
- поддерживаются только два условия фильтрации;
- в первом значении указывается нижний порог диапазона, а во втором — верхний;
- условия фильтра должны разделяться словом `AND`;
- любые значения, совпадающие с пороговыми, включаются в результаты выборки.

Используя оператор BETWEEN, перепишем предыдущий запрос:

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price BETWEEN 8.95 AND 10.95
ORDER BY Price ASC;
```

Хотя SQL-код и стал на строку короче, результат остался таким же, как на рис. 6.7.

> **ВНИМАНИЕ** Следует еще раз напомнить: при использовании BETWEEN первое значение всегда обозначает нижнюю границу диапазона, а второе — верхнюю. Если в фильтре BETWEEN первое значение окажется больше второго, запрос не вернет результатов. Так, SQL-запрос WHERE Price BETWEEN 10.95 AND 8.95 не вернет ни одной строки, вне зависимости от прочих условий выборки. Логически невозможно, чтобы значение одновременно было больше 10.95 и меньше 8.95.

6.3. Исключение из выборки

До сих пор мы пользовались предложением WHERE для отбора строк, удовлетворяющих заданным условиям — конкретным значениям или диапазонам данных. Теперь рассмотрим противоположную задачу: как исключить из результата строки, которые соответствуют определенным критериям, оставив все остальные. Для этого используются условия с отрицанием.

6.3.1. Исключение конкретного значения

В главе 5 мы узнали, как осуществлять выборку по конкретным значениям при помощи знака равенства (=). Теперь посмотрим, как делать обратное — исключать значения, используя знак *неравенства* (<>), объединивший в себе знаки < и >.

Чтобы выбрать все книги, цена которых не равна 7 долларам 95 центам, можно следующим образом задействовать знак <>, упорядочив при этом результаты, как показано на рис. 6.8:

TitleName	Price
Catcher in the Try	8.95
The Call of the While	8.95
Pride and Predicates	9.95
The Join Luck Club	9.95
The Great GroupBy	10.95
Anne of Fact Tables	12.95

Рис. 6.8. Все заглавия, кроме двух — со значением Price, равным 7.95

```
SELECT
    TitleName,
    Price
FROM title
WHERE Price <> 7.95
ORDER BY Price ASC;
```

Оператор <> — не только для чисел. Он прекрасно справляется и с датами, и со строками. Если необходимо извлечь все названия, кроме тех, что были опубликованы 6 февраля 2016 года, можно воспользоваться <> в сочетании с одинарными кавычками — результат отображен на рис. 6.9:

TitleName	PublicationDate
Pride and Predicates	2015-04-30 00:00:00
Catcher in the Try	2017-04-03 00:00:00
Anne of Fact Tables	2018-01-12 00:00:00
The DateTime Machine	2019-02-04 00:00:00
The Great GroupBy	2019-12-23 00:00:00
The Call of the While	2020-03-14 00:00:00
The Sum Also Rises	2021-11-12 00:00:00

Рис. 6.9. Все заглавия, кроме одного («The Join Luck Club»), со значением PublicationDate, равным 2016-02-06

```
SELECT
    TitleName,
    PublicationDate
FROM title
WHERE PublicationDate <> '2016-02-06'
ORDER BY PublicationDate ASC;
```

НА ЗАМЕТКУ Некоторые реляционные СУБД позволяют применять оператор != вместо <>. Оба варианта выполняют одну и ту же функцию — отрицают отдельное условие. Но не спешите привыкать к операторам с восклицательным знаком, ибо не все СУБД их поддерживают. Чтобы ваш код был универсальным и не подводил в неожиданный момент, лучше выработать привычку ставить везде <> — его понимает любая база данных.

6.3.2. Исключение всего условия фильтрации

Оператор <> хорош, когда нужно исключить одно конкретное значение в условии фильтра. В то же время существует логический оператор, способный исключить все условие целиком. Оператор — NOT — меняет логику условия на противоположную: вместо «выбрать где...» получается «выбрать где НЕ...».

Так, в предыдущем запросе мы прибегли к <> для исключения всех названий с датой публикации 6 февраля 2016 года. Того же результата можно достичь, применив оператор NOT к противоположному условию — то есть к оператору =. Следующий запрос возвращает результаты, приведенные на рис. 6.9, с сортировкой по дате публикации:

```
SELECT
    TitleName,
    PublicationDate
FROM title
WHERE NOT PublicationDate = '2016-02-06'
ORDER BY PublicationDate ASC;
```

В представленном случае оператор `NOT` следует сразу после ключевого слова `WHERE`. Впрочем, его можно поставить и после начала любого фильтрующего условия для его отрицания. Это означает, что оператор `NOT` также можно применять сразу после условий, которые начинаются с операторов `AND` или `OR`.

К практике!

Выполните два предыдущих запроса и сравните способы применения `<>` и `NOT` для исключения конкретных значений.

Не следует сочетать `NOT` с операторами `>` или `<`, поскольку логически эквивалентный результат можно получить с помощью их положительных противоположностей — `<=` и `>=` соответственно, — что обеспечивает более простой и понятный синтаксис. В то же время использование `NOT` в связке с оператором `IN` (рассмотренным в разделе 6.1) является широко распространенной и вполне корректной практикой.

Помните, как ранее мы формулировали единое условие с оператором `IN`, чтобы выбрать все названия книг ценой 10.95 или 12.95 доллара, заменив им два условия с оператором `OR`? Теперь мы можем применить `NOT` для инверсии этого условия, чтобы вывести все издания, *кроме* тех, у которых значение `Price` равно 10.95 или 12.95. Результаты отсортированы по столбцу `Price` (рис. 6.10):

```
SELECT
    TitleName,
    Price
FROM title
WHERE NOT Price IN (10.95, 12.95)
ORDER BY Price ASC;
```

TitleName	Price
The DateTime Machine	7.95
The Sum Also Rises	7.95
Catcher in the Try	8.95
The Call of the While	8.95
Pride and Predicates	9.95
The Join Luck Club	9.95

Рис. 6.10. Все заглавия, кроме тех, чье значение Price равно 10.95 либо 12.95

Возможно, вы обратили внимание на условие `WHERE NOT Price IN` в запросе выше — оно звучит немного неуклюже для носителей английского языка. В SQL существует более естественный и распространенный способ получить тот же результат, а именно отдельный оператор `NOT IN`, благодаря которому можно поместить `NOT` непосредственно перед `IN` в условии фильтрации, чтобы сделать запрос более удобочитаемым, — вот так:

```
SELECT TitleName, Price
FROM title
WHERE Price NOT IN (10.95, 12.95)
ORDER BY Price ASC;
```

Этот запрос также возвращает результаты, показанные на рис. 6.10. Когда требуется применить исключающий список значений, как в предыдущем примере, предпочтительной является конструкция `NOT IN`.

6.4. Сочетание различных типов условий фильтрации

В этой главе мы познакомились с рядом новых способов фильтрации данных — с применением как включающих, так и исключающих условий. Один важный момент, на который стоит обратить внимание: в своих запросах вы можете (и будете) сочетать оба типа условий. Скажем, можно составить запрос, который находит все названия книг с авансом (`Advance`) больше 5000, но при этом с авторским гонораром (`Royalty`), не равным 12 %. Результат такого запроса проиллюстрирован на рис. 6.11:

```
SELECT
    TitleName,
    Advance,
    Royalty
FROM title
WHERE Advance > 5000
     AND Royalty <> 12;
```

TitleName	Advance	Royalty
Anne of Fact Tables	10000.00	15.00
The DateTime Machine	5500.00	15.00

Рис. 6.11. Заглавия со значением Advance свыше 5000 и Royalty, не равным 12 (%)

Владея арсеналом операторов из этой и предыдущей глав, уже можно составлять относительно сложные условия фильтра. Если необходимо включить в выборку как строки из предыдущего примера (все названия с `Advance > 5000` и `Royalty <> 12` [%]), так и все названия, опубликованные после 1 января 2020 года, это легко сделать, сочетая SQL-конструкции для включающих и исключающих

условий и управляя логикой посредством круглых скобок. При таком подходе ваш SQL-запрос может выглядеть примерно так (а его результаты — как показано на рис. 6.12):

```
SELECT
    TitleName,
    Advance,
    Royalty,
    PublicationDate
FROM title
WHERE (Advance > 5000
    AND Royalty <> 12)
     OR (PublicationDate > '2020-01-01');
```

TitleName	Advance	Royalty	PublicationDate
Anne of Fact Tables	10000.00	15.00	2018-01-12 00:00:00
The DateTime Machine	5500.00	15.00	2019-02-04 00:00:00
The Call of the While	2500.00	15.00	2020-03-14 00:00:00
The Sum Also Rises	5000.00	12.00	2021-11-12 00:00:00

Рис. 6.12. Названия книг со значением Advance свыше 5000 и Royalty, не равным 12 (%), либо книги, опубликованные после 1 января 2020 года

> **СОВЕТ** Хотя включающие и исключающие условия фильтрации могут давать одинаковые результаты, предпочтительнее использовать включающие условия везде, где возможно. Они легче читаются и обычно обрабатываются СУБД эффективнее. Исключение — когда нужно отсечь всего несколько значений из большого набора: тогда проще написать, например, NOT IN (1, 2, 3), чем перечислять десятки оставшихся.

Прочитав всего несколько глав этой книги, вы уже научились осуществлять осмысленную и точную выборку данных. До сих пор мы в основном применяли условия фильтрации к числам и датам. А вот следующая, седьмая глава откроет вам новый набор инструментов предложения `WHERE` для выполнения расширенного поиска данных в строковых значениях.

6.5. Сводный обзор операторов сравнения

В этой главе было рассмотрено свыше десятка операторов сравнения, применимых для фильтрации в предложении `WHERE`. Если вы вдруг не успели все законспектировать (хотя я уверен, что вы старательно это делали!), взгляните на табл. 6.1 — в ней сведена вся палитра изученных нами операторов.

Таблица 6.1. Обзор операторов сравнения в предложении WHERE

Оператор	Описание
=	Равенство
<>	Неравенство
!=	Неравенство*
<	Меньше
>	Больше
!<	Не меньше*
!>	Не больше*
<=	Меньше или равно
>=	Больше или равно
BETWEEN	Между двух значений, включая эти значения
IN	Равенство списку из нескольких значений
NOT IN	Неравенство списку из нескольких значений
NOT	Неравенство заданному условию

* Поддерживается не всеми СУБД.

6.6. Практическое занятие

Сегодня вам предстоит выполнить лишь одно задание, но это настоящий вызов для вашей творческой натуры. В главах 5 и 6 было рассмотрено немало способов включения и исключения данных, и сейчас самое время вспомнить все, что вы узнали о применении фильтров в предложении `WHERE`.

Ваша задача: используя предложение `WHERE`, выявить максимальное количество способов для извлечения из таблицы `title` заглавий книг (`TitleName`) и их цен (`Price`) для всех строк со значением столбца `Price`, не равным 9.95. Каждый ваш запрос должен выдать точно такой же результат, что изображен на рис. 6.13, отсортированный по возрастанию цены.

TitleName	Price
The DateTime Machine	7.95
The Sum Also Rises	7.95
Catcher in the Try	8.95
The Call of the While	8.95
The Great GroupBy	10.95
Anne of Fact Tables	12.95

Рис. 6.13. Названия книг с ценой, не равной 9.95, отсортированные по возрастанию цены

6.7. Ответы

Вот лишь некоторые из множества вариантов предложения `WHERE`, позволяющих исключить из выборки названия произведений стоимостью 9.95 доллара:

- `WHERE Price <> 9.95`
- `WHERE NOT Price = 9.95`
- `WHERE Price < 9.95 OR Price > 9.95`
- `WHERE PRICE NOT IN (9.95)`
- `WHERE Price IN (7.95, 8.95, 10.95, 12.95)`
- `WHERE Price BETWEEN 7.95 AND 8.95 OR Price BETWEEN 10.95 and 12.95`
- `WHERE NOT Price BETWEEN 9.95 and 9.95`

Последний способ может показаться несколько экстравагантным, поскольку он, по сути дела, исключает диапазон значений, состоящий из одного-единственного значения. Я привожу его здесь лишь для того, чтобы показать, что диапазон с совпадающими верхней и нижней границами является корректным с точки зрения синтаксиса и может быть выполнен.

7 Выборка по подстановочным знакам и NULL-значениям

Предыдущие главы изобиловали разнообразными приемами фильтрации данных в запросах при помощи целого арсенала операторов сравнения, позволяющих задавать условия равенства или неравенства для одного либо нескольких значений или диапазонов. Посвятим еще одну главу изучению ряда примечательных способов поиска по менее точным совпадениям.

В частности, речь пойдет о выборке данных при отсутствии точного значения для поиска. Вместо поиска конкретных значений мы задействуем сопоставление с *шаблонами значений*. Такой подход невероятно удобен, скажем, при составлении перечня товаров, в названии которых встречаются слова *«томат»* или *«кабель»*, или при выводе списка всех покупателей, чьи фамилии начинаются с буквы *A*.

А еще мы исследуем наиболее сложную категорию значений для поиска: значения null («пустые», или «отсутствующие», значения). NULL-значения часто понимаются ошибочно, что приводит к получению некорректных результатов. Мы выясним, что же скрывается за этим понятием (а чего в нем нет) и как правильно формировать запросы для таких значений.

7.1. Фильтрация при помощи подстановочных знаков

В главе 6 вы ознакомились с методами поиска значений в диапазонах чисел или дат. Даже если неизвестны все конкретные значения в пределах заданного диапазона, вы умеете формировать корректные запросы при помощи операторов `>`, `<` и `BETWEEN`.

Примечательно, что те же самые операторы можно применять и для поиска строковых значений. Так, к примеру, если нам необходимо установить все имена и фамилии авторов, фамилия которых начинается с буквы *S*, мы можем написать

SQL-запрос, задействовав >= и <, чтобы получить выборку, отображенную на рис. 7.1:

```
SELECT
    FirstName,
    LastName
FROM author
WHERE LastName >= 'S'
    AND LastName < 'T';
```

FirstName	LastName
Jen	Strong
Gail	Shawn

Рис. 7.1. Выборка из таблицы author по диапазону фамилий, начинающихся с S

НА ЗАМЕТКУ Отныне мы не будем сортировать результаты, если в этом нет необходимости. Мы уже говорили в главе 4, что сортировка данных при помощи `ORDER BY` повышает вычислительную нагрузку, связанную с обработкой запроса, и поэтому по возможности ее следует избегать. Просто помните, что без `ORDER BY` результаты одного и того же запроса могут всякий раз выводиться в разном порядке.

Этот метод поиска строковых значений в диапазоне работает в большинстве случаев, но не всегда. Как я упоминал вкратце раньше, в зависимости от параметров схемы сопоставления (collation), регистра символов (верхнего или нижнего) и применяемого набора символов (таких, как буквы с тильдами или умлаутами) вы можете получить нестабильные результаты при использовании такого метода для фильтрации строковых значений.

К тому же подобная форма запроса выглядит неестественно, и в устной речи пользователь вряд ли стал бы формулировать условие фильтрации подобным образом. Нам нужен список фамилий, начинающихся с буквы *S*, а не список имен, принадлежащих диапазону от *S* до *T*. И вот тут на помощь приходят подстановочные знаки.

Подстановочный знак (wildcard) — это специальный символ, предназначенный для замены произвольной последовательности символов в строковом значении. Он позволяет осуществлять поиск значения по определенным шаблонам, а не ограничиваться диапазонами, как в предыдущем запросе.

7.1.1. Фильтрация при помощи знака процента

Первый подстановочный знак, который мы рассмотрим, — это `%` (знак процента). Выступая в качестве подстановочного знака, `%` соответствует любой строке, включая пустую строку без символов. Вот как можно применить его для поиска авторов с фамилией, начинающейся на *S*:

```
SELECT
    FirstName,
    LastName
FROM author
WHERE LastName LIKE 'S%';
```

Обратите внимание, что здесь мы задействуем новый оператор сравнения — `LIKE`. Именно `LIKE` всегда применяется при поиске с подстановочными знаками, поскольку в языке SQL он указывает, что мы ищем данные по шаблону, а не по точным значениям условия. Попытка заменить `LIKE` на любой другой оператор сравнения (скажем, `=` или `>`) приведет к пустому результирующему набору.

К практике!

Выполните предыдущий запрос, а затем попытайтесь поставить оператор `=` вместо `LIKE`.

И пускай мы уже знаем, как выглядит окончательный запрос, давайте на секунду представим, как бы мы объяснили его другу за чашкой кофе: "I would like the first name and last name from the author table where the last names start with *S*" («Мне нужны имена и фамилии из таблицы author, начинающиеся на *S*»).

Увы, в SQL нет оператора STARTS WITH («начинается с»). И хотя он мог бы пригодиться для нашего запроса, он был бы крайне ограничен в употреблении. Вдобавок нам пришлось бы придумывать и другие операторы — например, ENDS WITH («оканчивается на») или даже HAS IN THE MIDDLE («содержит в середине»). Такие операторы были бы не только излишне многословны, но и попросту нелепы.

В SQL символ `%` не только короче, но и функционально заменяет все эти вымышленные операторы, вместе взятые. Самый простой способ запомнить, как применять оператор `%`, — произносить его про себя как *что угодно* (something). Это *что угодно* может быть нулем символов, одним символом или даже сотней. Тогда весь наш запрос прозвучит так: "I would like the first name and last name from the author table where the last names are like S and then *something*" («Мне нужны имя и фамилия из таблицы author, где фамилии — это *S*, а затем *что угодно*»).

По смыслу фраза эта довольно близка тому, как выглядит наш новый запрос, и результаты будут аналогичными представленным на рис. 7.1. Чуть выше мы уже отметили, что оператор `%` можно поставить в любом месте строки. Это значит, что если мы хотим найти все фамилии, оканчивающиеся на *N*, то можем сформулировать наш запрос так: "I would like the first name and last name from the author table where the last names are like *something* and then *N*" («Мне нужны имя и фамилия из таблицы author, где фамилии — это *что угодно*, а затем *N*»).

Нетрудно догадаться, что наш SQL-запрос будет очень похож на его словесное описание. Вот он, с результатами, приведенными на рис. 7.2:

```
SELECT
    FirstName,
    LastName
FROM author
WHERE LastName LIKE '%N';
```

FirstName	LastName
Robert	Davidson
Gail	Shawn
Andy	Melkin
Deepthi	Mahadevan

Рис. 7.2. Авторы, чья фамилия заканчивается на *N*

Допустим, необходимо сузить круг поиска и выбрать лишь те фамилии, которые не просто оканчиваются на *N*, но и начинаются с *M*, что также нетрудно реализовать — результаты отображены на рис. 7.3:

```
SELECT
    FirstName,
    LastName
FROM author
WHERE LastName LIKE 'M%N';
```

FirstName	LastName
Andy	Melkin
Deepthi	Mahadevan

Рис. 7.3. Авторы, чья фамилия начинается на *M* и заканчивается на *N*

Оператор `%` ставится и в начале, и в конце искомой комбинации символов — это позволяет находить совпадения с шаблоном в любой части данных. К примеру, так можно найти всех авторов, в чьих фамилиях встречается последовательность символов «*de*». Выборка по такому запросу показана на рис. 7.4:

```
SELECT
    FirstName,
    LastName
FROM author
WHERE LastName LIKE '%DE%';
```

FirstName	LastName
Buck	Fernandez
Deepthi	Mahadevan

Рис. 7.4. Авторы, в чьей фамилии присутствует подстрока DE

Этот прием может пригодиться при поиске по столбцу с комментариями, примечаниями и прочими заметками в свободном стиле. Скажем, если требуется

вывести все комментарии, содержащие слово *good*, следует искать значения столбца, соответствующие шаблону `LIKE '%good%'`. Большинство систем управления реляционными базами данных не чувствительны к регистру, поэтому в результатах вы получите и «Good», и «GOOD». Вместе с тем в выборку могут попасть и такие строки, как «not very good» или «goodbye», поскольку они тоже совпадают с заданным шаблоном.

ВНИМАНИЕ По умолчанию в параметрах схемы сопоставления MySQL, Microsoft SQL Server и SQLite оператор `LIKE` не является регистрозависимым, тогда как в PostgreSQL и Oracle он может быть по умолчанию чувствительным к регистру.

Как бы ни был полезен подстановочный знак `%` для поиска шаблонов символов, ему не хватает точности. Если требуется найти значения в определенной позиции строки, на помощь приходит другой подстановочный знак.

7.1.2. Фильтрация с помощью символа подчеркивания

В отличие от оператора `%`, который совпадает с любой последовательностью символов (включая строки нулевой длины), оператор `_` (*нижнее подчеркивание*) соответствует строго одному произвольному символу. Более того, существует возможность сочетания подстановочных знаков `_` и `%` в рамках единого поискового шаблона.

ВНИМАНИЕ Оператор `_` не поддерживается в СУБД DB2.

Так, если требуется найти имя и фамилию любого автора, имя которого начинается на *R* и имеет букву *b* в качестве третьей буквы, как показано на рис. 7.5, можно задействовать комбинацию `_` и `%`:

```
SELECT
    FirstName,
    LastName
FROM author
WHERE FirstName LIKE 'R_b%';
```

FirstName	LastName
Robert	Davidson
Rebecca	Miller

Рис. 7.5. Авторы, чья фамилия начинается на *R*, а третьей буквой является *b*

Хотя оператор `_` используется реже, чем `%`, могут возникать ситуации, когда требуется искать шаблоны с неопределенным символом в конкретной позиции — в частности, если нужно найти элементы со значением цвета «gray», допускающим написание *gray* или *grey*. Вместо перечисления всех вариантов можно сформулировать следующее условие: `WHERE color LIKE 'gr_y'`.

Кроме того, с помощью подстановочного знака _ можно искать значения, в которых различаются первые несколько символов. Вот пример: поиск всех авторов, у которых третий символ имени — *u*. Результаты запроса приведены на рис. 7.6:

```
SELECT
    FirstName,
    LastName
FROM author
WHERE FirstName LIKE '__u%'
ORDER BY FirstName ASC;
```

FirstName	LastName
Doug	Li
Paul	Tripp

Рис. 7.6. Авторы, в имени которых третья буква *u*

Помимо % и _, каждая СУБД поддерживает и другие подстановочные знаки, но поскольку они различаются от системы к системе, я не буду рассматривать их в этом руководстве. Тем не менее, если вы работаете с конкретной СУБД, я рекомендую изучить документацию — возможно, она поддерживает дополнительные операторы, которые расширят ваши возможности поиска по шаблонам.

А теперь перейдем к поиску... того, чего нет.

7.2. Фильтрация по значениям null

При проектировании реляционных СУБД значения некоторых столбцов могут быть объявлены как требующие обязательного заполнения, тогда как для других столбцов допускается отсутствие данных. Если в строке для такого столбца данные не заданы, в нем отображается NULL.

Как я упоминал в начале главы, среди понятий, связанных с базами данных, значения null являются рекордсменом в части неверного истолкования. Выражаясь самым простым языком, NULL-значения — это в буквальном смысле «ничто»: они обозначают отсутствие данных. Казалось бы, в чем тут сложность? Однако, коль скоро NULL не являются типичными значениями вроде 30, ‘Аризона’ или ‘2012-05-12’, при выполнении запросов к данным их необходимо обрабатывать особым образом.

Заглянем в таблицу author и проанализируем ее содержимое, выбрав все 11 строк по всем столбцам. Сразу бросается в глаза столбец `MiddleName` — в нем довольно много значений NULL. Не у каждого человека есть второе имя, поэтому его отсутствие у автора обозначается как NULL. MySQL Workbench подчеркивает этот факт визуально: NULL отображается иначе, чем остальные значения, — уменьшенным шрифтом и белым текстом на темном фоне (рис. 7.7).

```
SELECT *
FROM author;
```

AuthorID	FirstName	MiddleName	LastName	PaymentMethod
1	Paul	K	Tripp	Cash
2	Doug	NULL	Li	Check
3	Jen	NULL	Strong	Check
4	Jorge	Armando	Guerra	Check
5	Robert	Grant	Davidson	Check
6	Gail	Anne	Shawn	Check
7	Rebecca	NULL	Miller	Check
8	Andy	NULL	Melkin	Direct Deposit
9	Buck	NULL	Fernandez	Cash
10	Chris	NULL	Walenski	Direct Deposit
11	Deepthi	NULL	Mahadevan	Direct Deposit

Рис. 7.7. Выборка всех столбцов таблицы author, включая значения null в столбце MiddleName

7.2.1. Как не следует выполнять поиск NULL-значений

Подчеркну еще раз: NULL означает отсутствие данных, что создает определенные сложности при составлении запросов к столбцам, содержащим такие значения. Чтобы избежать типичных ошибок, рассмотрим сначала, как нельзя искать значения null. Так, если мы захотим выбрать строки в таблице `author`, в которых `MiddleName` содержит NULL, ни один из следующих трех запросов не даст желаемого результата:

```
/* Этот код не сработает, поскольку null не являются пустыми строками. */
 SELECT *
 FROM author
 WHERE MiddleName = '';
```

Этот запрос не вернет NULL-значения, так как он ищет строку нулевой длины — так называемую *пустую* строку. Понимаю, это может сбивать с толку: когда я говорю «отсутствие значения» и «пустая строка», кажется, будто речь идет об одном и том же. Но пустая строка отличается от NULL, поскольку она все же является строкой. Иными словами, «под капотом» СУБД все равно резервирует байты для хранения пустой строки, и потому к ней применимы условия фильтрации с различными операторами сравнения, включая поиск по подстановочным знакам. NULL-значения не занимают байтов и обрабатываются обычными операторами сравнения или шаблонами. Вот еще один распространенный, но ошибочный способ поиска значений null:

```
/* Этот код не сработает, поскольку значения null не являются словом null. */
SELECT *
```

```
FROM author
WHERE MiddleName = 'NULL';
```

Этот запрос не действует, так как он ищет не NULL, а слово «NULL» — то есть четыре буквы: N-U-L-L. Он не извлечет ни одной строки, если только в ваших данных явно не содержится текст из четырех символов — слова «NULL». Вы не поверите, но иногда разработчики баз данных, не в полной мере разобравшись с концепцией NULL-значений, пытаются использовать само слово «NULL», видя в нем выразитель значения null. И поскольку «NULL» — это обычная строка, подобного рода практика создает массу проблем при выполнении запросов. Прошу вас, ни в коем случае так не делайте.

И напоследок, еще один неправильный способ поиска NULL-значений:

```
/* Этот код не сработает, поскольку ни одно значение не равно null. */
SELECT *
FROM author
WHERE MiddleName = NULL;
```

На первый взгляд данный запрос может показаться корректным, однако оператор = предназначен для проверки равенства, а по логике вещей невозможно установить равенство с тем, чего нет. На самом глубинном уровне все операторы сравнения, включая =, оценивают условие как истинное или ложное. Поскольку NULL по определению представляет собой отсутствие значения, оно не может быть равно какому-либо другому значению — включая другое NULL-значение — и, следовательно, никогда не удовлетворит условиям, требующим истинности сравнения.

К практике!

Выполните любой (или все три) из приведенных выше запросов и убедитесь, что они не возвращают ожидаемых строк.

7.2.2. Как правильно выполнять поиск NULL-значений

Чтобы производить выборку по значениям null надлежащим образом, еще раз проговорим простыми словами то, что мы пытаемся сделать. Если требуется выбрать полные имена авторов, для которых значение второго имени отсутствует, соответствующее высказывание может звучать так: “I would like the first, middle, and last name of authors where the middle name is null” («Мне нужны имя, второе имя и фамилия авторов, у которых второе имя — это NULL»).

Ранее, преобразуя подобные словесные формулировки в SQL-запросы, мы заменяли слово *is* на оператор =. Но поскольку теперь мы имеем дело с условием фильтрации, включающим NULL-значения — которые несовместимы

с операторами сравнения, — вместо этого мы задействуем новый оператор, буквально соответствующий двум последним словам устной формулировки: `IS NULL`. Вот как будет выглядеть наш запрос, а его результаты показаны на рис. 7.8:

```
SELECT
    FirstName,
    MiddleName,
    LastName
FROM author
WHERE MiddleName IS NULL;
```

FirstName	MiddleName	LastName
Doug	NULL	Li
Jen	NULL	Strong
Rebecca	NULL	Miller
Andy	NULL	Melkin
Buck	NULL	Fernandez
Chris	NULL	Walenski
Deepthi	NULL	Mahadevan

Рис. 7.8. Выборка авторов без второго имени из таблицы author, включающая столбцы с именем, вторым именем и фамилией

Будьте всегда начеку с оператором `IS NULL` и NULL-значениями — последние могут преподнести и другие сюрпризы. Взгляните на рис. 7.7: у первой из 11 строк в таблице author в столбце `MiddleName` указано значение «K». Допустим, вы захотите выбрать все строки, кроме этой, применив исключающий запрос (глава 6). И попытаетесь сделать это при помощи следующего SQL-кода, результаты которого отображены на рис. 7.9:

```
SELECT
    FirstName,
    MiddleName,
    LastName
FROM author
WHERE MiddleName <> 'K';
```

FirstName	MiddleName	LastName
Jorge	Armando	Guerra
Robert	Grant	Davidson
Gail	Anne	Shawn

Рис. 7.9. Выборка строк, где второе имя не «K», из которой исключены все авторы без второго имени

Казалось бы, запрос этот должен выводить 10 строк, где второе имя не равно «K». Но не тут-то было. Поскольку условие фильтра ищет любые значения, не равные «K», оно отбрасывает все строки, содержащие значение null. «Ничто»

не может быть равным (или неравным) «чему-то», поэтому фильтр учитывает лишь те строки, где `MiddleName` имеет значение, отличное от NULL.

Возможно, именно к этому вы и стремились — получить только строки со значением в `MiddleName`. Но если предполагалась выборка всех 10 строк, то необходимо включить в условие фильтра оператор `IS NULL`, добавив его посредством `OR`. Результат показан на рис. 7.10:

```
SELECT
    FirstName,
    MiddleName,
    LastName
FROM author
WHERE MiddleName <> 'K'
    OR MiddleName IS NULL;
```

FirstName	MiddleName	LastName
Doug	NULL	Li
Jen	NULL	Strong
Jorge	Armando	Guerra
Robert	Grant	Davidson
Gail	Anne	Shawn
Rebecca	NULL	Miller
Andy	NULL	Melkin
Buck	NULL	Fernandez
Chris	NULL	Walenski
Deepthi	NULL	Mahadevan

Рис. 7.10. Выборка всех строк, где второе имя не K, включая строки с NULL-значениями

7.2.3. Как выполнять поиск значений, отличных от NULL

Теперь, когда мы научились включать в выборку строки с NULL-значениями, давайте посмотрим, как извлечь все строки, не содержащие NULL. Начнем по традиции со словесной формулировки того, что нам требуется: “I would like the first, middle, and last names of authors where the middle name isn’t null” («Мне нужны имя, второе имя и фамилия авторов, второе имя которых не является NULL»).

Слово *isn’t* — это сокращенная форма *is not* («не является»), и именно так выглядит наш следующий оператор, представленный в запросе ниже (результаты его применения см. на рис. 7.11):

```
SELECT
    FirstName,
    MiddleName,
    LastName
FROM author
WHERE MiddleName IS NOT NULL;
```

FirstName	MiddleName	LastName
Paul	K	Tripp
Jorge	Armando	Guerra
Robert	Grant	Davidson
Gail	Anne	Shawn

Рис. 7.11. Выборка всех строк таблицы author со вторым именем

Оператор `IS NOT NULL` позволяет выбрать все строки, имеющие какое-либо значение, отличное от `NULL`, в заданном столбце. Что примечательно, можно получить те же результаты, пользуясь подстановочным знаком, с которым мы познакомились ранее в этой главе:

```
SELECT
    FirstName,
    MiddleName,
    LastName
FROM author
WHERE MiddleName LIKE '%';
```

Почему этот запрос работает аналогично оператору `IS NOT NULL`? Шаблон `%` соответствует любой строке, при условии, что в столбце имеются данные. Поскольку значения null не содержат данных, подстановочные знаки никогда не совпадают с ними и не возвращают их в результирующем наборе, что по большому счету делает и оператор `IS NOT NULL`.

К практике!

Выполните два предыдущих запроса, задействовав `IS NOT NULL` и подстановочный знак `%`, и удостоверьтесь, что их результаты совпадают с рис. 7.11.

Отлично! Мы посвятили целых три главы изучению всевозможных способов фильтрации данных при выполнении запросов к таблице. И это только начало. В главе 8 мы поднимемся на новый уровень владения SQL, освоив формирование запросов к нескольким таблицам одновременно.

7.3. Практическое занятие

Сделаем паузу и освежим в памяти все способы фильтрации строк, которые мы освоили. Составьте несколько SQL-запросов, чтобы выбрать:

1. Полные имена всех авторов, второе имя которых *Anne* либо оно отсутствует.
2. Полные имена всех авторов, у которых нет второго имени, а имя начинается с буквы *D*.
3. Название и цену всех произведений, заглавия которых начинаются с артикля *The* и цена которых меньше 10 долларов.

4. Название и дату публикации любой книги, опубликованной после 1 января 2020 года, заглавие которой оканчивается на *S*.
5. Название всех произведений, в заглавии которых встречаются предлоги *of* или *in*.

7.4. Ответы

1.
```
SELECT
    FirstName,
    MiddleName,
    LastName
FROM author
WHERE MiddleName = 'Anne'
    OR MiddleName IS NULL;
```
2.
```
SELECT
    FirstName,
    MiddleName,
    LastName
FROM author
WHERE FirstName LIKE 'D%'
     AND MiddleName IS NULL;
```
3.
```
SELECT
    TitleName,
    Price
FROM title
WHERE TitleName LIKE 'The%'
    AND Price < 10;
```
4.
```
SELECT
    TitleName,
    PublicationDate
FROM title
WHERE TitleName LIKE '%s'
    AND PublicationDate > '2020-01-01';
```
5. Вопрос с подвохом, своего рода логическая задача! Необходимо учитывать, что при составлении запроса результаты могут оказаться либо слишком обширными, либо чересчур ограниченными. Поначалу вас может соблазнить следующее простое решение:

```
SELECT
    TitleName
FROM title
WHERE TitleName LIKE '%of%'
    OR TitleName LIKE '%in%';
```

Выполнив такой запрос, вы получите не только «The Call of the While», «Anne of Fact Tables» и «Catcher in the Try», соответствующие указанному критерию, но также «The Join Luck Club» и «The DateTime Machine», которые не должны были попасть в результат. Последние два включены в выборку потому, что их названия содержат последовательности букв, совпадающие с предлогами *of* и *in* в составе других слов.

Как исключить нежелательные результаты? Один из способов — добавить пробелы до и после искомой подстроки. Вот как выглядит такое решение:

```
SELECT
    TitleName
FROM title
WHERE TitleName LIKE '% of %'
    OR TitleName LIKE '% in %';
```

Теперь запрос выдает что нужно, но не торопитесь праздновать победу — его применение все еще ограничено нашим контекстом. Поскольку теперь мы ищем строки с пробелами до или после искомых слов, мы не получим совпадений, если название начинается или заканчивается предлогами *in* или *of*. Чтобы охватить и такие допустимые случаи (пусть даже маловероятные), необходимо добавить следующие условия:

```
SELECT
    TitleName
FROM title
WHERE TitleName LIKE '% of %'
    OR TitleName LIKE '% in %'
    OR TitleName LIKE 'of %'
    OR TitleName LIKE 'in %'
    OR TitleName LIKE '% of'
    OR TitleName LIKE '% in';
```

Признаю еще раз, задачка довольно сложная, и это неслучайно! Ее цель — пробудить ваше воображение, подстегнуть пытливость и позволить вам самостоятельно оценить данные: какие значения они могут принимать и как творчески применить ваши новые знания, чтобы получить именно тот результат, который необходим.

8 Запросы к нескольким таблицам

Еще в самом начале книги (глава 2) мы обсуждали, как реляционные базы данных хранят данные в объектах, именуемых *таблицами*, и с тех пор рассматривали различные способы построения запросов к этим таблицам. Не знаю, задумывались ли вы, что делает СУБД «реляционной», — и сейчас я как раз отвечу на этот вопрос.

Одна из ключевых особенностей реляционной СУБД — возможность организовывать наборы данных таким образом, чтобы они оставались связанными между собой, даже когда они хранятся в разных таблицах. Отсюда и название *реляционная* (от англ. relation, «отношение, взаимосвязь»). Такой подход к хранению информации обладает колоссальными возможностями: он позволяет не только логически группировать данные по таблицам, но и легко извлекать связанные данные из нескольких таблиц одним запросом.

Такого рода извлечение данных осуществляется посредством *соединения таблиц* (joining tables) — то есть объединения данных из двух и более таблиц на основе значений, устанавливающих реляционную связь между ними. Хотя соединение таблиц — операция распространенная и относительно простая, для получения ожидаемых результатов необходимо соблюдать ряд строгих формальных правил. Вскоре вы познакомитесь с этими правилами и научитесь реализовывать соединение таблиц в SQL надлежащим образом. Однако прежде необходимо рассмотреть ключевые понятия реляционных баз данных, а также основы реляционного моделирования и проектирования данных.

8.1. Принципы построения связей между данными

До сих пор мы не уделяли особого внимания словам *реляционный* и *связь*, однако при знакомстве со строками таблиц в главе 2 мы уже затронули один из аспектов взаимосвязей в СУБД. Представьте любую отдельную строку в таблице как совокупность связанных между собой значений. К примеру, первая строка таблицы title содержит такие значения, как `TitleID`, `TitleName` и ряд других, относящихся к книге «Pride and Predicates», — благодаря чему все эти значения связаны между собой. То же самое верно и для остальных строк таблицы, с той лишь разницей, что каждая строка представляет собой связанный набор значений для какого-то отдельного произведения.

Хотя до сих пор мы не рассматривали таких примеров, значения могут быть связаны и со строками в других таблицах. Все сведения, напрямую относящиеся к определенному названию книги, мы поместили в таблицу `title`, однако в базе данных `sqlnovel` есть и другие данные, связанные с этими произведениями, — просто хранятся они в другом месте. Так, таблица orderitem применяется для учета информации о заказах. Выполните следующий запрос и взгляните на результаты выборки, представленные на рис. 8.1:

```
SELECT *
FROM orderitem;
```

OrderID	OrderItem	TitleID	Quantity	ItemPrice
1001	1	101	1	9.95
1002	1	101	1	9.95
1003	1	101	1	9.95
1004	1	101	1	9.95
1005	1	101	1	9.95
1006	1	101	1	7.95
1007	1	101	2	7.95
1008	1	101	1	7.95
1009	1	101	1	9.95
1010	1	102	1	9.95

Рис. 8.1. Первые 10 строк таблицы orderitem, включая столбец TitleID

Заголовок третьего столбца `TitleID` совпадает с названием первого столбца в таблице `title`. Это имя указывает на то, что значения в таблице `orderitem` связаны со значениями в таблице `title`, что логично, поскольку речь идет об изданиях из нашей базы, которые заказывают покупатели.

Спрашивается, зачем хранить эти значения в разных таблицах, почему не поместить все необходимые нам сведения, связанные с книгами, непосредственно в таблицу `orderitem`? Что ж, существует ряд веских причин для хранения данных в отдельных таблицах. Однако вместо сухого перечисления куда полезнее продемонстрировать эти причины на примере данных, с которыми мы уже работали.

8.1.1. Данные без реляционных связей

Предположим, мы разрабатываем версию базы данных `sqlnovel` для учета заказов, решив хранить все необходимые данные в одной гипотетической таблице `orders`. Таблица эта будет содержать столбцы для даты заказа, названия книги, цены, а также имени и фамилии конкретного покупателя. Таблица может выглядеть примерно так, как показано на рис. 8.2.

OrderDate	TitleName	Price	FirstName	LastName
2015-06-01 00:00:00	Pride and Predicates	9.95	Chris	Dixon
2015-06-15 00:00:00	Pride and Predicates	9.95	David	Power
2016-09-02 00:00:00	Pride and Predicates	9.95	Chris	Dixon
2016-09-02 00:00:00	The Join Luck Club	9.95	Chris	Dixon
2017-11-14 00:00:00	The Join Luck Club	9.95	David	Power

Рис. 8.2. Наша предполагаемая таблица для учета заказов, содержащая дату заказа, название книги, цену и имя покупателя

На первый взгляд такая таблица кажется вполне рациональным способом учета заказов — возможно, вы даже реализовывали подобные структуры в электронных таблицах. Для небольшого числа заказов она может быть вполне приемлемой, однако более тщательный анализ данных выявляет существенную избыточность информации. В представленном фрагменте наблюдается повторение записей для двух идентичных наименований, заказанных разными покупателями в разное время. Главная проблема не в том, что для представления этих заказов используется пять строк, а в том, что одни и те же данные приходится многократно повторять.

А теперь представьте, что в этой таблице миллионы строк. И обозначенная проблема становится уже критичной — как с точки зрения времени выполнения запросов, так и в отношении объема хранимой информации. Особенно это касается строковых значений — названий книг (`TitleName`), а также имен и фамилий покупателей (`FirstName`, `LastName`) — ведь строки обычно занимают гораздо больше места, чем числовые значения.

И это далеко не единственная проблема. Что произойдет, если какие-либо данные изменятся — например, фамилия покупателя? Если клиент сменит фамилию и оформит новый заказ под новым именем, как мы свяжем заказы, сделанные под старой фамилией, с заказами под новой? С текущей структурой таблицы это невозможно: с разными фамилиями заказы будут выглядеть как оформленные разными людьми.

Та же проблема возникнет, если данные будут введены с ошибкой. Как мы сможем отслеживать продажи, если название книги случайно введут как «The Join Luck Clubs» (с лишней буквой «s» в слове «Club»)? А никак — это будет

считаться другим значением. И хотя мы с вами понимаем, что это опечатка, для базы данных такая запись — отдельное название, которое не попадет в выборку, если мы составим SQL-запрос с условием фильтра `WHERE TitleName = 'The Join Luck Club'`.

Как видите, хранение всех данных в одной таблице может породить массу проблем. Рассмотрим теперь, как применение основных принципов реляционных баз данных позволит организовать эти данные более эффективно.

8.1.2. Данные с реляционными связями

В реляционной базе данных мы стремимся максимально сократить количество повторяющихся вхождений одних и тех же значений. Добиться этого можно несколькими способами:

- *Организовать данные в логические группы значений.* Мы размещаем эти значения в отдельных таблицах, с тем чтобы каждая строка в каждой таблице относилась к чему-то уникальному — например, к названию книги. Еще раз обратите внимание на то, что любая строка в таблице `title` хранит данные именно таким образом.
- *Определить, какой столбец (или набор столбцов) будет содержать уникальное значение в каждой из новых таблиц.* Такой столбец или набор столбцов называется *первичным ключом* (primary key) — он позволяет связывать данные из других таблиц с этой таблицей. В таблице `title` первичным ключом является столбец `TitleID`.
- *Заменить повторяющиеся данные в других таблицах на значения первичных ключей — они послужат указателями на соответствующие строки.* Поскольку такие ключевые значения в нашей умозрительной таблице `orders` ссылаются на данные в других таблицах, они именуются *внешними ключами* (foreign key). В таблице на рис. 8.2 мы начнем с замены столбца `TitleName` на соответствующие значения `TitleID` из таблицы `title`, поскольку именно `TitleID` является ключевым идентификатором.

Итак, применим этот подход к таблице `orders`. Для начала еще раз взгляните на рис. 8.2 — он подскажет, как правильно структурировать данные.

Начнем с очевидного: у нас есть повторяющиеся значения в столбце `TitleName`, поэтому целесообразно создать отдельную таблицу, где будут храниться эти названия. Хорошая новость — как вы, вероятно, уже заметили, в нашей базе данных `sqlnovel` таблица `title` уже существует, и она хранит данные именно в таком виде. Давайте посмотрим на значения столбцов `TitleID` и `TitleName` в таблице `title` для двух произведений, фигурирующих в нашей таблице `orders`, — выборка по запросу представлена на рис. 8.3:

```
SELECT
    TitleID,
    TitleName
FROM title
WHERE TitleName IN ('Pride and Predicates', 'The Join Luck Club')
ORDER BY TitleID;
```

TitleID	TitleName
101	Pride and Predicates
102	The Join Luck Club

Рис. 8.3. Результаты по столбцам TitleID и TitleName для книг «Pride and Predicates» и «The Join Luck Club»

НА ЗАМЕТКУ Значения `TitleID` в таблице `title` должны быть уникальными, чтобы мы могли точно знать, на какую строку в таблице `title` следует ссылаться. Дублирование этих значений нарушит целостность данных: система запутается, а мы просто перестанем понимать, с какими значениями связана каждая строка.

Поскольку столбец `TitleID` выполняет функцию первичного ключа, мы можем заменить столбец `TitleName` в нашей таблице `orders` на `TitleID`, соответствующий значениям в таблице `title`. На рис. 8.4 показано, как выглядит теперь наша гипотетическая таблица.

OrderDate	TitleID	Price	CustomerFirstName	CustomerLastName
2015-06-01 00:00:00	101	9.95	Chris	Dixon
2015-06-15 00:00:00	101	9.95	David	Power
2016-09-02 00:00:00	101	9.95	Chris	Dixon
2016-09-02 00:00:00	102	9.95	Chris	Dixon
2017-11-14 00:00:00	102	9.95	David	Power

Рис. 8.4. Так будет выглядеть таблица orders после замены TitleName на TitleID

Теперь между этими таблицами установлена реляционная связь, позволяющая избежать проблем, о которых мы говорили ранее, — по крайней мере, в части, касающейся заглавий изданий. Мы экономим место, сохраняя компактное числовое значение вместо строки всякий раз, когда нужно сослаться на название книги. Кроме того, риск несогласованности данных снижается, поскольку название хранится в одном месте и любые правки заголовка автоматически распространяются на все связанные записи. Если потребуется сослаться на конкретное название из других таблиц, можно всегда задействовать значения `TitleID`.

НА ЗАМЕТКУ Именно этим и объясняется повсеместное применение реляционных СУБД для хранения данных самого разного рода. Реляционный подход к хранению информации обеспечивает эффективный расход памяти, простоту внесения изменений и — самое главное — гарантирует целостность и согласованность данных во всей системе.

Проанализировав таблицу `orders`, можно изыскать и другие возможности для оптимизации хранения данных. Так, поскольку все покупатели уникальны, целесообразно вынести сведения о них в отдельную таблицу, заменив эти данные в таблице `orders` ключевым значением. И снова нам повезло: в базе данных уже есть готовая таблица `customer` с первичным ключом `CustomerID`. На рис. 8.5 приведены результаты нашего запроса к таблице `customer` для конкретных значений имени (`FirstName`) и фамилии (`LastName`) в таблице `orders`:

```
SELECT
    CustomerID,
    FirstName,
    LastName
FROM customer
WHERE (FirstName = 'Chris' AND LastName = 'Dixon')
    OR (FirstName = 'David' AND LastName = 'Power')
ORDER BY CustomerID;
```

CustomerID	FirstName	LastName
1	Chris	Dixon
2	David	Power

Рис. 8.5. Выборка по столбцам CustomerID, FirstName и LastName для покупателей с именами Chris Dixon или David Power

OrderDate	TitleID	Price	CustomerID
2015-06-01 00:00:00	101	9.95	1
2015-06-15 00:00:00	101	9.95	2
2016-09-02 00:00:00	101	9.95	1
2016-09-02 00:00:00	102	9.95	1
2017-11-14 00:00:00	102	9.95	2

Рис. 8.6. Гипотетическая таблица orders со столбцом CustomerID для ссылки на имена в таблице customer

Теперь у нас есть три связанные таблицы, поэтому давайте заменим два столбца с именем покупателя в таблице `orders` одним столбцом, ссылающимся на соответствующие значения `CustomerID` из таблицы `customer`. На рис. 8.6 показана обновленная таблица заказов.

Между таблицей `customer` и нашей таблицей `orders` установилось отношение *один ко многим* (one-to-many). Так, для каждого заказа допустим лишь один покупатель, но любой конкретный покупатель может разместить свыше одного заказа. Это весьма распространенный тип связи в реляционных базах данных.

Теперь наши данные оптимально структурированы, однако можно внести еще один, последний штрих. Взгляните на третью и четвертую строки на рис. 8.6. Похоже, что заказчик с `CustomerID`, равным 1, приобрел две разные книги в один и тот же день, и мы рассматриваем это как один заказ. Покупатели часто включают в заказ сразу несколько позиций, и это вполне естественная ситуация, которую нам следует предусмотреть.

Однако здесь возникает сложность: как создать уникальный первичный ключ для таблицы `orders`? Ведь мы не можем назначить уникальный ключ для строк заказов, если имеются повторяющиеся строки для одного и того же заказа. Один из распространенных способов решения этой проблемы — разместить заказанные товары в отдельной таблице, разбив таким образом данные, связанные с заказом, на две таблицы. Поскольку любой заказ может содержать одно или несколько наименований, этот тип связи тоже образует отношение «один ко многим».

> **НА ЗАМЕТКУ** В реляционных базах данных между таблицами также существуют отношения *один к одному* (one-to-one) и *многие ко многим* (many-to-many). Как правило, эти типы реляционных связей встречаются реже, поэтому сейчас мы не будем углубляться в примеры. Просто имейте в виду, что в любой базе данных между таблицами могут существовать и другие виды отношений[1].

Коль скоро мы собираемся разбить данные в таблице `orders`, то необходимо выяснить для каждого столбца, относятся ли его значения к конкретному товару в заказе или к заказу в целом. Проведем анализ столбцов.

- `OrderDate` — эти значения одинаковы для всего заказа, так как все товары заказываются одновременно в рамках существующего заказа.
- `TitleID` — эти значения относятся к конкретному товару, поскольку заказ может содержать более одного произведения.
- `Price` — это значение относится к отдельным наименованиям, поэтому оно также является значением на уровне конкретного товара.
- `CustomerID` — это значение относится ко всему заказу, потому что у всех товаров в заказе один покупатель.

Теперь, когда мы подразделили значения по принадлежности к товару либо к заказу, можно разбить данные таблицы `orders` на две отдельные таблицы:

- `orderheader` — содержит значения, уникальные для всего заказа. Здесь мы создадим уникальный идентификатор заказа — столбец первичного ключа с именем `OrderID`. Эта таблица также будет включать столбцы с датой заказа (`OrderDate`) и идентификатором покупателя (`CustomerID`);
- `orderitem` — содержит значения, уникальные для наименований в конкретном заказе. Здесь мы создадим уникальный идентификатор товарной позиции —

[1] Связь «один к одному» технически допустима и используется для разделения данных по соображениям производительности, безопасности или при работе с опциональными атрибутами. Связь «многие ко многим» недопустима физически и всегда реализуется через промежуточную таблицу, создающую две связи «один ко многим». — *Примеч. науч. ред.*

столбец первичного ключа с именем `OrderItemID`, а также внешний ключ `OrderID` для связи между `orderitem` и `orderheader`. Эта таблица также будет включать столбцы с идентификатором названия книги (`TitleID`) и ценой (`Price`).

Мы реструктурировали данные нашей гипотетической таблицы `orders`, преобразовав их в фактические таблицы базы данных `sqlnovel`, и разобрались в том, как эти таблицы связаны между собой. Приступим к составлению SQL-команд, осуществляющих соединение данных этих таблиц на основе установленных реляционных связей.

8.2. Как соединять данные

Разобравшись, как устроены таблицы и ключи, посмотрим, как они работают в реальных запросах. Для этого мы сосредоточимся на предложении `FROM` — ведь именно в нем мы указываем, с какими данными будем работать.

8.2.1. Соединение двух таблиц

Когда требуется установить, кто из покупателей разместил первый заказ (OrderID 1001), можно начать с такого запроса:

```
SELECT CustomerID
FROM orderheader
WHERE OrderID = 1001;
```

К практике!

Знаю, вы уже заждались того момента, когда в этой главе можно будет написать настоящий SQL-код, так что для разминки выполните этот запрос.

Вряд ли есть смысл приводить иллюстрацию с одним столбцом и единственным значением в результатах запроса, поэтому, если вы предпочитаете продолжить чтение, а не выполнять запрос, просто запомните: он возвращает значение `CustomerID`, равное 1.

Знание того, что идентификатор `CustomerID` для первого заказа равен 1, может пригодиться для многих запросов. Но что, если нам нужно выяснить имя этого покупателя? Тогда придется воспользоваться реляционной связью между таблицами `orderheader` и `customer`, применив операцию *соединения* (join). Для этого следует явно указать имя второй таблицы (`customer`) и общий для обеих таблиц столбец (`CustomerID`), выступающий в роли условия соединения. Все это мы описываем в предложении `FROM` при помощи ключевых слов `JOIN` и `ON`.

Следующий запрос задействует JOIN и ON, чтобы соединить таблицы orderheader и customer и выдать не только CustomerID, но также имя и фамилию покупателя, как демонстрирует рис. 8.7:

```
SELECT
    orderheader.CustomerID,
    customer.FirstName,
    customer.LastName
FROM orderheader
JOIN customer
    ON orderheader.CustomerID = customer.CustomerID
WHERE orderheader.OrderID = 1001;
```

CustomerID	FirstName	LastName
1	Chris	Dixon

Рис. 8.7. Результат соединения таблиц orderheader и customer, отображающий идентификатор CustomerID, а также имя и фамилию покупателя для первого заказа

Давайте внимательно рассмотрим этот запрос. Первое, что бросается в глаза: ключевое слово JOIN ставится сразу после FROM, а за ним следует ON. Конструкция JOIN считается частью предложения FROM — она сообщает СУБД, что мы хотим «подтянуть» данные из другой таблицы.

Здесь уместно провести параллель с предложением WHERE: если в нем требуется задать несколько условий выборки, первое условие вводится после ключевого слова WHERE, а каждое последующее — посредством логического оператора AND. Аналогичным образом, в предложении FROM первая таблица указывается сразу после FROM (здесь — orderheader), а каждая последующая таблица, подключаемая для соединения, добавляется при помощи ключевого слова JOIN, за которым следует ее имя (а здесь — customer). Таким образом, если AND расширяет условия фильтрации в предложении WHERE, то JOIN расширяет набор источников данных в разделе FROM.

Однако одного лишь указания на то, что мы хотим соединить таблицы с помощью JOIN, недостаточно — необходимо еще явным образом обозначить, *каким образом* они связаны, то есть задать столбцы, по которым устанавливается реляционная связь. Для этого мы применяем ключевое слово ON в составе предиката. *Предикат* (predicate) — это любая часть SQL-команды, которая определяет, является ли некое условие истинным, ложным или неопределенным. Именно это и делает раздел ON в составе предложения JOIN: он находит все строки, в которых значения CustomerID совпадают в обеих таблицах. Если совпадение установлено, то есть условие истинно, то эти строки рассматриваются для включения в результирующее множество.

Хотя ранее я не уточнял это, но условие фильтрации в предложении WHERE — к примеру, WHERE orderheader.OrderID = 1001 — тоже является предикатом,

который подвергается логической оценке со стороны СУБД. Как и в случае с WHERE, соединение через JOIN действует как *исключающее условие* (exclusive condition): после проверки всех предикатов (и в JOIN, и в WHERE) в результирующий набор попадают только те строки, для которых все условия оказались истинными. У нас в выборке всего одна строка, поскольку лишь она удовлетворяет всем заданным условиям.

Еще один важный нюанс: в разделе ON нашего запроса мы используем *двухкомпонентные имена* (two-part names) столбцов. Такая запись имеет вид [имя_таблицы].[имя_столбца] — и это принципиально важно, потому что столбец CustomerID присутствует в обеих таблицах. Мы не можем просто написать ON CustomerID = CustomerID — система не поймет, о каком именно CustomerID идет речь. Если в этом примере все кажется очевидным, не обольщайтесь: во многих базах данных таблицы связаны по столбцам с разными именами. Именно поэтому мы всегда должны явно указывать и имя таблицы, и имя столбца.

> **СОВЕТ** Хотя и не важно, какая таблица и столбец указаны первыми в разделе ON конструкции JOIN, рекомендуется начинать с таблицы, упомянутой первой. Такой подход не только способствует удобочитаемости и организации данных, но и как вы увидите в главе 9, крайне важен при работе с другими типами соединений.

В итоге мы применяем двухкомпонентные обозначения столбцов во всем коде запроса — преимущественно в целях единообразия и удобочитаемости. Говорю *преимущественно*, ибо почти все они могут быть заменены на простые имена столбцов без указания имени таблицы — кроме случаев с CustomerID. Дело в том, что столбец CustomerID присутствует в обеих таблицах, и, если бы мы где-нибудь в запросе написали просто CustomerID, не сославшись на его родительскую таблицу, система выдала бы ошибку неоднозначности ссылки.

К практике!

Выполните запрос, выдавший результат на рис. 8.7. Также замените orderheader.CustomerID на CustomerID в предложении SELECT и убедитесь, что на панели вывода отображается ошибка.

8.2.2. Соединение нескольких таблиц

Несомненное преимущество операции соединения заключается в том, что мы не ограничены двумя таблицами. Можно вновь и вновь задействовать оператор JOIN для подключения дополнительных данных, при условии что нам известны столбцы, обеспечивающие реляционные связи между таблицами.

Помните, мы разнесли данные о заказах по двум таблицам: orderheader и orderitem. Если требуется найти дополнительную информацию о первом

заказе, к примеру цену товара, можно без особого труда добавить еще одно соединение (JOIN) к нашему запросу. Поскольку ранее мы установили, что таблицы orderheader и orderitem соединяются по столбцу OrderID, который является первичным ключом таблицы orderheader, то имеется возможность изменить наш запрос, добавив еще несколько строк SQL-кода, связанных с таблицей orderitem. На рис. 8.8 представлены результаты такого запроса:

```
SELECT
    orderheader.CustomerID,
    customer.FirstName,
    customer.LastName,
    orderitem.ItemPrice
FROM orderheader
JOIN customer
    ON orderheader.CustomerID = customer.CustomerID
JOIN orderitem
    ON orderheader.OrderID = orderitem.OrderID
WHERE orderheader.OrderID = 1001;
```

Здесь JOIN для таблицы orderitem следует после JOIN для таблицы customer, однако в этом конкретном запросе порядок подключения таблиц не имеет значения — мы с тем же успехом могли бы его поменять. Порядок перечисления соединяемых таблиц сводится к личным предпочтениям и удобочитаемости, при условии что все таблицы, участвующие в операции JOIN, объявлены в предложении FROM до достижения соответствующего условия ON.

CustomerID	FirstName	LastName	ItemPrice
1	Chris	Dixon	9.95

Рис. 8.8. Сведения о покупателе из таблицы customer и стоимость товара из первого заказа в таблице orderitem

Можно добавить еще одну таблицу, чтобы получить название заказанного товара, так как значение TitleName находится в четвертой таблице с именем title. Если помните, ссылка на TitleName в таблице orderitem осуществляется посредством внешнего ключа TitleID, это означает, что мы должны разместить наше JOIN-соединение с таблицей title после orderitem. Вот запрос, позволяющий получить результаты, приведенные на рис. 8.9:

```
SELECT
    orderheader.CustomerID,
    customer.FirstName,
    customer.LastName,
    orderitem.ItemPrice,
    title.TitleName
FROM orderheader
JOIN customer
    ON orderheader.CustomerID = customer.CustomerID
```

```
JOIN orderitem
    ON orderheader.OrderID = orderitem.OrderID
JOIN title
    ON orderitem.TitleID = title.TitleID
WHERE orderheader.OrderID = 1001;
```

CustomerID	FirstName	LastName	ItemPrice	TitleName
1	Chris	Dixon	9.95	Pride and Predicates

Рис. 8.9. Сведения о покупателе из таблицы customer, стоимость товара из первого заказа в таблице orderitem и значение TitleName из таблицы title

Можно и дальше добавлять соединения с другими таблицами для получения связанных данных — и в последующих главах, по мере знакомства с остальными таблицами БД, мы именно так и поступим. Однако, как вы уже заметили, наши SQL-запросы с обилием двухкомпонентных обозначений становятся слишком громоздкими. Даже для опытных разработчиков такой способ написания запросов с несколькими соединениями выглядит многословным. В то время как существует куда более удобочитаемый способ записи двухкомпонентных имен — и опирается он на очень простую и уже знакомую нам идею.

8.3. Псевдонимы таблиц

Помните, в главе 3 мы научились переименовывать столбцы в результирующем множестве при помощи *псевдонимов* (aliases). По сути дела, мы объявляли имя, под которым столбец должен отображаться в выборке. К счастью, язык SQL позволяет использовать псевдонимы не только для столбцов, но и для таблиц.

Впрочем, псевдонимы таблиц и столбцов различаются по своему назначению. Когда речь идет о столбцах, мы чаще всего хотим сделать их имена более понятными и информативными. Что касается таблиц, то тут все наоборот — мы стремимся к краткости. Обычно псевдоним таблицы состоит из одной-двух букв — это сокращает общий объем текста запроса и, при должном применении, делает его более удобным для чтения.

Среди самых распространенных способов задания псевдонимов таблиц — присвоение им одно- или двухбуквенных сокращений от их имен. К примеру, таблице `customer` можно присвоить псевдоним `c`, а таблице `title` — `t`. Псевдоним `o` подошел бы для `orderheader`, но поскольку в нашем запросе есть еще одна таблица, начинающаяся на «o» — `orderitem`, — лучше применить к ним двухбуквенные псевдонимы. Логичный подход — взять первую букву каждого слова в названии таблицы, в частности, `oh` для `orderheader` и `oi` для `orderitem`. Вот как это выглядит на практике — берем наш последний запрос и «переименовываем» таблицы:

```
SELECT
    oh.CustomerID,
    c.FirstName,
    c.LastName,
    oi.ItemPrice,
    t.TitleName
FROM orderheader oh
JOIN customer c
    ON oh.CustomerID = c.CustomerID
JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
JOIN title t
     ON oi.TitleID = t.TitleID
WHERE oh.OrderID = 1001;
```

Теперь в нашем запросе куда меньше символов, а значит, и анализировать его станет проще. Разумеется, мы могли бы присваивать псевдонимы таблицам так же, как и столбцам, — скажем, написать `FROM orderheader AS oh`. Однако на практике так поступают редко: ведь наша цель — сократить общий объем текста в коде запроса. Именно поэтому я настоятельно рекомендую всегда задействовать короткие псевдонимы таблиц, как показано здесь, особенно при соединении таблиц — это значительно улучшает читаемость запроса. В то же время существует пара правил, которые следует соблюдать при работе с псевдонимами таблиц.

- Псевдоним должен начинаться с буквы, а не с цифры или специального символа.
- За исключением первого символа, псевдоним может содержать цифры, но не специальные символы.

Единственная дополнительная рекомендация по псевдонимам — выбирайте осмысленные имена. Не следует называть первую таблицу `a`, вторую `b` и т. д. Если ваши псевдонимы хотя бы отдаленно напоминают реальные названия таблиц, ваши SQL-запросы будет гораздо легче читать и понимать.

К практике!

Перепишите любой из запросов в разделе 8.2 с вашими собственными псевдонимами.

8.4. Альтернативный способ присоединения данных

Ранее мы затронули понятие предиката — логического выражения, применяемого для оценки условий как в конструкциях соединения в предложении `FROM`, так и в условиях фильтрации в предложении `WHERE`. И хотя подход этот используется сегодня крайне редко, существует способ объединить все предикаты в предложении `WHERE`.

Я упоминаю о нем лишь потому, что рано или поздно вы наверняка столкнетесь с SQL-запросом, написанным кем-то другим, где соединения реализованы именно таким образом. Как вы убедитесь сейчас сами, способ этот в целом не рекомендуется по ряду причин. Вот как выглядел бы наш последний запрос, предпочти мы такую альтернативу:

```
SELECT
    oh.CustomerID,
    c.FirstName,
    c.LastName,
    oi.ItemPrice,
    t.TitleName
FROM
    orderheader oh,
    customer c,
    orderitem oi,
    title t
WHERE oh.OrderID = 1001
    AND oh.CustomerID = c.CustomerID
    AND oh.OrderID = oi.OrderID
    AND oi.TitleID = t.TitleID;
```

Первое, что бросается в глаза, — это упрощенный синтаксис предложения `FROM`, где перечисление таблиц осуществляется через запятую. Он немного ближе к формату разговорных речевых высказываний, которые выручают нас на протяжении всей книги. Ведь мы могли бы описать намерение составителя запроса простыми словами примерно так: “I would like the customer ID, first name, last name, item price, and title name from the orderheader table, customer table, orderitem table, and title table where the order ID is 1001” («Мне нужны идентификатор покупателя, его имя, фамилия, цена товара и название книги из таблиц orderheader, customer, orderitem и title, где идентификатор заказа равен 1001»).

Впрочем, даже такой метод непросто перевести из устной формулировки в SQL — ведь нам все равно придется точно указать СУБД, как именно соединять все эти таблицы. Хотя представленный способ присоединения данных технически корректен и полностью допустим в SQL, у него имеются существенные недостатки.

- Чтобы понять, как подключаются все эти таблицы, приходится внимательно читать каждую строку в предложении `WHERE` — иначе не разберешь, по какому условию выполняется то или иное соединение: условия реляционной связи не выделены структурно и смешаны с предикатами фильтрации. Да, в таком запросе меньше символов, поскольку не требуется писать `JOIN` для каждого соединения. Но в этом и заключается главное неудобство: чтобы выяснить, как связаны даже две таблицы, необходимо тщательным образом изучить все предложение `WHERE`. Смешение воедино всех условий соединения и фильтров ощутимо затрудняет отладку и сопровождение кода — словно вы выуживаете отдельно взятую макаронину из тарелки спагетти.

- Кроме того, этот метод позволяет задействовать лишь один тип соединения — внутреннее (`INNER JOIN`). В главе 9 мы рассмотрим более гибкие и мощные способы подключения данных. При помощи такого подхода реализовать их будет невозможно. И поэтому не следует писать SQL-запросы, в которых соединения задаются в предложении `WHERE`.

Операции соединения (`JOIN`) станут критически важными компонентами подавляющего большинства запросов, которые мы напишем в дальнейшем. И если вы еще не вполне усвоили этот материал, не поленитесь перечитать главу и отточить навыки на примерах из раздела 8.2, а также выполнить практическое задание. Как только соединения таблиц станут вашей второй натурой — добро пожаловать в главу 9!

8.5. Практическое занятие

1. Чем будут отличаться результаты выполнения этих двух запросов, использующих для фильтрации в предложении `WHERE` различные таблицы?

```
SELECT
    t.TitleName
FROM orderheader oh
JOIN customer c
    ON oh.CustomerID = c.CustomerID
JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
JOIN title t
    ON oi.TitleID = t.TitleID
WHERE oh.OrderID = 1001;

SELECT
    t.TitleName
FROM orderheader oh
JOIN customer c
    ON oh.CustomerID = c.CustomerID
JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
JOIN title t
    ON oi.TitleID = t.TitleID
WHERE oi.OrderID = 1001;
```

2. Сколько заказов разместил покупатель по имени Крис Диксон (Chris Dixon)? Составьте запрос для получения ответа на этот вопрос.
3. При помощи запроса выберите имена покупателей, заказавших книгу в 2015 году.
4. Как при помощи соединений таблиц (`JOIN`) и псевдонимов переписать следующий запрос, извлекающий имена всех покупателей книги «The Sum Also Rises»?

```
SELECT
    customer.FirstName,
    customer.LastName
FROM title, orderheader, customer, orderitem
WHERE title.TitleName = 'The Sum Also Rises'
    AND orderheader.OrderID = orderitem.OrderID
    AND orderitem.TitleID = title.TitleID
    AND orderheader.CustomerID = customer.CustomerID;
```

5. В разделе 8.4 мы узнали, что допустимо переместить все предикаты в предложение `WHERE`. А что произойдет в том случае, если мы перенесем все предикаты в предложение `FROM`, как показано в следующем запросе?

```
SELECT
    t.TitleName
FROM orderheader oh
JOIN customer c
    ON oh.CustomerID = c.CustomerID
JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
    AND oh.OrderID = 1001
JOIN title t
    ON oi.TitleID = t.TitleID;
```

8.6. Ответы

1. Результаты выполнения запросов идентичны. Поскольку наш запрос сопоставляет все значения `OrderID` из таблицы `orderheader` с таблицей `orderitem`, любой из столбцов `OrderID` можно задействовать в условии фильтрации `WHERE` для возврата одинаковых результатов.
2. Покупатель Крис Диксон разместил три заказа. Выяснить это можно при помощи следующего запроса:

```
SELECT
    oh.OrderID
FROM orderheader oh
JOIN customer c ON oh.CustomerID = c.CustomerID
WHERE c.FirstName = 'Chris'
    AND c.LastName = 'Dixon';
```

3. Следующие восемь покупателей разместили заказ в 2015 году:
   - Chris Dixon
   - David Power
   - Arnold Hinchcliffe
   - Keanu O'Ward
   - Lisa Rosenqvist

- Maggie Ilott
- Cora Daly
- Dan Wilson

Их список можно получить при помощи следующего запроса:

```
SELECT
    c.FirstName,
    c.LastName,
    oh.OrderDate
FROM orderheader oh
JOIN customer c ON oh.CustomerID = c.CustomerID
WHERE oh.OrderDate >= '2015-01-01 00:00:00'
AND oh.OrderDate < '2016-01-01 00:00:00';
```

4. Переписать запрос при помощи соединений таблиц (`JOIN`) и псевдонимов можно так:

```
SELECT

    c.FirstName,
    c.LastName
FROM orderheader oh
JOIN customer c
    ON oh.CustomerID = c.CustomerID
JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
JOIN title t
    ON oi.TitleID = t.TitleID
WHERE t.TitleName = 'The Sum Also Rises';
```

5. Запрос отработает корректно и даст тот же результат, однако для лучшей читаемости кода рекомендуется не смешивать условия соединения таблиц (`JOIN`) с условиями фильтрации (`WHERE`).

9 Применение различных видов соединений

Умение объединять таблицы — важнейший навык при написании SQL-запросов. Пока что мы ограничивались лишь одним видом соединения. Справедливости ради стоит отметить, что соединение это является самым распространенным, однако, как вы узнаете из этой главы, во многих случаях его применение не позволяет получить нужные результаты.

К примеру, вас могут попросить составить перечень всех заказов за год и указать, применялся ли в каждом из них конкретный промокод. Или вывести имена клиентов, не сделавших ни одного заказа в течение года. Или получить список клиентов из определенного города или региона и показать, кто из них оформлял заказы, а кто — нет. Ни одну из этих задач невозможно решить при помощи соединения, описанного в главе 8. Значит, пришло время освоить и другие виды соединений для выполнения перечисленных SQL-запросов.

9.1. Внутренние соединения

Для начала давайте подробнее поговорим о ключевом слове `JOIN`, с которым мы познакомились в главе 8. На самом деле это сокращенная запись для `INNER JOIN` — соединения, которое выбирает только совпадающие строки из обеих таблиц. Если строка не имеет соответствия в другой таблице, она не попадет в результат.

На протяжении почти всей главы мы будем работать с двумя таблицами: уже известной вам `orderheader` и новой, `promotion`. Эта таблица содержит промокоды для получения скидок на заказываемые книги. Первичный ключ таблицы `promotion` — `PromotionID`, и на него ссылается одноименный столбец `PromotionID` (внешний ключ) в таблице `orderheader`.

Мы выбрали эту пару таблиц неслучайно: в каждой из них имеются строки, не связанные с другой таблицей. Так, некоторые строки в таблице `promotion` содержат промокоды, которые не были востребованы ни в одном заказе, а часть строк в таблице `orderheader` содержат заказы, оформленные без промокода. Более того, связь между таблицами характеризуется отношением «один ко многим»: любой промокод можно применить к нескольким заказам, в то время как каждый заказ может использовать лишь один промокод.

Также необходимо отметить, что в таблице `promotion` 12 строк, а в таблице `orderheader` — 50. На протяжении главы я буду периодически ссылаться на количество строк в этих таблицах.

К практике!

Для определения количества строк в таблицах выполните команды `SELECT * FROM promotion` и `SELECT * FROM orderheader`. Уточните число строк в каждой таблице, просмотрев сообщение в столбце `Message` на панели вывода (Output). Разумеется, при желании можно подсчитывать возвращенные строки самостоятельно, но это более трудоемко и не исключает вероятность ошибки.

Начнем со следующего запроса, который находит идентификатор заказа и промокод для любого заказа, где он задействован. Часть результирующей выборки представлена на рис. 9.1:

```
SELECT
    oh.OrderID,
    p.PromotionCode
FROM orderheader oh
JOIN promotion p
    ON oh.PromotionID = p.PromotionID;
```

OrderID	PromotionCode
1006	2OFF2015
1007	2OFF2015
1008	2OFF2015
1013	2OFF2016
1014	2OFF2016
1015	2OFF2016
1016	2OFF2016
1022	2OFF2017

Рис. 9.1. Фрагмент выборки (8 из 20 строк), содержащей значения столбцов OrderID и PromotionCode для всех заказов, использовавших промокод

В столбце `Message` панели вывода отображается сообщение 20 row(s) returned («Возвращено 20 строк»). Обратите внимание: значения `PromotionCode` не представляют собой 20 неповторяющихся промокодов, а вот значения `OrderID` — это

20 уникальных идентификаторов. Это и есть проявление реляционной связи «один ко многим», когда один промокод может применяться ко множеству заказов.

Тот же запрос можно написать и более развернуто, прибегнув к полной форме записи `INNER JOIN` — результат останется прежним (20 строк):

```
SELECT
    oh.OrderID,
    p.PromotionCode
FROM orderheader oh
INNER JOIN promotion p
    ON oh.PromotionID = p.PromotionID;
```

> **СОВЕТ** В запросах, где присутствуют лишь внутренние соединения, допустимо писать `JOIN` вместо `INNER JOIN`. Однако если вы планируете добавлять другие типы соединений (речь о них впереди), следует явным образом указывать `INNER JOIN` для наглядности и удобочитаемости.

Круговая диаграмма Венна — это весьма популярный способ визуализации логической связи между значениями в таблицах. Представьте два пересекающихся круга, каждый из которых содержит данные одной таблицы. Пересекающаяся область кругов отображает общие значения обеих таблиц, а непересекающиеся — значения, уникальные для каждой из них.

На рис. 9.2 представлена диаграмма Венна для внутреннего соединения таблиц `promotion` и `orderheader`. В этой главе мы рассмотрим несколько диаграмм Венна, чтобы наглядно продемонстрировать, какие значения включаются в результирующее множество при использовании разных типов соединений. Так, закрашенная область пересечения на рис. 9.2 соответствует данным, возвращаемым в выборке, а непересекающиеся участки — данным, исключенным из результатов.

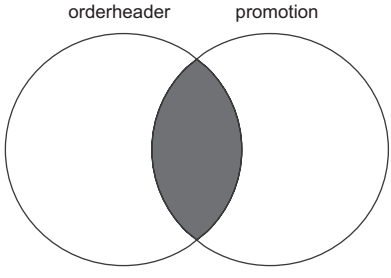


Рис. 9.2. Диаграмма Венна для внутреннего соединения, примененного в последнем запросе

Данные, возвращаемые нашим внутренним соединением, включают только общие значения, удовлетворяющие условию соединения, то есть те строки, в которых значения `PromotionID` в обеих таблицах совпадают. Но мы помним, что это не единственный способ соединения таблиц. Мы можем выполнить соединение и таким образом, чтобы в результат вошли все строки одной из таблиц, даже если для них отсутствуют соответствующие значения в другой таблице. В этих целях применяются ключевые слова `OUTER JOIN` («внешние соединения»).

9.2. Внешние соединения

Синтаксис, применяемый для внешних соединений, похож на синтаксис внутренних соединений: вы используете ключевое слово `JOIN` для указания соединяемых таблиц и `ON` для определения условия связи между столбцами таблиц. Если требуются дополнительные условия соединения, они задаются с помощью ключевого слова `AND`.

Одно из главных различий между внутренними и внешними соединениями заключается в существовании разных типов внешних соединений, поэтому в SQL необходимо явно обозначать, какой тип внешнего соединения вы задействуете. Начнем с `LEFT OUTER JOIN` — левого внешнего соединения.

9.2.1. Левые внешние соединения

Слово *left* («левый») в `LEFT OUTER JOIN` указывает на то, что мы хотим выбрать все строки из левой таблицы в нашем соединении, независимо от наличия совпадений. Если данный термин не вполне понятен, можно истолковать его еще проще: возвращаются все строки из *первой* таблицы, указанной до `JOIN`.

Допустим, требуется вывести список всех идентификаторов заказов и, если в них присутствует промокод, выяснить, какой именно код был активирован. Для этого мы возьмем предыдущий запрос и просто заменим `INNER JOIN` на `LEFT OUTER JOIN`:

```
SELECT
    oh.OrderID,
    p.PromotionCode
FROM orderheader oh
LEFT OUTER JOIN promotion p
    ON oh.PromotionID = p.PromotionID;
```

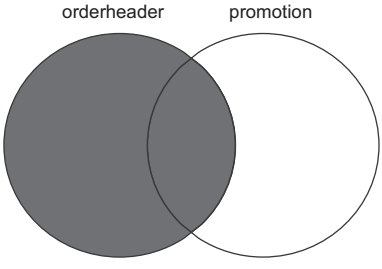


Рис. 9.3. Диаграмма Венна для левого внешнего соединения

В этом запросе левой таблицей является `orderheader`, так как она упомянута первой. В нашем оформлении кода `orderheader` расположена выше `promotion`, а не слева, однако при линейной записи без переносов `orderheader` находилась бы слева от `promotion` в тексте запроса. Данный тип левого внешнего соединения проиллюстрирован в форме диаграммы на рис. 9.3.

Сообщение на панели вывода информирует о возврате 50 строк, что соответствует ожиданиям, учитывая известное количество строк в таблице `orderheader`. Для удобства работы ограничим выборку первыми восемью заказами, применив фильтр, как показано на рис. 9.4:

```
SELECT
    oh.OrderID,
    p.PromotionCode
FROM orderheader oh
LEFT OUTER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
WHERE oh.OrderID <= 1008;
```

OrderID	PromotionCode
1001	NULL
1002	NULL
1003	NULL
1004	NULL
1005	NULL
1006	2OFF2015
1007	2OFF2015
1008	2OFF2015

Рис. 9.4. Выборка содержит столбец OrderID и все промокоды, примененные в первых восьми заказах. Значение PromotionCode указано лишь для заказов с идентификаторами 1006, 1007 и 1008

Результирующий набор включает строку для каждой записи в таблице `orderheader`, удовлетворяющей нашему условию (`OrderID` меньше или равен 1008), независимо от того, присутствует ли в ней значение `PromotionID` для соединения с таблицей `promotion`. Для заказов без промокода в столбце `PromotionCode` указывается NULL-значение.

Хотя у нас уже большой опыт работы с фильтрами, необходимо проявлять осторожность при добавлении условий фильтрации к внешним соединениям. Если добавить условие для конкретного значения из правой таблицы в предложение `WHERE` запроса с `LEFT OUTER JOIN`, мы в действительности превратим наше левое внешнее соединение в обычное внутреннее (`INNER JOIN`). Фильтрация по конкретным значениям в правой таблице исключит из выборки все строки, которые могут содержать значения null в правой таблице. Продемонстрируем эту ситуацию на примере схожего запроса, фильтрующего по конкретному значению `PromotionCode`, — результаты показаны на рис. 9.5:

```
SELECT
    oh.OrderID,
    p.PromotionCode
FROM orderheader oh
LEFT OUTER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
WHERE p.PromotionCode = '2OFF2015';
```

OrderID	PromotionCode
1006	2OFF2015
1007	2OFF2015
1008	2OFF2015

Рис. 9.5. Результаты левого внешнего соединения с фильтром по правой таблице (promotion) — выборка сократилась до размера, идентичного внутреннему соединению

Теперь результаты включают только три строки для заказов с промокодом 2OFF2015, а не по одной строке для каждого идентификатора заказа в `orderheader`, как следовало бы ожидать от левого внешнего соединения. Поскольку фильтр в предложении `WHERE` производил отбор по значениям из правой таблицы (в этом запросе — по значению `PromotionCode`), он был применен к результатам из обеих таблиц. Фактически, с таким условием фильтрации по правой таблице в `WHERE` можно было бы спокойно заменить `LEFT OUTER JOIN` на `INNER JOIN` и получить те же результаты.

К практике!

Выполните предыдущий запрос для другого промокода, скажем, 2OFF2016, сначала с `LEFT OUTER JOIN`, а затем с `INNER JOIN`, чтобы удостовериться, что в обоих случаях результаты одинаковы.

Если нужно получить перечень всех заказов, но отметить именно те, где применялся конкретный промокод (например, 2OFF2015), это тоже вполне осуществимо. Просто перенесите условие фильтра из предложения `WHERE` в само соединение:

```
SELECT
    oh.OrderID,
    p.PromotionCode
FROM orderheader oh
LEFT OUTER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
    AND p.PromotionCode = '2OFF2015';
```

В итоге будет выбрано 50 строк — по одной для каждого `OrderID`, — поскольку мы не применяем фильтрацию к левой таблице. Значение столбца `PromotionCode` в результирующем наборе равно 2OFF2015 либо NULL.

Постарайтесь запомнить этот важный нюанс в применении фильтров к условию соединения. Нередко возникает необходимость задействовать внешнее соединение и одновременно накладывать фильтры на конкретные значения в двух или более таблицах. Если вы поместите такой фильтр в предложение `WHERE` для таблицы, подключенной посредством внешнего соединения, вы рискуете ненароком преобразовать его во внутреннее соединение.

9.2.2. Правые внешние соединения

Если `LEFT OUTER JOIN` возвращает все строки из левой (первой) таблицы независимо от их соответствия правой (второй) таблице, `RIGHT OUTER JOIN` действует противоположным образом. А именно эти ключевые слова возвращают все строки из *правой* таблицы, независимо от их соответствия строкам левой таблицы по условию соединения.

Задействуем правое соединение, чтобы вывести все промокоды — даже те, что ни разу не использовались. Фрагмент результатов отображен на рис. 9.6:

```
SELECT
    p.PromotionCode,
    oh.OrderID
FROM orderheader oh
RIGHT OUTER JOIN promotion p
    ON oh.PromotionID = p.PromotionID;
```

PromotionCode	OrderID
2OFF2015	1008
2OFF2015	1007
2OFF2015	1006
2OFF2016	1016
2OFF2016	1015
2OFF2016	1014
2OFF2016	1013
2OFF2017	1024

Рис. 9.6. Часть выборки (8 из 23 строк), на которой представлены все идентификаторы заказов и промокоды, даже те, что не были востребованы

Результаты нашего запроса для правого соединения можно визуализировать при помощи диаграммы на рис. 9.7.

Последний запрос возвращает 23 строки, что больше, чем 12 строк в таблице `promotion`. При ближайшем рассмотрении видно, что многие строки содержат повторяющиеся значения `PromotionCode`, поскольку один код можно задействовать с несколькими заказами. Именно эти повторы и увеличили общее число строк в выдаче.

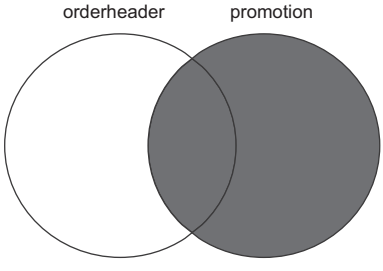


Рис. 9.7. Диаграмма Венна для правого внешнего соединения

Кроме того, пролистав результаты, вы увидите, что у некоторых значений `PromotionCode` указано NULL для `OrderID`. Это как раз те промокоды, которые не использовались ни в одном заказе. Тем не менее они присутствуют в нашем результирующем наборе, поскольку все строки таблицы `promotion` (являющейся правой таблицей в запросе) гарантированно включаются в выборку благодаря правому внешнему соединению.

9.2.3. Используем внешние соединения для поиска строк без совпадений

Внешние (левое или правое) соединения можно применять не только для отображения всех строк таблицы независимо от наличия соответствий, но и для

обратной задачи — для поиска всех строк, не имеющих «пар». Для этого достаточно явным образом задать условие фильтра `IS NULL` для таблицы, в которой осуществляется поиск совпадений.

К примеру, чтобы выбрать промокоды, не востребованные ни в одном заказе, добавим к предыдущему запросу условие `WHERE oh.PromotionID IS NULL` (рис. 9.8):

```
SELECT
    p.PromotionCode,
    oh.OrderID
FROM orderheader oh
RIGHT OUTER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
WHERE oh.PromotionID IS NULL;
```

PromotionCode	OrderID
1OFF2020	NULL
1OFF2021	NULL
2OFF2021	NULL

Рис. 9.8. Выборка всех промокодов, не имеющих соответствующего идентификатора заказа, то есть никогда не использовавшихся

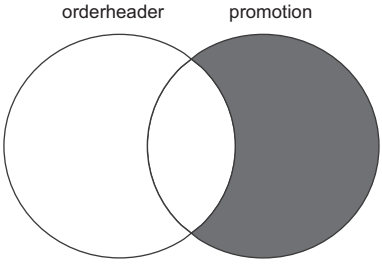


Рис. 9.9. Диаграмма Венна для правого внешнего соединения, исключающего строки левой таблицы

Хотя ранее я отмечал, что применение условия фильтрации к присоединяющей таблице фактически превращает внешнее соединение во внутреннее, проверка на NULL-значения является исключением из правила. Помните, что такое действие — это не проверка на равенство двух значений, а запрос на наличие пустых значений. Такой запрос относится к числу типичных; вам часто придется находить значения, которые имеются в одной таблице, но отсутствуют в другой, что можно изобразить при помощи диаграммы, приведенной на рис. 9.9.

9.2.4. Смена левых и правых соединений

При попытке выполнить предыдущий запрос в SQLite он не сработает, поскольку эта СУБД не поддерживает команду `RIGHT OUTER JOIN`, что, впрочем, легко обойти — достаточно поменять таблицы местами и задействовать `LEFT OUTER JOIN`:

```
SELECT
    p.PromotionCode,
    oh.OrderID
```

```
FROM promotion p
LEFT OUTER JOIN orderheader oh
    ON p.PromotionID = oh.PromotionID
WHERE oh.PromotionID IS NULL;
```

Такой запрос выдает те же результаты, что отображены на рис. 9.8, поскольку мы просто поменяли порядок указания двух таблиц в предложении FROM и сменили соединение с RIGHT OUTER JOIN на LEFT OUTER JOIN. Диаграмма на рис. 9.10 иллюстрирует логику операции.

> **СОВЕТ** По возможности старайтесь задействовать в запросах только левые или только правые внешние соединения, но не оба типа одновременно. Лишь в исключительных случаях потребуется включать оба типа внешних соединений, тогда как последовательное применение одного типа повышает удобочитаемость запроса. Кроме того, некоторые СУБД не поддерживают правые внешние соединения. По этой причине в оставшейся части книги мы будем отдавать предпочтение левым внешним соединениям.

И последнее замечание относительно левых и правых соединений: их не обязательно записывать полностью. Если INNER JOIN можно сократить до JOIN, то LEFT OUTER JOIN и RIGHT OUTER JOIN — до LEFT JOIN и RIGHT JOIN соответственно. Так, для улучшения удобочитаемости приведенный выше запрос может быть оформлен без ключевого слова OUTER:

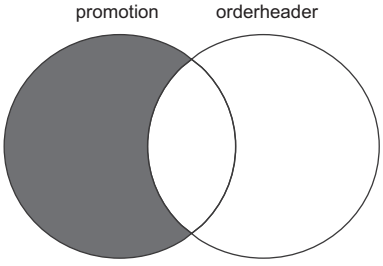


Рис. 9.10. Диаграмма Венна для левого внешнего соединения, исключающего строки правой таблицы

```
SELECT
    p.PromotionCode,
 oh.OrderID
FROM promotion p
LEFT JOIN orderheader oh
    ON p.PromotionID = oh.PromotionID
WHERE oh.PromotionID IS NULL;
```

> **НА ЗАМЕТКУ** Некоторые СУБД поддерживают FULL OUTER JOIN (или просто FULL JOIN). Этот редко используемый тип соединения возвращает объединенный результат LEFT JOIN и RIGHT JOIN для двух таблиц. Мы не будем писать запросы с FULL OUTER JOIN, поскольку MySQL, MariaDB и SQLite не поддерживают такой тип соединения. Тем не менее в главе 10 мы рассмотрим альтернативный способ формирования аналогичной выборки.

9.2.5. Ключевое слово USING

Существует два других способа написания внутренних, левых, правых и внешних соединений, однако они применяются значительно реже. Первый

способ — применение ключевого слова USING, которое заменяет в соединении ON и избавляет от необходимости указывать имена таблиц или их псевдонимы. Взамен USING требует, чтобы имена связываемых столбцов были одинаковыми в обеих таблицах. При помощи USING можно следующим образом переписать последний запрос и получить результаты, показанные на рис. 9.8:

```
SELECT
    p.PromotionCode,
    oh.OrderID
FROM promotion p
LEFT JOIN orderheader oh
    USING (PromotionID)
WHERE oh.PromotionID IS NULL;
```

Требование идентичности имен столбцов для соединения таблиц является главным препятствием при работе с USING, ведь в связанных таблицах реальных баз данных редко встречаются столбцы с одинаковыми именами. Поэтому оператор USING непопулярен, и многие SQL-разработчики либо вовсе о нем не знают, либо не умеют им грамотно пользоваться. Мы же знакомимся с ним исключительно на тот случай, если вы вдруг наткнетесь на ключевое слово USING в сторонних SQL-запросах.

9.2.6. Естественные соединения

Вторым редким способом реализации внутренних, левых, правых и внешних соединений является *естественное соединение* (natural join). При естественном соединении не требуется указывать имена столбцов, участвующих в реляционной связи: система автоматически объединяет столбцы с одинаковыми именами из двух таблиц. Для этого предназначено ключевое слово NATURAL, а предложения ON или USING опускаются. Перепишем предыдущий запрос еще раз — на этот раз с использованием естественного соединения:

```
SELECT
    p.PromotionCode,
    oh.OrderID
FROM promotion p
NATURAL LEFT JOIN orderheader oh
WHERE oh.PromotionID IS NULL;
```

Я настоятельно не рекомендую применять естественные соединения, хотя, безусловно, они позволяют еще больше сократить SQL-код. В первую очередь, не рекомендую потому, что естественные соединения скрывают логику связи между таблицами. И тот, кто будет читать ваш код, просто не поймет, по каким полям связаны таблицы promotion и orderheader.

Вторая, более серьезная проблема связана со столбцами, имена которых совпадают случайно. В наших учебных таблицах `promotion` и `orderheader` присутствует лишь одна, хорошо заметная пара столбцов с одинаковыми именами, тогда как в реальных базах данных множество таблиц содержит типовые столбцы, такие как `CreateDate` («Дата создания») или `ModifiedDate` («Дата изменения»), позволяющие отслеживать изменения в строках. И хотя такие столбцы нередко имеют одинаковые имена в пределах одной БД, они не предназначены для связывания данных между таблицами. Применение естественного соединения к таблицам со случайно совпавшими именами столбцов приведет к автоматическому объединению по этим столбцам, что даст непредсказуемый и, скорее всего, неверный результат.

ВНИМАНИЕ Естественные соединения не поддерживаются в SQL Server.

9.3. Перекрестные (декартовы) соединения

Последний тип соединения, который мы рассмотрим, представляет собой еще один «экзотический» вариант — *перекрестное соединение* (cross join). Его необычность заключается в том, что, в отличие от всех прочих соединений, обсуждавшихся выше, оно применяется не для поиска строк с определенными значениями. Вместо этого перекрестное соединение находит все возможные сочетания строк, сопоставляя каждую строку одной таблицы с каждой строкой другой.

Перекрестное соединение также известно как *декартово соединение* (Cartesian join), по аналогии с математической операцией «декартово произведение», которая описывает именно такое результирующее множество всех возможных парных значений из двух наборов данных. Чтобы при помощи декартова соединения выбрать все возможные комбинации PromotionCodes из таблицы `promotion` и OrderIDs из таблицы `orderheader`, можно составить следующий запрос:

```
SELECT
    p.PromotionCode,
    oh.OrderID
FROM promotion p
CROSS JOIN orderheader oh;
```

Результаты запроса отражают все возможные варианты сопоставления значений из двух таблиц, поэтому неудивительно, что выборка по запросу составляет 600 строк. У нас есть 12 строк в таблице `promotion` и 50 строк в таблице `orderheader`, поэтому простое умножение (12×50) подтверждает, что в результирующем наборе ожидается 600 строк.

Перекрестное соединение не предназначено для поиска конкретных строк, зато оно незаменимо, когда необходимо сгенерировать полный список всех

возможных комбинаций. В частности, оно позволяет создать матрицу всех размеров и цветов определенного продукта. Или быстро заполнить большой объем тестовых данных — скажем, искусственно раздуть базу заказов для нагрузочного тестирования.

> **ВНИМАНИЕ** Ранее отмечалось, что в MySQL можно сокращать `INNER JOIN` до `JOIN`. Однако при этом важно помнить: если вы соединяете таблицы, пользуясь лишь словом `JOIN` и опуская условие соединения, результаты будут отражать декартово соединение, а не внутреннее соединение, как вы планировали. В других СУБД при применении `INNER JOIN` оператор `ON` может быть обязательным.

Перекрестное соединение — последний из множества типов соединений, рассмотренных в этой главе. Возможно, вас несколько расстроил тот факт, что целый ряд коротких форм записи SQL-соединений не рекомендуются к применению, однако в главе 10 мы познакомимся с другими методами объединения данных — более надежными, эффективными, распространенными, а в иных случаях и лучшими при работе с реляционными базами данных. А пока давайте закрепим пройденный материал на практике.

9.4. Практическое занятие

В начале главы я описал несколько ситуаций, с которыми вы можете столкнуться в работе. Пришла пора применить ваши знания для написания запросов, позволяющих получить эти результаты. В первых трех упражнениях постарайтесь задействовать левое соединение:

1. Составьте запрос, отображающий значения идентификатора (`OrderID`) и даты (`OrderDate`) всех заказов за 2019 год, а также промокод (`PromotionCode`), если он был задействован.
2. Составьте запрос, выводящий имена и фамилии клиентов, не размещавших заказы в 2020 году.
3. Составьте запрос, выводящий имена и фамилии клиентов, а также идентификатор (`OrderID`) и дату (`OrderDate`) всех заказов, сделанных в 2021 году покупателями из Калифорнии (для которых значение штата (`State`) в таблице `customer` равно `'CA'`).
4. Создайте запрос с применением декартова соединения (`CROSS JOIN`) для генерации списка всех возможных комбинаций имен клиентов из таблицы `customer` и фамилий авторов из таблицы `author`.

9.5. Ответы

1.

```
SELECT
    oh.OrderID,
    p.PromotionCode
FROM orderheader oh
LEFT JOIN promotion p
    ON oh.PromotionID = p.PromotionID
WHERE oh.OrderDate >= '2019-01-01'
    AND oh.OrderDate < '2020-01-01';
```

2.

```
SELECT
    c.FirstName,
    c.LastName
FROM customer c
LEFT JOIN orderheader oh
    ON c.CustomerID = oh.CustomerID
    AND oh.OrderDate >= '2021-01-01'
    AND oh.OrderDate < '2022-01-01'
WHERE oh.CustomerID IS   NULL;
```

3.

```
SELECT
    c.FirstName,
    c.LastName,
    oh.OrderID,
    oh.OrderDate
FROM customer c
LEFT JOIN orderheader oh
    ON c.CustomerID = oh.CustomerID
    AND oh.OrderDate >= '2021-01-01'
    AND oh.OrderDate < '2022-01-01'
WHERE c.State = 'CA';
```

4.

```
SELECT
    c.FirstName,
    a.LastName
FROM customer c
CROSS JOIN author a;
```

10 Объединение запросов с помощью операций над множествами

В предыдущих главах мы рассматривали способы соединения таблиц на основе их логической взаимосвязи. Все написанные нами запросы включали лишь одну инструкцию `SELECT`. В этой главе вы узнаете, как составлять запросы с несколькими инструкциями `SELECT` и объединять их результаты в набор данных.

Такой подход особенно полезен, когда нужно вычислить результаты, требующие разных условий, — в частности, извлечь данные из разных таблиц, между которыми отсутствует ключ для их соединения. Несмотря на то что при использовании традиционных соединений (`JOIN`) строки, содержащие NULL-значения, исключаются из результирующего набора, с помощью теоретико-множественных операторов SQL можно включить такие значения в итоговую выборку, если они присутствуют в обоих наборах и вам необходимо их учесть.

10.1. Применение операторов над множествами

Мы составили массу запросов, начинающихся с `SELECT`, и каждый из них возвращал одну выборку. В этом и заключается назначение `SELECT`-запросов — получать набор результатов. Точнее, они формируют список строк, удовлетворяющих заданным условиям.

Тем не менее нередко возникает необходимость объединить или сравнить два и более результирующих набора. Для этого применяются специальные ключевые слова, именуемые *операторами над множествами* или *теоретико-множественными операторами* (set operators). Хотя в SQL таких операторов немного, все они задействуют один и тот же синтаксис для обработки двух результирующих наборов:

```
SELECT <some column>, <another column>
FROM <some table>
WHERE <some condition>
<set operator>
SELECT <some column>, <another column>
FROM <some table>
WHERE <another condition>;
```

Несмотря на то что в этом запросе мы применили две разные инструкции SELECT, оператор над множествами позволяет объединить их результаты в единый результирующий набор. Чаще всего теоретико-множественные операторы применяются именно для объединения данных, но, как мы увидим далее, их возможности гораздо шире. Правда, сначала необходимо запомнить несколько важных правил работы с операторами над множествами.

- Первое и самое простое правило: *количество столбцов в обоих запросах должно совпадать*. Как нам известно из введения в работу с таблицами, каждая строка в таблице обязана содержать набор значений из одинакового числа столбцов. Выборка, полученная при помощи оператора над множествами, не является исключением. Попытка обработать запросы с разным количеством столбцов приведет к ошибке.
- *Типы данных в соответствующих столбцах должны совпадать*. Мы еще не углублялись в тему типов данных, но уже видели, что существуют разные типы для чисел, текста и дат. При попытке объединить в одном результирующем наборе столбцы с несовместимыми типами данных система выдаст сообщение об ошибке.
- *В итоговом результирующем наборе задействуются имена столбцов из первого запроса*. Это означает, что можно объединять столбцы с разными именами, но их порядковые позиции в каждой инструкции SELECT должны совпадать. Если используются псевдонимы столбцов, то в выборке применяются лишь те из них, что указаны в первом блоке SELECT. Допускается добавлять псевдонимы и в последующие блоки SELECT — это не вызовет ошибки, однако помните: реляционная СУБД проигнорирует их, и они никоим образом не повлияют на итоговый результат. Запомните это правило — оно напрямую связано со следующим.
- *Предложение* ORDER BY *может присутствовать только в последней инструкции* SELECT. Ведь сортировка — это завершающий этап выполнения любого запроса, поэтому ORDER BY допускается лишь в последнем блоке SELECT. Если попытаться добавить сортировку в любую другую инструкцию SELECT, система выдаст сообщение об ошибке.

10.2. Оператор UNION

Наиболее распространенным теоретико-множественным оператором является `UNION`, который позволяет объединить результаты двух или более команд `SELECT` в единый результирующий набор, удаляя все дубликаты.

> **СОВЕТ** Одним из важнейших свойств оператора `UNION` является то, что он автоматически удаляет повторяющиеся строки из результирующего набора. Не забывайте об этом!

В качестве примера объединим имена всех людей в нашей базе данных `sqlnovel` в единый перечень имен. То есть мы можем выбрать имена и фамилии из таблиц `customer` и `author` и сформировать из них общий результирующий набор. Давайте выберем имена из таблиц и отсортируем их по фамилии и имени (результаты отображены на рис. 10.1):

```
SELECT FirstName, LastName
FROM customer
UNION
SELECT FirstName, LastName
FROM author
ORDER BY LastName, FirstName;
```

FirstName	LastName
Sandra	Calderon
Cora	Daly
Kevin	Daly
Robert	Davidson
Tara	Di Silvestro
Chris	Dixon
Jordan	Ericsson
Buck	Fernandez

Рис. 10.1. Фрагмент выборки (8 из 31 строки) имен и фамилий из таблиц customer и author

Результаты двух инструкций `SELECT` объединены в одну выборку и отсортированы, как показано на рис. 10.1. Для чего же мы употребляем слово `UNION`? Представьте, что вы объясняете наш запрос вслух: “I would like the first and last names from the customer table, and I would like to combine the results with the first and last names from the author table” («Мне нужны имя и фамилия из таблицы customer, и я хочу объединить эти результаты с именем и фамилией из таблицы author»).

Слово *combine* («объединять, комбинировать») вполне корректно описывает суть операции, однако, как мы увидим в дальнейшем, в SQL существует множество способов комбинировать данные. Можно по-разному объединять строки, столбцы, отдельные значения и целые результирующие наборы, поэтому слово

combine звучит слишком общо и не отражает специфику нашего запроса. Вместо него мы пользуемся словом *union*, обозначающим именно слияние двух или более наборов данных в единый результат.

В английском языке существительное *union* часто означает союз — к примеру, брак, в котором представители двух разных семей образуют одну новую. По аналогии можно представить оператор `UNION` как сочетание двух различных наборов данных, в результате которого рождается единый результирующий набор. Таким образом, слово *union* позволяет уточнить наше устное описание запроса: “I would like the first and last names from the customer table, and I would like to union the results with the first and last names from the author table” («Мне нужны имя и фамилия из таблицы customer, и я хочу сочетать эти результаты с именем и фамилией из таблицы author»).

Хотя изначально мы не можем определить, из какой именно таблицы — `customer` или `author` — получена та или иная строка, источник можно установить, добавив третий столбец с литералами, указывающими исходную таблицу для каждой строки. Мы добавим такие литералы в обе инструкции `SELECT`, при этом имя столбца необходимо задать только в первой из них (рис. 10.2). Имена же столбцов в выборке берутся исключительно из первой инструкции `SELECT`.

```
SELECT FirstName, LastName, 'customer'
     TableName
FROM customer
UNION
SELECT FirstName, LastName, 'author'
FROM author
ORDER BY LastName, FirstName;
```

FirstName	LastName	TableName
Sandra	Calderon	customer
Cora	Daly	customer
Kevin	Daly	customer
Robert	Davidson	author
Tara	Di Silvestro	customer
Chris	Dixon	customer
Jordan	Ericsson	customer
Buck	Fernandez	author

Рис. 10.2. Фрагмент выборки (8 из 31 строки) имен и фамилий из таблиц customer и author, наряду с третьим столбцом, где указана исходная таблица для каждой строки

Чаще всего `UNION` применяют, чтобы объединить в одной `SELECT`-инструкции различные, порой даже взаимоисключающие, условия фильтрации, в частности для получения значений из разных таблиц. Добавим условия фильтра по столбцу `LastName` из таблицы `customer` и по столбцу `FirstName` из таблицы `author` (результаты представлены на рис. 10.3):

```
SELECT FirstName, LastName, 'customer' TableName
FROM customer
WHERE LastName LIKE 'D%'
UNION
SELECT FirstName, LastName, 'author'
FROM author
WHERE FirstName LIKE 'C%'
ORDER BY LastName, FirstName;
```

FirstName	LastName	TableName
Cora	Daly	customer
Kevin	Daly	customer
Tara	Di Silvestro	customer
Chris	Dixon	customer
Chris	Walenski	author

Рис. 10.3. Выборка по полным именам покупателей, чья фамилия начинается на D, и авторам, чье имя начинается на C, отсортированная по фамилии и имени

На рис. 10.3 показано, что условиям фильтрации удовлетворяют всего пять строк, и среди них есть хотя бы по одной из каждой таблицы. Заметьте, что имя Chris и фамилия Daly встречаются дважды. Как отмечалось ранее в этой главе, `UNION` сам убирает полные дубликаты, то есть строки, где все значения совпадают.

В этом легко удостовериться, чуть изменив запрос. В выдаче на рис. 10.3 два раза встречается имя Chris — по одному разу из каждой таблицы. Уберем из результата столбцы `LastName` и `TableName`, поскольку именно они делают строки уникальными, несмотря на одинаковое значение `FirstName` (Chris). Без этих столбцов мы увидим имя Chris лишь в одной строке, так как `UNION` автоматически исключит повторы.

Помимо исключения этих столбцов, мы также изменим сортировку с `LastName` на `FirstName`. При использовании оператора над множествами, такого как `UNION`, сортировать результаты можно только по тем столбцам, которые присутствуют в инструкции `SELECT`. Поскольку мы убрали из нее `LastName`, то, оставив в запросе сортировку по `LastName`, мы получим ошибку: unknown column (неизвестный столбец). Вот наш новый запрос (результаты отображены на рис. 10.4):

```
SELECT FirstName
FROM customer
WHERE LastName LIKE 'D%'
UNION
SELECT FirstName
FROM author
WHERE FirstName LIKE 'C%'
ORDER BY FirstName;
```

FirstName
Chris
Cora
Kevin
Tara

Рис. 10.4. Выборка из таблицы customer имен покупателей, фамилии которых начинаются на D, объединенная посредством UNION с выборкой из таблицы author имен авторов, начинающихся на C. Две строки с именем Chris представлены одной строкой, поскольку UNION удалил повторяющуюся строку из результирующего набора

Но что, если мы не захотим удалять повторяющиеся строки? Если вместо этого нам потребуется включить в выборку все дубликаты? Тогда на помощь приходит другой теоретико-множественный оператор: `UNION ALL`.

10.3. Оператор UNION ALL

Оператор над множествами `UNION ALL` работает почти так же, как `UNION`, с той существенной разницей, что он не удаляет повторяющиеся строки. Вместо этого `UNION ALL` предписывает СУБД извлечь все данные, запрошенные в каждой инструкции `SELECT`, и вернуть результаты в том же виде, в каком они были прочитаны. Для демонстрации указанного различия заменим оператор над множествами `UNION` на `UNION ALL` в последнем запросе (результаты проиллюстрированы на рис. 10.5):

```
SELECT FirstName
FROM customer
WHERE LastName LIKE 'D%'
UNION ALL
SELECT FirstName
FROM author
WHERE FirstName LIKE 'C%'
ORDER BY FirstName;
```

FirstName
Chris
Chris
Cora
Kevin
Tara

Рис. 10.5. Выборка из таблицы customer имен покупателей, фамилии которых начинаются на D, объединенная посредством UNION ALL с выборкой из таблицы author имен авторов, начинающихся на C

На рис. 10.5 обе строки с именем Chris включены в выборку — ведь `UNION ALL` не удаляет повторяющиеся строки из результирующего набора. По этой причине применяемые с `UNION ALL` условия фильтра могут работать аналогично фильтрации в предложении `WHERE`. Сравните — результаты выполнения следующих двух запросов идентичны:

```
SELECT LastName
FROM customer
WHERE LastName = 'Daly'
UNION ALL
SELECT LastName
FROM customer
WHERE LastName = 'Dixon'
ORDER BY LastName;

SELECT LastName
FROM customer
WHERE LastName = 'Daly'
     OR LastName = 'Dixon'
ORDER BY LastName;
```

К практике!

Выполните два последних запроса и убедитесь в том, что оба возвращают одинаковый результат.

Оба запроса возвращают выборку из трех строк — две с именем Daly и одну с Dixon, — хотя и делают это разными способами. Так, первый запрос выполняет две инструкции `SELECT` для таблицы `customer` и затем объединяет результаты, причем каждый запрос ищет строки, удовлетворяющие одному условию. Тогда как второй запрос, содержащий одну инструкцию `SELECT`, ищет по нескольким условиям сразу.

СОВЕТ По мере вашего погружения в SQL важно не забывать: одну и ту же задачу можно решить разными путями, и в реальной практике один подход может оказаться значительно эффективнее другого. Например, как вы узнаете позже, в силу ряда причин запрос с `UNION ALL` иногда выполняется куда быстрее, чем аналогичный запрос с `OR`, несмотря на то что `UNION ALL` выполняет два отдельных SQL-запроса для извлечения тех же данных. Пока же запомните: если ваш запрос работает медленнее, чем ожидалось, всегда стоит рассмотреть альтернативные методы достижения того же результата.

Еще одно различие между `UNION` и `UNION ALL` состоит в том, что запросы с `UNION`, как правило, выполняются медленнее, чем с `UNION ALL`, поскольку СУБД приходится дополнительно обрабатывать данные для удаления дубликатов. Однако если результирующий набор `UNION ALL` оказывается очень большим и содержит множество повторяющихся строк, то запрос с `UNION` может оказаться быстрее — ведь в итоге по сети передается меньше данных. Имейте это в виду, когда пишете SQL-запросы, обрабатывающие сотни гигабайт и более, независимо от того, используете вы `UNION` или `UNION ALL`.

Наконец, с `UNION ALL` можно получить тот же результат, что и при помощи полного внешнего соединения (`FULL OUTER JOIN`), которое, однако, не поддерживается в MySQL, MariaDB и SQLite.

10.4. Эмуляция операции FULL OUTER JOIN в MySQL

Как я отмечал в заключительной части главы 9, полное внешнее соединение (`FULL OUTER JOIN`) возвращает не только строки, имеющие совпадения в обеих таблицах, но и все несовпадающие строки из обеих таблиц. Круговая диаграмма на рис. 10.6 визуализирует результат такого запроса для поиска общих промокодов между таблицами `promotion` и `orderheader`.

Вот как выглядел бы соответствующий SQL-запрос. В MySQL он не запустится, но сработает в других СУБД, поддерживающих `FULL OUTER JOIN`:

```
SELECT
    p.PromotionCode,
    oh.OrderID
FROM orderheader oh
FULL OUTER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
```

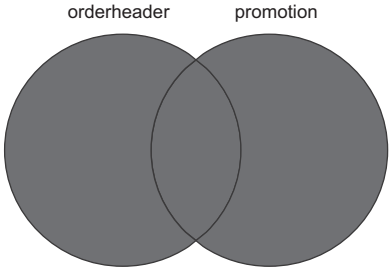


Рис. 10.6. Диаграмма Венна для значений, включаемых в запрос с полным внешним соединением (FULL OUTER JOIN) таблиц orderheader и promotion

Результаты `FULL OUTER JOIN` очень похожи на одновременное выполнение `LEFT JOIN` и `RIGHT JOIN`. Однако поскольку в MySQL это невозможно, мы можем эмулировать такое соединение, объединив левое и правое соединения посредством `UNION ALL`. В этом случае предпочтителен именно `UNION ALL`, так как он не удаляет повторяющиеся строки, а такие дубликаты могут существовать между таблицами и должны включаться в выборку, как это делает `FULL OUTER JOIN`.

Есть один важный нюанс: при эмуляции `FULL OUTER JOIN` описываемым способом необходимо изменить одно из соединений так, чтобы исключить общие значения. Если этого не сделать, в результате появятся повторяющиеся строки, поскольку и левое, и правое соединения включают общие значения, которые вернул бы `INNER JOIN`.

Ниже приведен пример корректной эмуляции `FULL OUTER JOIN` в MySQL при помощи `UNION ALL`. Чтобы избежать дублирования совпадающих строк, мы исключим их из второй инструкции `SELECT`, добавив условие `WHERE oh.PromotionID IS NULL`, — с таким приемом мы уже знакомились в главе 9:

```
SELECT
    p.PromotionCode,
    oh.OrderID
FROM orderheader oh
LEFT JOIN promotion p
    ON oh.PromotionID = p.PromotionID
UNION ALL
SELECT
    p.PromotionCode,
```

```
    oh.OrderID
FROM orderheader oh
RIGHT JOIN promotion p
    ON oh.PromotionID = p.PromotionID
WHERE oh.PromotionID IS NULL;
```

Данный запрос возвращает 53 строки, включающие:

- строки, совпадающие со значениями `PromotionID` в обеих таблицах;
- строки в таблице `orderheader`, не содержащие значений `PromotionID`;
- строки в таблице `promotion`, содержащие значения `PromotionID`, не задействованные в таблице `orderheader`.

Для наглядности полезно обратиться к диаграммам Венна, чтобы показать, какие именно действия выполняет каждая из двух инструкций `SELECT` в нашем запросе. Первый запрос находит первые два из трех наборов строк, приведенных выше: это строки, в которых значения `PromotionID` присутствуют и совпадают в обеих таблицах, а также строки из таблицы `orderheader`, где отсутствуют значения `PromotionID`. Эти данные представлены на диаграмме на рис. 10.7.

Второй запрос подхватывает третий набор — строки из таблицы `promotion`, содержащие значения `PromotionID`, не задействованные в таблице `orderheader`. Эту часть данных иллюстрирует диаграмма на рис. 10.8.

При помощи `UNION ALL` мы объединили все эти результаты в единый результирующий набор, как показано выше на рис. 10.6.

Эффективность теоретико-множественных операторов `UNION` и `UNION ALL` трудно переоценить. Тем не менее практически все реляционные СУБД поддерживают еще два оператора над множествами, о которых следует знать: `INTERSECT` и `EXCEPT`.

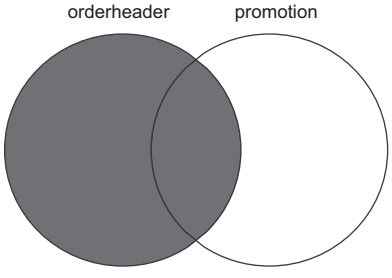


Рис. 10.7. Диаграмма Венна для значений, включаемых в запрос с левым внешним соединением (LEFT OUTER JOIN) таблиц orderheader и promotion

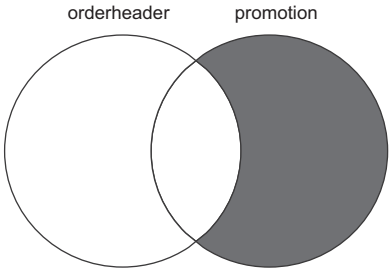


Рис. 10.8. Диаграмма Венна для значений, включаемых только в запрос к таблице promotion с правым внешним соединением (RIGHT OUTER JOIN) с таблицей orderheader

ВНИМАНИЕ MySQL не поддерживал операторы `INTERSECT` и `EXCEPT` до версии 8.0.31. При попытке применения этих операторов в более ранних версиях возникнут ошибки.

10.5. Оператор INTERSECT

Еще один полезный оператор над множествами — `INTERSECT`. Он способен возвращать результаты, похожие на запрос с `INNER JOIN`. Однако между ними есть два важных различия:

- `INNER JOIN` может возвращать повторяющиеся строки, тогда как `INTERSECT` — нет. В этом они сходны с парой `UNION ALL` и `UNION`.
- `INNER JOIN` никогда не возвращает значения null, потому что «ничто не может быть равно ничему». В противоположность этому, `INTERSECT` ищет общие значения в двух наборах данных, не проверяя их на равенство, поэтому в его результаты войдут и те NULL-значения, которые присутствуют в обоих запросах.

Чтобы лучше понять разницу, давайте вспомним, как работает внутреннее соединение. Следующий запрос демонстрирует применение оператора `INNER JOIN` для выявления значений `PromotionID`, присутствующих одновременно в таблицах `orderheader` и `promotion`:

```
SELECT
    oh.PromotionID
FROM orderheader oh
INNER JOIN promotion p
    ON oh.PromotionID = p.PromotionID;
```

Напиши мы этот запрос для СУБД, поддерживающей `INTERSECT`, он имел бы следующий вид:

```
SELECT
    PromotionID
FROM orderheader
INTERSECT
SELECT
    PromotionID
FROM promotion;
```

Несмотря на то что в примере использовался единственный столбец, оператор `INTERSECT` допускает применение нескольких атрибутов в инструкциях `SELECT`. Хотя наименования столбцов могут различаться, все запросы должны содержать идентичное количество столбцов, а порядок их следования должен быть единообразным для корректной обработки оператором `INTERSECT`.

10.6. Оператор EXCEPT

Еще один теоретико-множественный оператор, часто поддерживаемый другими СУБД, позволяет получить данные из одного набора, отсутствующие во втором. Оператор `EXCEPT` функционально эквивалентен методу, рассмотренному ранее при анализе левых внешних соединений (`LEFT JOIN`) в главе 9.

> **НА ЗАМЕТКУ** СУБД Oracle не поддерживает оператор `EXCEPT`, но предоставляет оператор `MINUS`, полностью идентичный по семантике и синтаксису.

Допустим, нам нужно найти все значения `PromotionID` из таблицы `promotion`, которые не востребованы ни в одном заказе из таблицы `orderheader`, как показано на диаграмме рис. 10.9. Мы уже знаем, что их можно получить посредством такого запроса:

```
SELECT
    p.PromotionID
FROM promotion p
LEFT JOIN orderheader oh
    ON p.PromotionID = oh.PromotionID
WHERE oh.PromotionID IS NULL;
```

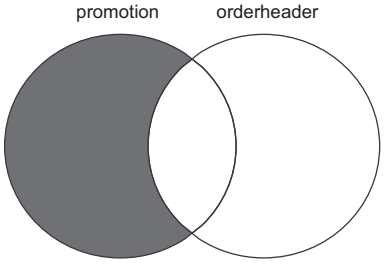


Рис. 10.9. Диаграмма Венна для значений, включаемых только в запрос к таблице promotion с левым внешним соединением (LEFT OUTER JOIN) с таблицей orderheader

Тот же результат, что дает LEFT JOIN с отбором NULL-значений, можно сформировать с применением оператора EXCEPT:

```
SELECT
    PromotionID
FROM promotion
EXCEPT
SELECT
    PromotionID
FROM orderheader;
```

Как и в случае с INTERSECT, результаты EXCEPT демонстрируют два существенных отличия от результатов LEFT JOIN:

- LEFT JOIN возвращает повторяющиеся значения, а EXCEPT — нет;
- LEFT JOIN никогда не возвращает значения null, потому что «ничто» не может равняться «ничему». Поскольку EXCEPT ищет общие значения между двумя наборами данных, а не проверяет равенство, результаты EXCEPT также будут включать все NULL-значения, присутствующие в результатах первого запроса и отсутствующие во втором.

Несмотря на то что INTERSECT и EXCEPT используются гораздо реже, чем INNER JOIN и LEFT JOIN, помните о них при работе с СУБД, поддерживающей эти ключевые слова, особенно когда нужно включить в выборку пустые значения.

На этом мы завершим знакомство с теоретико-множественными операторами. В следующей главе мы рассмотрим другие способы соединения таблиц и наборов данных посредством логических операторов.

10.7. Практическое занятие

1. Из этой главы вы узнали, что имена столбцов в запросах с UNION и UNION ALL берутся из первого предложения SELECT. Каков будет результат выполнения запроса, в котором последний столбец первого запроса останется безымянным? Выясните это, запустив следующий SQL-код:

```
SELECT FirstName, LastName, 'customer'
FROM customer
UNION
SELECT FirstName, LastName, 'author' TableName
FROM author
ORDER BY LastName, FirstName;
```

2. Учитывая, что в таблице customer есть строки для покупателей Cora Daly и Kevin Daly, будут ли результаты следующих двух запросов идентичными? Если нет, то какие будут различия?

```
SELECT LastName
FROM customer
WHERE FirstName = 'Cora'
OR FirstName = 'Kevin';

SELECT LastName
FROM customer
WHERE FirstName = 'Cora'
UNION
SELECT LastName
FROM customer
WHERE FirstName = 'Kevin';
```

3. Мы выяснили, чем `UNION` отличается от `UNION ALL`, но ни разу не применяли их в одном и том же запросе. И лучше этого не делать! Во многих реляционных СУБД такая попытка просто выдаст ошибку. Но даже если ошибки не возникнет, результаты могут оказаться непредсказуемыми. Чтобы убедиться в этом, запустите два следующих запроса. Заметьте, что их результаты различаются. Как вы думаете, почему так происходит?

```
SELECT LastName
FROM customer
WHERE FirstName = 'Cora'
UNION
SELECT LastName
FROM customer
WHERE FirstName = 'Kevin'
UNION ALL
SELECT LastName
FROM customer
WHERE LastName = 'Daly';

SELECT LastName
FROM customer
WHERE LastName = 'Daly'
UNION ALL
SELECT LastName
FROM customer
WHERE FirstName = 'Kevin'
UNION
SELECT LastName
FROM customer
WHERE FirstName = 'Cora';
```

10.8. Ответы

1. Поскольку в первом предложении `SELECT` не задано имя для последнего столбца, в результирующем наборе СУБД автоматически присваивает ему имя — то самое литеральное значение `'customer'`, которое мы туда подставили.

2. Результаты не будут одинаковыми. Первый запрос, задействующий для фильтрации `OR`, возвращает две строки — по одной для каждого совпадения. Второй запрос с использованием `UNION` возвращает лишь одну строку, так как `UNION` удаляет из результатов дубликаты.
3. Первый запрос возвращает три строки, а второй — только одну. Результаты могут показаться неожиданными, ведь во втором запросе просто поменяли местами команды `SELECT`.

   Ответ кроется в очередности выполнения (приоритете): операторы обрабатываются в том порядке, в котором они указаны. Сначала применяется первый `UNION` или `UNION ALL`, а затем — следующий. Не забывайте, что `UNION` удаляет дубликаты, а `UNION ALL` — нет.

   В первом запросе каждый из первых двух `SELECT` возвращает по одной строке со значением `'Daly'`, и поскольку используется `UNION`, дубликаты удаляются — на этом этапе остается одна строка. Однако третий `SELECT` возвращает две строки, и при объединении с предыдущим результатом посредством `UNION ALL` итоговый набор содержит три строки: одна строка от первых двух `SELECT` и две строки от последнего `SELECT`.

   Во втором запросе сначала посредством `UNION ALL` объединяются первый `SELECT` (возвращающий две строки) и второй `SELECT` (возвращающий одну строку). Выполни мы только эту часть запроса, выборка содержала бы три строки. Но затем к результату добавляется еще один `SELECT`, возвращающий одну строку, и здесь мы задействуем `UNION`, который удаляет все дубликаты из итогового набора. Поэтому второй запрос возвращает всего одну строку.

   Если это объяснение кажется вам запутанным, не беда. Главное — избегать сочетания `UNION` и `UNION ALL` в одном запросе, и тогда вам никогда не придется сталкиваться с подобной головоломкой.

11 Применение подзапросов и логических операторов

В предыдущей главе мы расширили наш понятийный аппарат работы с SQL: помимо прямых запросов к таблицам было показано, как при помощи теоретико-множественных операторов, таких как `UNION` и `INTERSECT`, объединять результаты нескольких команд `SELECT` в единое результирующее множество. В этой главе мы пойдем еще дальше и рассмотрим один из ключевых механизмов языка SQL — подзапросы, позволяющие в рамках одного запроса выполнять вложенные выборки и осуществлять многоуровневую обработку данных.

Подзапросы (subqueries) — это запросы, вложенные в другой запрос (поэтому их также называют *вложенными* или *внутренними запросами*). Они применяются в тех случаях, когда требуемый результат невозможно получить при помощи одной команды `SELECT`. Вместо того чтобы составлять два или более отдельных запроса, мы объединяем их в один. Не переживайте — этот процесс не так сложен, как может показаться.

К концу главы вы увидите, как подзапросы позволяют вычислять результаты команд `SELECT` способами, недоступными операторам над множествами, которые мы рассматривали в предыдущих главах. Существует множество способов применения подзапросов, так что пора без промедления приступить к знакомству с ними.

11.1. Простой подзапрос

Как я уже не раз отмечал в книге, команда `SELECT` — это разновидность SQL-запроса, возвращающего набор данных, именуемый *выборкой*, *результирующим множеством* или *набором* (result set). Вы уже выполнили десятки запросов,

создающих такие наборы. Вы запрашивали данные из одной или нескольких таблиц, и полученные результаты выглядели почти как обычные таблицы: они состояли из строк и столбцов, а столбцы имели имена.

Разовьем подобный сценарий. Поскольку выборка по запросу представляет собой результирующий набор, по структуре схожий с таблицей, мы можем обрабатывать его теми же многочисленными способами, что и данные в реальной таблице. Это означает, что к таким результатам можно применять соединения и фильтры точно так же, как к самим таблицам. Для этого и нужны подзапросы. Достаточно теории — разберем конкретный пример.

Предположим, мы хотим найти идентификатор и дату для всех заказов, сделанных после конкретного заказа, размещенного покупателем по имени Маргарет Монтойя (Margaret Montoya). Мы выбрали ее, так как она сделала лишь один заказ. Если бы нам захотелось сформулировать этот запрос в разговорной форме, мы могли бы сказать следующее: “I would like the order ID, customer ID, and order date from the customer table, but I want only the orders placed after the one order placed by the customer named Margaret Montoya” («Мне нужны идентификатор заказа, идентификатор клиента и дата заказа из таблицы customer, однако меня интересуют лишь те заказы, которые были оформлены после единственного заказа, размещенного покупателем по имени Маргарет Монтойя»).

Это наша первая команда, состоящая из двух простых предложений в составе сложного. На основе изученного материала для получения нужных результатов мы могли бы написать два отдельных запроса. Первый запрос для поиска заказа, сделанного Маргарет Монтойей, мог бы выглядеть так:

```
SELECT
    oh.OrderID,
    oh.OrderDate
FROM orderheader oh
INNER JOIN customer c
    ON oh.CustomerID = c.CustomerID
WHERE c.FirstName = 'Margaret'
    AND c.LastName = 'Montoya';
```

Результат запроса показывает, что Маргарет Монтойя разместила лишь один заказ 23 апреля 2021 года. Зная эту дату, мы можем применить ее во втором запросе, чтобы получить результаты, приведенные на рис. 11.1. SQL-код запроса может быть следующим:

```
SELECT
    OrderID,
    CustomerID,
    OrderDate
FROM orderheader
WHERE OrderDate > '2021-04-23';
```

OrderID	CustomerID	OrderDate
1044	19	2021-06-06 00:00:00
1045	11	2021-10-01 00:00:00
1046	4	2021-11-13 00:00:00
1047	19	2021-11-28 00:00:00
1050	1	2022-03-10 00:00:00

Рис. 11.1. Значения OrderID, CustomerID и OrderDate для всех заказов, размещенных после покупки, сделанной Маргарет Монтойей 23 апреля 2021 года

Написание двух запросов для получения этого результата представляется нерациональным и слишком громоздким, поэтому дату, жестко заданную в программном коде, мы заменим подзапросом. Для этого достаточно заменить дату заказа (`'2021-04-23'`), явно указанную в коде, первым запросом, который теперь становится подзапросом, и заключить его в круглые скобки.

> **ВНИМАНИЕ** При фильтрации с помощью подзапроса можно выбрать только один столбец, поскольку SQL позволяет нам обрабатывать лишь одно значение или набор значений за раз в предложении `WHERE`. Выбор более одного столбца в подзапросе приведет к ошибке выполнения.

Вот как выглядит инструкция `SELECT` с подзапросом, выводящая те же результаты, что и два предыдущих запроса:

```
SELECT
    OrderID,
    CustomerID,
    OrderDate
FROM orderheader
WHERE OrderDate > (
    SELECT
        oh.OrderDate
    FROM orderheader oh
    INNER JOIN customer c
        ON oh.CustomerID = c.CustomerID
    WHERE c.FirstName = 'Margaret'
        AND c.LastName = 'Montoya'
     );
```

Результаты запроса совпадают с результатами, показанными на рис. 11.1, поскольку мы объединили логику двух запросов в одном.

Подзапрос оформлен с размещением ограничивающих скобок на разных строках. Хотя такой способ форматирования не является обязательным, он имеет свои преимущества. Во-первых, это упрощает чтение кода по сравнению с размещением скобок в начале и в конце подзапроса. И во-вторых, что, возможно, еще важнее, такое форматирование облегчает выделение подзапроса курсором и его отдельное выполнение. При написании SQL-кода этот прием удобен для проверки выдачи подзапросом ожидаемых результатов.

К практике!

Выполните приведенный выше подзапрос. Затем выделите и выполните только строки подзапроса внутри скобок, чтобы убедиться в том, что возвращаемое значение равно `2021-04-23`.

Здесь в сочетании с подзапросом мы применили оператор сравнения `>`, однако другие операторы сравнения, такие как `=` и `<>`, имеют ограничение — они могут сравнивать лишь одно значение. Чтобы раскрыть весь потенциал подзапросов, необходимо задействовать совершенно другой набор ключевых слов, известных как *логические операторы*.

11.2. Логические операторы и подзапросы

Логические операторы в чем-то похожи на операторы сравнения, поскольку они проверяют, является ли некоторое условие истинным, ложным или неизвестным. Вам уже знаком ряд таких операторов: `IN`, `NOT IN`, `BETWEEN` и `LIKE`. Некоторые операторы сравнения не способны вычислять результирующий набор с более чем одной строкой, поэтому давайте рассмотрим пример подзапроса, возвращающего несколько строк.

Заказ `OrderID 1034` включает более одной книги, как отражено на рис. 11.2. Для поиска книг, входящих в этот заказ, можно написать SQL-запрос следующего вида:

```
SELECT
    t.TitleName
FROM title t
INNER JOIN orderitem oi
    ON oi.TitleID = t.TitleID
WHERE OrderID = 1034;
```

TitleName
The Join Luck Club
Catcher in the Try
Anne of Fact Tables
The DateTime Machine

Рис. 11.2. Четыре книги, включенные в заказ 1034

Перепишем этот SQL-код, преобразовав часть запроса, которая производит отбор по `OrderID`, в отдельный запрос и поместив эту часть в предложение `WHERE` в качестве вложенного запроса, а затем при помощи оператора `=` отфильтруем по условию, где значение `TitleID` из таблицы `title` равно результатам нашего подзапроса:

```
SELECT
    t.TitleName
FROM title t
WHERE TitleID = (
    SELECT TitleID
    FROM orderitem
    WHERE OrderID = 1034
    );
```

Увы, наш запрос не сработает. При попытке его выполнения в панели вывода отобразится сообщение об ошибке Subquery returns more than 1 row («Подзапрос возвращает более одной строки»). И это закономерно — ведь мы же знаем, что в заказ входило четыре книги, и следовательно, подзапрос возвращает четыре строки. Наш подзапрос не может быть обработан оператором сравнения =, поскольку он пытается определить, равен ли каждый TitleID в таблице title одному-единственному значению. Наш запрос был бы выполнен, будь в заказе всего одна книга, но их четыре, и, к сожалению, оператор = не может с ними справиться. Для подобного сценария нам потребуется первый логический оператор — ANY.

К практике!

Выполните последний запрос и удостоверьтесь в том, что на панели вывода отображается ошибка.

11.2.1. Операторы ANY и IN

Логический оператор ANY оценивает набор значений, чтобы определить, равно ли любое из них искомым значениям, — отсюда и название ANY («любой»). Представьте, что ANY «расширяет» возможности обычного оператора = (или любого другого оператора сравнения), разрешая включать в подзапрос множественные значения. Чтобы «починить» наш запрос, просто добавьте ANY после = в предложении WHERE:

```
SELECT
     t.TitleName
FROM title t
WHERE TitleID = ANY (
    SELECT TitleID
    FROM orderitem
    WHERE OrderID = 1034
     );
```

Результаты этого запроса совпадают с выборкой, представленной на рис. 11.2. Если перевести его на обычный язык, он прозвучит так: “I would like the title name from the title table, and I would like the titles to match any of the titles from order 1034” («Мне нужно название книги из таблицы title, и я хочу, чтобы эти названия соответствовали любой из книг заказа 1034»).

НА ЗАМЕТКУ Большинство реляционных СУБД, включая MySQL, также поддерживают логический оператор `SOME`, функционально эквивалентный оператору `ANY`. Однако применяется он гораздо реже, чем `ANY`.

Можно получить те же результаты, заменив `ANY` на другой логический оператор — ключевое слово `IN`. В этом случае наше намерение можно перефразировать так: "I would like the title name from the title table, and I would like the titles to be in titles from order 1034" («Мне нужно название книги из таблицы title, и я хочу, чтобы эти названия входили в набор книг из заказа 1034»). Тогда наш запрос примет вид:

```
SELECT
    t.TitleName
FROM title t
WHERE TitleID IN (
    SELECT TitleID
    FROM orderitem
    WHERE OrderID = 1034
     );
```

Поскольку оба логических оператора могут применяться с подзапросами и давать эквивалентные результаты, возникает вопрос: какой из них выбрать? Ответ зависит от того, нужно ли вам применять оператор сравнения. Если требуется задействовать операторы сравнения (`>`, `>=`, `<` или `<=`), то следует выбрать `ANY`, так как `IN` не поддерживает такие сравнения. Однако если нужна только проверка на равенство, стоит предпочесть `IN` — он используется в подобных подзапросах значительно чаще, чем `= ANY`. А теперь взглянем на противоположный способ фильтрации — когда нужно не включить, а исключить результаты подзапроса.

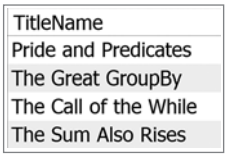
11.2.2. Операторы ALL и NOT IN

А теперь допустим, что нам нужно найти названия книг, которые *не* входят в заказ 1034, как показано на рис. 11.3. Мы могли бы сказать: "I would like the title name from the title table, and I would like the titles to not be in titles from order 1034" («Мне нужно название книги из таблицы title, и я хочу, чтобы эти названия не входили в названия книг из заказа 1034»). Добавим частицу *not* («не») из нашей устной формулировки в SQL-код:

```
SELECT
    t.TitleName
FROM title t
WHERE TitleID NOT IN (
    SELECT TitleID
    FROM orderitem
    WHERE OrderID = 1034);
```

В базе данных `sqlnovel` содержатся восемь книг, а заказ 1034 включает четыре из них, как отображено на рис. 11.2. Теперь нам известны оставшиеся четыре

книги, не вошедшие в этот заказ. Наш SQL-запрос сначала выбирает все наименования из заказа 1034, а затем находит в таблице `title` книги, отсутствующие в конкретном заказе.



TitleName
Pride and Predicates
The Great GroupBy
The Call of the While
The Sum Also Rises

Рис. 11.3. Четыре книги, не включенные в заказ 1034

Это подводит нас к еще одной словесной формулировке запроса: “I would like the title name from the title table, and I would like the titles to not match any of the titles from order 1034” («Мне нужны названия книг из таблицы title, и я хочу, чтобы эти названия не совпадали ни с одной из книг в заказе 1034»). На первый взгляд кажется, что для получения такого результата можно воспользоваться оператором `ANY` в сочетании с оператором неравенства (`<>`):

```
SELECT
    t.TitleName
FROM title t
WHERE TitleID <> ANY (
    SELECT TitleID
    FROM orderitem
    WHERE OrderID = 1034
    );
```

Запрос выполнится без ошибок, но не даст нужного результата. Он проверяет все названия книг в таблице `title`, чтобы определить, какие из них не совпадают ни с одним из названий в подзапросе. Поскольку наш подзапрос возвращает несколько значений, каждое название из таблицы `title` окажется «неподходящим» — ведь хотя бы одно из значений в подзапросе будет от него отличаться.

К практике!

Выполните этот запрос и обратите внимание на то, что он возвращает все названия из таблицы `title`.

Для получения желаемого результата необходимо прибегнуть к оператору `ALL` вместо `ANY`, раз нам нужны названия книг, не совпадающие со *всеми* значениями в подзапросе:

```
SELECT
    t.TitleName
FROM title t
WHERE TitleID <> ALL (
```

```
    SELECT TitleID
    FROM orderitem
    WHERE OrderID = 1034
    );
```

Результат выполнения запроса представлен на рис. 11.3, и теперь он соответствует нашим ожиданиям. Как и в случае с `IN` и `ANY`, выбор между `NOT IN` и `ALL` продиктован необходимостью применения оператора сравнения.

> **НА ЗАМЕТКУ** Мы еще не касались этой темы, но имейте в виду: используя подзапросы, вы заставляете СУБД одновременно выполнять два запроса и сопоставлять результаты одного с другим. Как правило, подзапросы требуют больше вычислительных ресурсов и памяти, поэтому при написании SQL-кода следует хорошенько подумать: а нельзя ли обойтись без него?

11.2.3. Операторы EXISTS и NOT EXISTS

Существуют еще два оператора, которые могут сделать подзапросы эффективнее рассмотренных ранее: `EXISTS` и `NOT EXISTS`. Их преимущество в том, что они не проверяют значения каждой строки подзапроса, а лишь устанавливают наличие хотя бы одной подходящей строки. Как только найдено первое совпадение, дальнейшая проверка на равенство или неравенство прекращается.

Начнем с оператора `EXISTS`. Он применяется аналогично ранее рассмотренным `= ANY` и `IN` для поиска названий, включенных в заказ 1034. Отличие заключается в том, что при использовании `EXISTS` в предложении `WHERE` подзапроса необходимо прописать связь между таблицами, своего рода соединение. Вот как выглядит такой запрос:

```
SELECT
    t.TitleName
FROM title t
WHERE EXISTS (
    SELECT TitleID
    FROM orderitem oi
    WHERE OrderID = 1034
        AND t.TitleID = oi.TitleID
    );
```

Выполнение запроса дает результат, отображенный на рис. 11.2: возвращаются названия всех книг из заказа 1034. Обратите внимание, что мы задействовали `EXISTS` в предложении `WHERE` основного запроса, а в предложении `WHERE` вложенного запроса появилась дополнительная строка: `AND t.TitleID = oi.TitleID`. Именно здесь происходит сопоставление значений между основным запросом и подзапросом. Поскольку проверка на совпадение выполняется именно в этом условии, содержимое предложения `SELECT` в подзапросе не имеет значения.

По этой причине в стороннем SQL-коде вам нередко будут встречаться подзапросы с EXISTS, в которых предложение SELECT содержит формальное выражение, не несущее смысловой нагрузки — к примеру, просто SELECT 1:

```
SELECT
    t.TitleName
FROM title t
WHERE EXISTS (
    SELECT 1
    FROM orderitem oi
    WHERE OrderID = 1034
        AND t.TitleID = oi.TitleID
    );
```

К практике!

Выполните последний запрос и убедитесь в том, что он возвращает выборку, представленную на рис. 11.2. Попробуйте заменить 1 в предложении SELECT подзапроса на любое значение — и увидите, что оно никоим образом не влияет на результат.

Кроме того, для поиска книг, не входящих в заказ 1034, можно задействовать NOT EXISTS. Выполнение следующего запроса возвращает результаты, показанные на рис. 11.3:

```
SELECT
    t.TitleName
FROM title t
WHERE NOT EXISTS (
    SELECT 1
    FROM orderitem oi
    WHERE OrderID = 1034
        AND t.TitleID = oi.TitleID
    );
```

И вновь преимущества применения EXISTS или NOT EXISTS с подзапросами обнаруживаются при работе с большими наборами данных, поскольку эти операторы демонстрируют повышенную производительность по сравнению с IN / NOT IN, ANY и ALL.

11.3. Подзапросы в других частях запроса

До сих пор в этой главе мы рассматривали лишь подзапросы, применяемые для отбора в предложении WHERE, но этим их возможности не ограничиваются! Их можно включать и в другие разделы SQL-запроса.

11.3.1. Подзапросы в предложении FROM

Существует возможность составить запрос, возвращающий результаты, показанные на рис. 11.2, при помощи соединения в предложении FROM. Для этого мы

перемещаем подзапрос в соединение в предложении FROM, применяя внутреннее соединение. Нам не нужны операторы, так как мы не задействуем сам вложенный запрос для фильтрации, а просто подключаем результаты подзапроса, тогда как вся оценка происходит в части ON соединения.

И еще: поскольку наш подзапрос не имеет имени для обозначения итоговой выборки, необходимо воспользоваться псевдонимом для обеспечения возможности соединения с другой таблицей:

```
SELECT
    t.TitleName
FROM title t
INNER JOIN (
    SELECT TitleID
    FROM orderitem
    WHERE OrderID = 1034
    ) oisq
    ON t.TitleID = oisq.TitleID;
```

Переместив подзапрос в предложение FROM, мы расцениваем его результаты как таблицу, в которой результаты соединены с таблицей `title` по полю `TitleID`. На деле результаты подзапроса не являются таблицей, но СУБД должна вычислить их до того, как можно будет определить, какие строки из результатов вложенного запроса могут быть соединены с таблицей `title`.

Примечателен факт, что, поскольку подзапрос находится в предложении FROM, с ним можно работать в запросе как с таблицей. В результате мы больше не ограничены одним столбцом в подзапросе и вольны включать дополнительные столбцы, если это необходимо для целей соединения или отбора.

К практике!

Выполните последний запрос, замените SELECT TitleID на SELECT TitleID, OrderID и повторно выполните отредактированный код.

Кроме того, допустимо задействовать соединение в предложении FROM для поиска значений, которых нет в подзапросе, применив метод LEFT OUTER JOIN из главы 9. Мы можем применить этот тип соединения с фильтрацией по пустым значениям во втором наборе данных, которым является вложенный запрос, чтобы найти значения, существующие в первой таблице, но отсутствующие в присоединенной таблице:

```
SELECT
    t.TitleName
FROM title t
LEFT JOIN (
    SELECT TitleID
    FROM orderitem
WHERE OrderID = 1034
```

```
    ) oisq
    ON t.TitleID = oisq.TitleID
WHERE oisq.TitleID IS NULL;
```

Результаты запроса те же, что показаны на рис. 11.3.

11.3.2. Подзапросы в предложении SELECT

Последний способ применения подзапросов — в предложении `SELECT`. Можно получить те же названия книг (`TitleName`), что приведены на рис. 11.2, включив вложенный запрос в предложение `SELECT`, хотя для этого нам придется перестроить запрос, перенеся подзапрос из условия фильтра по `orderitem` в выборку `TitleName` из таблицы `title`:

```
SELECT
    (
    SELECT TitleName
    FROM title t
    WHERE t.TitleID = oi.TitleID
    ) AS TitleName
FROM orderitem oi
WHERE oi.OrderID = 1034;
```

> **ВНИМАНИЕ** Такой способ получения результата — скорее диковинка, чем рекомендация. Я привожу этот пример лишь для того, чтобы продемонстрировать, как выглядит вложенный запрос в предложении `SELECT`. Размещение подзапросов в предложении `SELECT` редко бывает наилучшим решением при написании SQL-запросов.

Итак, мы рассмотрели несколько способов применения вложенных запросов и возможности, которые они открывают при написании SQL-кода. Однако помните: подзапросы следует использовать с осторожностью, поскольку они зачастую отрицательно влияют на производительность. Дело в том, что в каждом вложенном запросе выполняется дополнительный `SELECT`, а значит, такие SQL-выражения создают дополнительную вычислительную нагрузку на базу данных.

В главе 12 мы обратимся к способам группировки наборов данных для вычисления агрегатных значений, таких как минимум и максимум. Но сначала попрактикуемся в написании подзапросов.

11.4. Практическое занятие

1. Напишите запрос с применением подзапроса и оператора `IN`, чтобы получить названия книг из единственного заказа, сделанного Джо Пажено (Joe Pagenaud).
2. Вернитесь к примерам из раздела 11.1, где мы пытались найти заказы, размещенные после заказа Маргарет Монтойи (Margaret Montoya). Напишите

аналогичный запрос, чтобы найти идентификатор заказа, идентификатор клиента и дату заказа для всех заказов, сделанных после всех заказов Коры Дейли (Cora Daly).

3. Известно, что деление 1 на 0 возвращает значение NULL. Скажите, можно ли в первом запросе раздела 11.2.3 заменить конструкцию `SELECT TitleID` в предложении `SELECT` подзапроса на `SELECT 1/0` и при этом получить правильные результаты?

11.5. Ответы

1. Решить эту задачу можно разными способами — все зависит от того, какие именно запросы вы решите объединить в подзапросе. Один из возможных вариантов выглядит так:

```
SELECT
    t.TitleName
FROM title t
INNER JOIN orderitem oi
    ON t.TitleID = oi.TitleID
WHERE oi.OrderID IN (
    SELECT
    oh.OrderID
    FROM orderheader oh
    INNER JOIN customer c
        ON oh.CustomerID = c.CustomerID
    WHERE c.FirstName = 'Joe'
        AND c.LastName = 'Pagenaud'
    );
```

2. Формулировка запроса может варьироваться, но вот один из способов найти нужную информацию о заказе:

```
SELECT
    OrderID,
    CustomerID,
    OrderDate
FROM orderheader
WHERE OrderDate > ALL (
    SELECT
        oh.OrderDate
    FROM orderheader oh
    INNER JOIN customer c
        ON oh.CustomerID = c.CustomerID
WHERE c.FirstName = 'Cora'
    AND c.LastName = 'Daly'
    );
```

3. Ответ утвердительный, поскольку операторы `EXISTS` и `NOT EXISTS` игнорируют значение или столбец, указанные в предложении `SELECT` подзапроса.

12 Группировка данных

Тем, кто привык работать с электронными таблицами, а не с реляционными базами данных, последние три главы могли показаться относительно сложными. Ведь в таблицах вы, как правило, имеете дело с одним набором данных, а не с несколькими. Если понятия, описанные в той части руководства, были для вас новы, то теперь вы можете облегченно вздохнуть: в этой главе рассматриваются темы, которые должны быть хорошо знакомы большинству пользователей электронных таблиц.

Одно из главных достоинств электронных таблиц — возможность оперативно производить математические операции над диапазоном данных. Скажем, чтобы найти сумму всех значений в столбце, достаточно нажать кнопку «Автосумма», и нужная сумма появится в указанной ячейке. При выделении этой ячейки видно, что таблица применила функцию `SUM` к указанному диапазону ячеек. `SUM` — это *функция*, то есть команда, выполняющая предварительно заданное вычисление.

Разумеется, в языке SQL нет кнопки для автоматического подсчета сумм, но зато есть функции, подобные `SUM`, позволяющие производить математические расчеты. Более того, в сравнении с электронными таблицами реляционные базы данных обеспечивают значительно большую гибкость при выполнении подобных операций.

12.1. Агрегатные функции

В оставшейся части книги нас ждет знакомство с различными типами SQL-функций. *Функция* (function) — это ключевое слово, позволяющее с легкостью выполнять вычисления или другие действия. SQL располагает богатым арсеналом функций для получения различных значений, от преобразования дат до форматирования данных.

Эта глава посвящена основным *агрегатным функциям* (aggregate function), которые производят базовые вычисления над диапазоном данных в столбце. Без них не обойтись, когда требуется осуществить подобного рода операции по множеству записей.

12.1.1. Функция SUM

Начнем с простейшей функции — `SUM`, возвращающей сумму всех значений в столбце данных. К примеру, чтобы выяснить общее число заказанных книг, достаточно просуммировать столбец `Quantity` в таблице `orderitem`. Мы могли бы выразить это намерение обычным языком, прибегнув к слову *sum* («сумма»): "I would like the sum of the quantity of titles in the orderitem table" («Мне нужна сумма количества книг в таблице `orderitem`»). Это описание довольно близко к нашему SQL-запросу (см. выборку по запросу на рис. 12.1):

```
SELECT SUM(Quantity)
FROM orderitem;
```

SUM(Quantity)
70

Рис. 12.1. Количество всех заказов отображено в столбце без псевдонима. Имя столбца по умолчанию соответствует агрегированному выражению

Прежде чем двигаться дальше, обратите внимание на ряд существенных моментов. Во-первых, при использовании функции `SUM` необходимо заключать имя столбца в круглые скобки. Без скобок запрос завершится синтаксической ошибкой.

Во-вторых, выполнив запрос, вы заметите, что имя результирующего столбца будет совпадать с самим выражением агрегатной функции, что не всегда удобно. В таких случаях принято задавать понятное имя столбца при помощи псевдонима, что повышает информативность и удобочитаемость выборки. Если вы захотите повторно выполнить этот запрос, рекомендую отредактировать его, добавив псевдоним, отражающий суть агрегатного вычисления:

```
SELECT SUM(Quantity) AS TotalQuantity
FROM orderitem;
```

ВНИМАНИЕ Имейте в виду, что функция `SUM` предназначена только для работы с числовыми значениями. Попытка применить эту функцию к датам или символьным значениям может привести к семантически некорректным результатам.

СОВЕТ При использовании псевдонима для выходного столбца избегайте имен, совпадающих с названиями столбцов таблицы. Это не только сбивает с толку читающих результаты выборки, но и недопустимо в некоторых реляционных СУБД.

12.1.2. Функция COUNT

Агрегатная функция SUM предназначена для вычисления суммарного значения всех элементов, а для определения количества непустых значений в столбце следует задействовать другую функцию — COUNT. Функция COUNT подсчитывает число строк в столбце. Это утверждение может показаться довольно очевидным, но обратите внимание на одну важную особенность агрегатных функций: по умолчанию они исключают значения null.

Возьмем таблицу orderheader и выясним, сколько в ней строк. Это легко сделать, выполнив запрос с функцией COUNT для всех OrderID — ведь в каждой строке это поле содержит значение. Результат запроса приведен на рис. 12.2.

```
SELECT COUNT(OrderID) AS TotalOrders
FROM orderheader;
```

TotalOrders
50

Рис. 12.2. Все 50 строк в таблице orderheader имеют значение OrderID, следовательно, это и есть общее число строк в таблице

По результатам запроса в таблице orderheader содержится 50 строк, что верно. Однако если мы попытаемся подсчитать количество активированных промокодов, выбрав COUNT для столбца PromotionID, мы получим другой результат (рис. 12.3):

```
SELECT COUNT(PromotionID) AS TotalOrdersWithPromotionCode
FROM orderheader;
```

TotalOrdersWithPromotionCode
20

Рис. 12.3. Только 20 строк в таблице orderheader имеют значение PromotionID

Теперь результаты показывают только 20 строк, что означает: лишь в 20 из 50 строк таблицы orderheader указано значение для PromotionID. В остальных 30 строках значение PromotionID отсутствует (равно NULL).

У функции COUNT есть еще одно полезное применение, к которому нередко прибегают на практике: с ее помощью можно получить общее количество строк в таблице, не указывая конкретный столбец. Если вам неизвестны имена столбцов в таблице orderheader, можно просто воспользоваться звездочкой (*), как вы уже делали при выборе всех столбцов в главе 3:

```
SELECT COUNT(*) AS TotalOrders
FROM orderheader;
```

Результат запроса будет таким же, как на рис. 12.2, что примечательно: даже если в каких-либо столбцах присутствуют значения NULL, выражение `COUNT(*)` всегда возвращает общее количество строк в таблице.

> **ВНИМАНИЕ** `SELECT COUNT(*)` — отличный инструмент для того, чтобы быстро определить число строк в большинстве таблиц, однако не стоит бездумно применять его к таблицам с миллионами или миллиардами строк (а то и больше). Подобный запрос может потребовать чрезмерных вычислительных ресурсов и вызвать задержки при выполнении других запросов.

12.1.3. Функция MIN

Функция `MIN` возвращает минимальное, то есть наименьшее значение в столбце (игнорируя NULL). Выбрать среди всех заказов самую недорогую книгу (как показано на рис. 12.4) можно при помощи следующего запроса:

```
SELECT MIN(ItemPrice) AS MinimumItemPrice
FROM orderitem;
```

MinimumItemPrice
4.95

Рис. 12.4. Наименьшая цена книги в таблице orderitem составляет 4 доллара 95 центов

12.1.4. Функция MAX

У функции `MIN` есть не менее популярная функция-антоним — `MAX`. Если `MIN` возвращает строку с наименьшим значением, то `MAX` возвращает строку, содержащую максимальное, то есть наибольшее значение. Заменим функцию в предыдущем запросе, чтобы найти самую дорогую книгу в таблице, как продемонстрировано на рис. 12.5:

```
SELECT MAX(ItemPrice) AS MaximumItemPrice
FROM orderitem;
```

MaximumItemPrice
12.95

Рис. 12.5. Наибольшая цена книги в таблице orderitem составляет 12 долларов 95 центов

Если функция `SUM` применима исключительно к числам, то функции `MIN` и `MAX` в MySQL и многих других СУБД умеют обрабатывать и нечисловые данные (строки и даты). При работе со строковыми данными функция `MIN` возвращает наименьшее значение в алфавитном порядке (например, Apple), `MAX` — наибольшее (например, Zebra). Однако для строковых значений важно помнить о настройках параметров сортировки, рассмотренных в предыдущих главах:

строчные и прописные буквы, а также неалфавитные символы могут ранжироваться по-разному в зависимости от выбранной схемы. С датами MIN и MAX работают естественным образом, возвращая самую раннюю и самую позднюю дату соответственно.

К практике!

Напишите короткий запрос для выбора минимального (MIN) значения столбца FirstName в таблице author.

12.1.5. Функция AVG

Последней агрегатной функцией, которую мы рассмотрим в этой главе, является AVG. Она вычисляет среднее (average) значение всех непустых значений в столбце. Так, чтобы установить среднюю цену книг в таблице title, приведенную на рис. 12.6, можно воспользоваться функцией AVG:

```
SELECT AVG(Price) AS AveragePrice
FROM title;
```

AveragePrice
9.700000

Рис. 12.6. Средняя цена всех книг в таблице title

Судя по результату, средняя цена изданий в базе данных составляет приблизительно 9,70 доллара. Вы, разумеется, заметили множество лишних нулей в результате; возникают они потому, что функция AVG пытается вычислить среднее значение с более высокой точностью. Впрочем, это не влияет на корректность полученного значения; в главе 14 мы узнаем, как при необходимости изменить точность результата.

ВНИМАНИЕ Функция AVG, как и функция SUM, применима лишь к числовым значениям.

12.1.6. Сочетание фильтрации и агрегации

Агрегатные функции отлично работают и с фильтрами. Допустим, требуется узнать среднюю цену книг, вышедших в свет после 1 января 2019 года. Сформулируем простыми словами: “I would like the average price of all titles with a publication date greater than January 1, 2019” («Мне нужна средняя цена всех книг, вышедших в свет после 1 января 2019 года»). Эту фразу легко преобразовать в следующий SQL-запрос:

```
SELECT AVG(Price) AS AveragePrice
FROM title
WHERE PublicationDate > '2019-01-01';
```

Допустимо даже объединять функции в одном запросе. Так, если необходимо установить даты первого и последнего заказов в таблице `orderheader`, можно прибегнуть к функциям `MIN` и `MAX`, поскольку они работают со значениями дат:

```
SELECT
    MIN(OrderDate) AS FirstOrder,
    MAX(OrderDate) AS LastOrder
FROM orderheader;
```

Кроме того, в агрегатных запросах имеется возможность выполнять математические вычисления прямо внутри скобок функций. Это особенно полезно, когда нужно объединить данные из нескольких столбцов, — к примеру, чтобы узнать общую стоимость всех проданных товаров. При условии, что общая выручка определяется умножением количества (`Quantity`) на цену каждой позиции (`ItemPrice`), можно задействовать этот расчет с функцией `SUM` для определения общего объема продаж с учетом всех строк в таблице `orderitem`, как показано на рис. 12.7:

```
SELECT SUM(Quantity * ItemPrice) AS TotalOrderValue
FROM orderitem;
```

TotalOrderValue
573.50

Рис. 12.7. Совокупная выручка по всем заказам в таблице orderitem составляет 573 доллара 50 центов

Как видите, вычислить общую сумму продаж довольно легко. До сих пор мы ограничивались расчетами на уровне всей таблицы или анализом данных, отфильтрованных по определенным условиям. А как быть в том случае, когда нужно узнать сумму по каждому отдельному заказу, количество проданных товаров для каждого промокода (`PromotionCode`) или число проданных книг по каждому автору?

12.2. Агрегирование данных посредством GROUP BY

Для проведения анализа данных на более глубоком уровне нам понадобится новый синтаксический элемент: `GROUP BY`. Предложение `GROUP BY` — это не просто пара ключевых слов, а полноценное предложение SQL, позволяющее разбить набор данных на группы, по каждой из которых можно выполнить агрегатные вычисления. Звучит слишком абстрактно? Тогда начнем следующий раздел с конкретного примера.

12.2.1. Сочетание фильтрации и агрегации

В предыдущем запросе мы извлекали общую стоимость всех заказов. Если же нам нужно определить общую стоимость по каждому отдельному заказу, следует разбить данные на группы — по одному заказу в каждой — и выполнить тот же расчет, что и ранее. В этих целях мы сгруппируем значения по `OrderID`. С опорой на формулировку задачи, составленную выше, новый запрос мог бы прозвучать так: "I would like the sum of the quantity multiplied by the item price of all the ordered items, and I want to group the sum by order ID" («Мне нужна сумма произведений количества товара на его цену для всех позиций заказа, и я хочу сгруппировать сумму по идентификатору заказа»).

Вот как это выражение выглядит в форме SQL-запроса. Мы добавим `ORDER BY` для сортировки данных по `OrderID`, чтобы повысить удобочитаемость (результат приведен на рис. 12.8):

```
SELECT OrderID, SUM(Quantity * ItemPrice) AS OrderTotal
FROM orderitem
GROUP BY OrderID
ORDER BY OrderID;
```

OrderID	OrderTotal
1001	9.95
1002	9.95
1003	9.95
1004	9.95
1005	9.95
1006	7.95
1007	15.90
1008	7.95

Рис. 12.8. Восемь из 50 строк выдачи с вычисленным значением OrderTotal для всех заказов

Последний запрос похож на предыдущий, за исключением того, что теперь мы подразделяем данные на логические наборы значений по каждому OrderID и рассчитываем `SUM(Quantity * ItemPrice)` для каждой такой группы. Реализуется это добавлением в запрос предложения `GROUP BY OrderID`, а затем самого `OrderID` в предложение `SELECT`.

> **НА ЗАМЕТКУ** При использовании предложения `GROUP BY` все столбцы, указанные в `SELECT`, должны входить либо в предложение `GROUP BY`, либо в расчетную формулу агрегатной функции. Если какой-либо столбец в `SELECT` не удовлетворяет ни одному из этих условий, при выполнении запроса возникнет синтаксическая ошибка.

Должно быть, вы обратили внимание на тот факт, что предложение `GROUP BY` следует за предложением `FROM` и перед предложением `ORDER BY`. Кроме того,

если в запросе присутствует предложение WHERE, GROUP BY должно располагаться и после него. С учетом вышесказанного, если требуется ограничить выборку заказами, сделанными после 1 января 2019 года, то необходимо присоединить таблицу orderheader для добавления этого условия фильтрации, а также желательно задействовать псевдонимы для имен столбцов:

```
SELECT
    oi.OrderID,
    SUM(oi.Quantity * oi.ItemPrice) AS OrderTotal
FROM orderitem oi
INNER JOIN orderheader oh
    ON oi.OrderID = oh.OrderID
WHERE oh.OrderDate > '2019-01-01'
GROUP BY oi.OrderID
ORDER BY oi.OrderID;
```

Добавление условия фильтрации по OrderDate сокращает результирующий набор с 50 до 21 строки, однако следует помнить, что мы по-прежнему выполняем расчет для каждого набора данных на основе OrderID для 21 строки.

12.2.2. Предложение GROUP BY и NULL-значения

У предложения GROUP BY есть еще одно преимущество: оно позволяет выполнять агрегирование по столбцам, содержащим пустые значения (NULL). Напомню, что в разделе 12.1.2 при использовании агрегатной функции COUNT для вычисления общего числа активированных промокодов NULL-значения автоматически исключались из расчета. Предложение GROUP BY позволяет учитывать и такие строки.

Допустим, нам необходимо учесть те 30 заказов, что были оформлены без промокода, как показано на рис. 12.9. Отсутствие промокода означает, что в столбце PromotionID таблицы orderheader для таких заказов указано значение NULL. Поскольку эти значения пусты, они были проигнорированы в запросе с COUNT из раздела 12.1.2. Однако при помощи GROUP BY мы можем логически сгруппировать заказы по PromotionID, поскольку GROUP BY включает в группировку и NULL-значения:

```
SELECT PromotionID, COUNT(*) AS RowCount
FROM orderheader
GROUP BY PromotionID
ORDER BY PromotionID;
```

В этом запросе мы не фильтруем результаты. Следует отметить, что применение условий фильтра в контексте GROUP BY требует особого подхода: предложение WHERE не может использоваться для отбора строк на основе результатов

агрегатных функций, поскольку оно обрабатывается до этапа группировки и агрегирования. Для фильтрации результатов агрегатных функций требуется введение дополнительного ключевого слова.

PromotionID	RowCount
NULL	30
1	3
2	4
3	3
4	1
5	2
6	1
7	2
9	2
12	2

Рис. 12.9. Значения PromotionID в таблице orderheader. Выборка содержит пустые значения, так как GROUP BY их не исключает

12.3. Фильтрация при помощи HAVING

Предложение `HAVING` логически дополняет `GROUP BY` и предназначено для фильтрации результатов агрегированных вычислений. Функционально оно схоже с предложением `WHERE`. Основное различие состоит в том, что `WHERE` фильтрует исходные строки, а `HAVING` — группы агрегированных значений. Важно отметить, что в предложении `HAVING` допустимо применение всех ранее изученных способов фильтрации.

Рассмотрим конкретный пример. Предположим, требуется найти промокоды (`PromotionCode`), использованные в заказах, и установить, какие из них применялись как минимум три раза. Мы можем начать с запроса, который находит все задействованные промокоды: для этого соединим таблицу `orderheader` с таблицей `promotion` по `PromotionID`, сгруппируем значения по `PromotionCode` и подсчитаем, сколько раз каждый `PromotionID` встречается в таблице `orderheader`. Для удобства чтения результата зададим псевдонимы таблицам и упорядочим вывод:

```
SELECT
    p.PromotionCode,
    COUNT(oh.PromotionID) AS OrdersWithPromotionCode
FROM orderheader oh
INNER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
GROUP BY p.PromotionCode
ORDER BY p.PromotionCode;
```

НА ЗАМЕТКУ Если вы уже выполнили все запросы из этой главы, вас может заинтересовать, куда исчезли пустые значения в результатах предыдущего запроса. Дело в том, что NULL-значения для PromotionID действительно присутствуют в таблице `orderheader`, но их нет в таблице `promotion`. Более того, даже если бы они там были, внутреннее соединение (`INNER JOIN`) все равно исключило бы такие строки, поэтому в результатах отображаются лишь совпадающие значения из обеих таблиц.

Теперь, когда у нас есть базовый запрос для просмотра того, какие промокоды использовались и как часто они включались в заказы, можно добавить предложение `HAVING` для фильтрации промокодов, задействованных трижды или более, с результатами, отображенными на рис. 12.10:

```
SELECT
    p.PromotionCode,
    COUNT(oh.PromotionID) AS OrdersWithPromotionCode
FROM orderheader oh
INNER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
GROUP BY p.PromotionCode
HAVING COUNT(oh.PromotionID) >= 3
ORDER BY p.PromotionCode;
```

PromotionCode	OrdersWithPromotionCode
2OFF2015	3
2OFF2016	4
2OFF2017	3

Рис. 12.10. Только промокоды, активированные как минимум трижды

После добавления предложения `HAVING` в выборке осталось всего три строки. При необходимости можно задействовать псевдоним нашей агрегатной функции (`OrdersWithPromotionCode`) в предложении `HAVING` следующим образом:

```
SELECT
    p.PromotionCode AS PromoCode,
    COUNT(oh.PromotionID) AS OrdersWithPromotionCode
FROM orderheader oh
INNER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
GROUP BY p.PromotionCode
HAVING OrdersWithPromotionCode >= 3
ORDER BY p.PromotionCode;
```

Запрос должен вернуть результаты, приведенные на рис. 12.10. На первый взгляд тут возникает противоречие, ведь в предыдущих главах я отмечал, что в предложении `WHERE` нельзя использовать псевдоним столбца. Вероятно, сейчас самое подходящее время обсудить один важный аспект SQL и составления запросов — логический порядок, в котором реляционная СУБД считывает и обрабатывает наши запросы.

12.4. Логический порядок обработки SQL-запроса

К этому моменту вам уже известно несколько SQL-предложений и порядок, в котором их необходимо записывать. В упрощенном виде последовательность предложений SQL выглядит следующим образом:

1. `SELECT`
2. `FROM` (включая `JOIN`)
3. `WHERE` (включая `AND` и `OR`)
4. `GROUP BY`
5. `HAVING`
6. `ORDER BY`

Однако это не тот порядок, в котором СУБД MySQL читает ваши запросы. Она обрабатывает их в следующем порядке:

1. `FROM` (включая `JOIN`)
2. `WHERE` (включая `AND` и `OR`)
3. `SELECT`
4. `GROUP BY`
5. `HAVING`
6. `ORDER BY`

Последовательность, в которой СУБД читает ваши запросы, имеет особое название — *логическая обработка запросов* (logical query processing). Именно она управляет порядком выполнения всех операций. Значимость этой модели в SQL трудно переоценить — понимание очередности логической обработки запросов не только поможет вам устранять неполадки в коде, который возвращает неожиданные или некорректные результаты, но и позволит определить, когда можно задействовать псевдонимы таблиц и столбцов.

Порядок логической обработки запросов может показаться несколько необычным, тем не менее он является оптимальным для СУБД:

1. Обработка данных в таблицах, с которыми будет работать запрос, в предложении `FROM`.
2. Фильтрация данных для сокращения результирующего набора в предложении `WHERE`.
3. Сбор столбцов для выдачи в предложении `SELECT`.
4. Группировка этих столбцов в предложении `GROUP BY` для агрегации.

5. Фильтрация агрегаций в предложении HAVING.
6. Сортировка результатов в предложении ORDER BY.

Теперь становится ясно, почему псевдонимы таблиц можно применять по всему запросу: в порядке логической обработки запроса они устанавливаются на самом раннем этапе — в предложении FROM. Вы также видите, отчего псевдонимы столбцов, заданные в предложении SELECT, можно использовать в предложениях HAVING и ORDER BY, но нельзя — в предложении WHERE.

> **ВНИМАНИЕ** Несмотря на то что MySQL выполняет логическую обработку запроса именно в такой последовательности, другие СУБД могут поддерживать иной порядок, логически обрабатывая предложение SELECT после GROUP BY и HAVING. Поэтому не стоит привыкать к псевдонимам столбцов и использовать их в предложении HAVING.

12.5. Ключевое слово DISTINCT

В этой главе нам осталось рассмотреть еще одно ключевое слово: DISTINCT. Это удобный и часто применяемый инструмент, назначение которого не всегда понимают верно.

Ключевое слово DISTINCT можно задействовать в предложении SELECT, чтобы исключить повторяющиеся значения. Скажем, если требуется выбрать список всех наименований, которые когда-либо заказывались (рис. 12.11), можно составить такой запрос с использованием DISTINCT:

```
SELECT DISTINCT t.TitleName
FROM title t
INNER JOIN orderitem oi
    ON t.TitleID = oi.TitleID
ORDER BY t.TitleName;
```

TitleName
Anne of Fact Tables
Catcher in the Try
Pride and Predicates
The Call of the While
The DateTime Machine
The Great GroupBy
The Join Luck Club
The Sum Also Rises

Рис. 12.11. Неповторяющиеся названия книг, представленные в заказах таблицы orderitem

Хотя в таблице содержится 50 заказов, причем некоторые из них включают сразу несколько наименований, запрос возвращает лишь по одной строке для каждого наименования. Ключевое слово DISTINCT может пригодиться для моментального определения диапазона значений в любой таблице, и вы, вне всякого сомнения,

будете неоднократно встречать его в сторонних запросах. Так почему же оно рассматривается в главе, посвященной агрегации? А причина вот в чем — при использовании SELECT DISTINCT ваша СУБД выполняет агрегацию, чтобы вернуть только уникальные значения, а это дополнительная работа. Применяя DISTINCT в приведенном выше SQL-коде, мы, по существу, просим СУБД обработать следующий запрос:

```
SELECT t.TitleName
FROM title t
INNER JOIN orderitem oi
    ON t.TitleID = oi.TitleID
GROUP BY t.TitleName
ORDER BY t.TitleName;
```

К практике!

Выполните два предыдущих запроса — с DISTINCT и с GROUP BY соответственно — и обратите внимание, что оба возвращают результаты, показанные на рис. 12.11.

Учитывая принцип действия оператора DISTINCT, старайтесь ограничивать его применение в ваших SQL-запросах, особенно при работе с большими наборами данных. Агрегация данных не вызывает сложностей при работе с небольшими учебными таблицами из нашей книги, но может оказаться весьма ресурсоемкой при оперировании крупными таблицами в промышленных средах.

СОВЕТ Одним из типичных случаев неоптимального применения ключевого слова DISTINCT является попытка устранить дубликаты в результатах запроса при объединении нескольких наборов данных. Если рука тянется поставить DISTINCT, чтобы «почистить» вывод от повторов, остановитесь и внимательно изучите условия соединений: возможно, вы связываете таблицы не по тем столбцам. Именно неправильные соединения, а не «грязные» данные чаще всего порождают эти пресловутые повторяющиеся строки в выборке.

12.6. Практическое занятие

1. Ранее в главе упоминалась несовместимость некоторых функций с отдельными типами данных. Для демонстрации этого ограничения попробуйте вычислить сумму (SUM) значений OrderDate в таблице orderheader и посмотрите, какой будет результат.
2. Почему этот запрос не сработает?

```
SELECT
    p.PromotionCode AS PromoCode,
    COUNT(oh.PromotionID) AS OrdersWithPromotionCode
FROM orderheader oh
INNER JOIN promotion p
    ON oh.PromotionID = p.PromotionID
```

```
WHERE PromoCode = '2OFF2015'
GROUP BY p.PromotionCode
HAVING OrdersWithPromotionCode >= 3
ORDER BY p.PromotionCode;
```

3. Напишите запрос для подсчета количества строк в таблице `author`.
4. Напишите запрос для выбора минимального и максимального значения дат публикации из таблицы `titles`.
5. В этой главе мы рассчитывали общую стоимость всех заказов, применив выражение `Quantity * ItemPrice` к таблице `orderitem`. А теперь составьте запрос с предложением `GROUP BY` для определения средней общей стоимости каждого отдельного заказа. (Подсказка: без подзапроса, скорее всего, не обойтись.)

12.7. Ответы

1. Если помните, в главе 5 я писал, что значения даты и времени хранятся как числовые значения, которые реляционная СУБД умеет преобразовывать в дату и время. Значение, которое вы здесь видите, — это попытка СУБД выполнить агрегацию `SUM` для значений дат, пускай на практике это и не имеет смысла.
2. Запрос завершается ошибкой, поскольку в предложении `WHERE` задействуется псевдоним столбца, а при логической обработке запроса предложение `WHERE` обрабатывается до предложения `SELECT`. Поэтому СУБД не знает, что такое `PromoCode`, когда обрабатывает `WHERE`: псевдоним определяется в `SELECT`, который выполняется на более позднем этапе обработки запроса.
3. Для подсчета количества строк в таблице можно воспользоваться `COUNT(*)`:

```
SELECT COUNT(*)
FROM author;
```

4. Для отбора минимального и максимального значений дат публикации следует применить агрегатные функции `MIN` и `MAX`:

```
SELECT
    MIN(PublicationDate) AS FirstPublication,
    MAX(PublicationDate) AS LastPublication
FROM title;
```

5. Существует несколько способов решения задачи. Первый — сгруппировать общую стоимость всех заказов, как мы делали в разделе 12.2.1, с последующим вычислением среднего значения полученных сумм:

```
SELECT AVG(OrderTotals.OrderTotal)
FROM (
    SELECT OrderID, SUM(Quantity * ItemPrice) AS OrderTotal
    FROM orderitem
    GROUP BY OrderID
    ) OrderTotals;
```

13 Работа с переменными

Мы уже написали и выполнили множество SQL-запросов, и многие из них включали фильтрацию результатов по конкретным значениям. В ходе рассмотрения различных примеров мы узнали, как фильтровать по определенному идентификатору заказа или наименования товара, имени клиента или диапазону дат, и каждый раз указывали конкретное литеральное значение для фильтрации в SQL. *Литеральное* значение — это конкретная величина, скажем, число `4` или дата `2020-10-06`. Литеральные значения отлично подходят для учебных упражнений, но в реальной жизни вам придется создавать более универсальные запросы.

Если необходимо выяснить общее количество продаж по конкретной позиции за определенный месяц, скажем, за март 2021 года, вы сможете написать соответствующий запрос прямо сейчас. Но что делать, если вам нужны аналогичные сведения за апрель или необходимо подсчитать продажи по другой товарной позиции или за другой промежуток времени? Неужели придется составлять отдельный запрос для каждого наименования и диапазона дат?

Смею вас заверить — не придется. Нужно всего лишь освоить работу с переменными. Переменная — это «контейнер», хранящий значение в памяти, который можно многократно задействовать в пределах одного или нескольких запросов. И, что гораздо важнее, содержимое такого контейнера может меняться в ходе выполнения программного кода, то есть является переменным — отсюда и название.

Благодаря своей гибкости переменные станут неотъемлемой частью ваших SQL-запросов. Итак, начнем!

13.1. Пользовательские переменные

При том что существует несколько типов переменных, в этой главе мы будем работать с так называемыми *пользовательскими переменными* (user-defined variables). Название говорит само за себя: такие переменные *определяет* (define) сам пользователь (вы или я), то есть присваивает им имя и значение. Все эти переменные начинаются со знака `@` («коммерческая at», «собака»), и потому, если вам встретился символ `@` в SQL-запросе, скорее всего, перед вами переменная.

Прежде чем использовать любую переменную, ее необходимо объявить. Начиная с главы 2, мы объявляли наши намерения на простом английском языке, чтобы лучше понять синтаксис запросов, но иногда требуется также объявлять объекты непосредственно в самом SQL-коде. Посмотрим, как это делается в MySQL.

13.1.1. Объявляем нашу первую пользовательскую переменную

Объявление переменной требует, как правило, указания двух обязательных компонентов: имени переменной и ее значения. Допустим, нам нужно написать запрос, фильтрующий строки по названию книги. Мы можем выбрать подходящее имя переменной, скажем, `@TitleName`, и присвоить ему значение `'The Sum Also Rises'`. Сформулируем простыми словами: "I would like to declare a variable named @TitleName, and I would like to assign it the value of The Sum Also Rises" («Хочу объявить переменную с именем @TitleName и присвоить ей значение The Sum Also Rises»). В SQL для такого объявления служит похожая по логике конструкция с новым ключевым словом `SET`:

```
SET @TitleName = 'The Sum Also Rises';
```

Как и во многих конструкциях языка SQL, его синтаксическая структура соотносится с естественным порядком речевого высказывания. Объявление переменной выполняется путем задания ее имени посредством ключевого слова `SET` (установить»), после чего осуществляется присваивание значения при помощи оператора `=` и соответствующего литерального выражения.

> **НА ЗАМЕТКУ** `SET` допускает два варианта оператора присваивания — `=` и `:=`. Выбор за вами: можно применять любой из них. Однако, как мы увидим далее в этой главе, как минимум в одном случае для объявления переменной необходимо задействовать именно `:=`.

Вы наверняка заметили, что мы не указали тип данных. Это связано с тем, что MySQL определяет тип данных на основе присваиваемого значения. В нашем примере в качестве значения переменной выступает строковое значение (`'The Sum Also Rises'`), поэтому наша переменная имеет строковый тип данных.

Другие допустимые типы данных для переменных включают целые числа (integer), десятичные (decimal) и числа с плавающей запятой (float), относящиеся к числовым типам. Если данные для переменной не соответствуют какому-либо из допустимых типов, СУБД преобразует значения в допустимый тип данных. Значения даты и времени, с которыми мы работаем на протяжении всей книги, интерпретируются как строки (string).

> **ВНИМАНИЕ** Подобный метод объявления переменных в MySQL не является универсальным. При работе с другими реляционными СУБД, такими как SQL Server или PostgreSQL, требуется объявлять пользовательские переменные при помощи ключевого слова `DECLARE` с обязательным указанием конкретного типа данных.

При необходимости можно в любой момент проверить значение переменной при помощи простой инструкции `SELECT` (рис. 13.1):

```
SELECT @TitleName;
```

@TitleName
The Sum Also Rises

Рис. 13.1. Результат выполнения SELECT @TitleName, показывающий значение переменной. В заголовке столбца также отображается имя переменной

Хотя проверка значения переменной посредством `SELECT` может показаться тривиальной операцией, вам придется частенько прибегать к ней при работе с переменными. При разработке SQL-запросов такой способ может пригодиться как для периодического мониторинга значений переменных, так и для диагностики сложных SQL-сценариев, возвращающих некорректные результаты.

13.1.2. Правила работы с пользовательскими переменными

Прежде чем продолжить, озвучу несколько ключевых правил работы с переменными. Их совсем немного, они простые, но обязательные к соблюдению.

- *Первый символ имени переменной должен быть* `@`. Применение `@` в имени переменной сообщает СУБД, что вы работаете с переменной.
- *Все последующие символы в имени переменной должны быть буквенными или цифровыми*. Как видно на примере `@`, прочие символы могут иметь специальное значение в SQL. Используйте только буквы и цифры в именах переменных.
- *Имена переменных могут содержать не более 64 символов*. Следует употреблять понятные и информативные имена, чтобы другим разработчикам, читающим ваш SQL-код, было легко понять их назначение. Если имя какой-либо из ваших переменных приближается к пределу в 64 символа, вы, вероятно, перестарались с подробностями.

- *Имена переменных не чувствительны к регистру*. Если вы объявили переменную с именем `@Variable`, то любые обращения вида `@VARIABLE`, `@variable` или `@VaRiAbLe` будут ссылаться на одну и ту же переменную.
- *Пользовательская переменная может хранить только одно значение*. Несколько значений в ней разместить нельзя, хотя при желании значение переменной можно изменять в процессе выполнения SQL-кода.
- *Пользовательская переменная существует лишь в течение сеанса подключения*. Базы данных и таблицы являются *постоянными объектами*, то есть они хранятся (persist) до тех пор, пока их явно не удалят. В отличие от них, переменные не сохраняются: как только вы закроете MySQL Workbench или любое другое приложение, через которое подключались к базе данных, все объявленные вами переменные исчезнут без следа.

13.1.3. Задействуем нашу первую пользовательскую переменную

После ознакомления со всеми правилами применим переменные в деле. Сформулируем SQL-запрос с переменной для выборки по столбцам `TitleID`, `TitleName` и `PublicationDate` из таблицы `title`. Результаты приведены на рис. 13.2:

```
SET @TitleName = 'The Sum Also Rises';
SELECT
    TitleID,
    TitleName,
    PublicationDate
FROM title
WHERE TitleName = @TitleName;
```

TitleID	TitleName	PublicationDate
108	The Sum Also Rises	2021-11-12 00:00:00

Рис. 13.2. Значения TitleID, TitleName и PublicationDate из таблицы title для заглавия 'The Sum Also Rises', отобранные при помощи пользовательской переменной

Пока что у нас получился совсем короткий фрагмент SQL-кода — честно говоря, даже слишком короткий. Однако представьте, что для заданного наименования необходимо выполнить гораздо больше операций: к примеру, определить количество проданных экземпляров или регион проживания покупателей, приобретших эту книгу. Тогда наш код был бы объемней. И тем не менее при использовании переменной для задания и фильтрации по конкретному заглавию для перехода к другому наименованию нам потребовалось бы внести правку лишь в одном месте SQL-кода.

Вот как это выглядит на практике. Присвоим переменной новое значение — `'Pride and Predicates'` — и повторно выполним запрос. Результат его выполнения показан на рис. 13.3:

```
SET @TitleName = 'Pride and Predicates';
SELECT
    TitleID,
    TitleName,
    PublicationDate
FROM title
WHERE TitleName = @TitleName;
```

TitleID	TitleName	PublicationDate
101	Pride and Predicates	2015-04-30 00:00:00

Рис. 13.3. Значения TitleID, TitleName и PublicationDate из таблицы title для заглавия 'Pride and Predicates', отобранные при помощи пользовательской переменной

Пусть этот пример и прост, зато он наглядно демонстрирует главное преимущество переменных: они делают код универсальным и пригодным для многократного использования. Переменные — основа любого языка программирования, и многочисленные примеры в этой и последующих главах позволят нам раскрыть эффективные способы их применения.

К практике!

Объявите переменную с произвольным именем и задействуйте ее для выбора значений TitleID, TitleName и PublicationDate из таблицы title для любой конкретной книги.

13.2. Фильтрация с помощью переменных в предложениях FROM и HAVING

Рассмотрим практические способы применения переменных в SQL. Допустим, нам нужно получить даты всех заказов для любой конкретной книги. Для решения этой задачи можно воспользоваться переменной. Давайте начнем с произведения `'The Sum Also Rises'`. Задействовав известную нам логику соединения таблиц, составим следующий запрос (результаты приведены на рис. 13.4):

```
SET @TitleName = 'The Sum Also Rises';

SELECT
    oh.OrderDate
FROM orderheader oh
INNER JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
```

```
INNER JOIN title t
    ON oi.TitleID = t.TitleID
WHERE t.TitleName = @TitleName;
```

OrderDate
2021-11-13 00:00:00
2021-11-28 00:00:00
2022-03-10 00:00:00

Рис. 13.4. Значения OrderDate для любого заказа, включающего в себя название 'The Sum Also Rises'

Замените это на любое другое применимое название из таблицы — и вы получите соответствующий набор дат заказов. Как вы, возможно, уже догадались, если присвоить переменной значение, которого нет в таблице `title`, результат окажется пустым (содержащим ноль строк).

Кроме того, переменными часто пользуются для получения сведений о заказах за конкретный период — день, неделю, месяц или год. В качестве примера установим имена всех покупателей, разместивших заказы на какие-либо книги в ноябре 2021 года (рис. 13.5). В этом случае мы задействуем две переменные для обозначения начальной и конечной даты интересующего нас диапазона:

```
SET @DateStart = '2021-11-01',
     @DateEnd = '2021-11.30';

SELECT
    c.FirstName,
    c.LastName,
    oh.OrderDate
FROM customer c
INNER JOIN orderheader oh
    ON c.CustomerID = oh.CustomerID
WHERE oh.OrderDate BETWEEN @DateStart and @DateEnd;
```

FirstName	LastName	OrderDate
Keanu	O'Ward	2021-11-13 00:00:00
Monica	Newgarden	2021-11-28 00:00:00

Рис. 13.5. Имена и фамилии всех покупателей, сделавших заказ в ноябре 2021 года, а также дата оформления заказа

Примечательно, что данный фильтр можно разместить в другой части логического предиката. Вместо того чтобы применять его в предложении `WHERE`, допустимо сделать его частью условия соединения `JOIN`. (Хотя такой способ фильтрации в SQL используется нечасто, он встречается в сторонних реализациях.) Вот как выглядел бы приведенный выше запрос, перенеси мы фильтрацию по переменным в предложение `FROM` — прямо в условие соединения `JOIN`:

```
SET
    @DateStart = '2021-11-01',
    @DateEnd = '2021-11-30';
```

```
SELECT
    c.FirstName,
    c.LastName,
    oh.OrderDate
FROM customer c
INNER JOIN orderheader oh
    ON c.CustomerID = oh.CustomerID
    AND oh.OrderDate BETWEEN @DateStart and @DateEnd;
```

Также можно задействовать переменную, чтобы увидеть, какие книги были проданы в количестве выше установленного. Здесь возможно применение методов агрегации, изученных нами в главе 12, с предложением `HAVING` в качестве фильтра, использующего переменную. Сформируем перечень всех названий книг, которые были проданы в количестве 10 или более экземпляров, как представлено на рис. 13.6:

```
SET @MinimumQuantitySold = 10;

SELECT
    t.TitleName,
    SUM(oi.Quantity) AS TotalQuantitySold
FROM orderitem oi
INNER JOIN title t
    ON oi.TitleID = t.TitleID
GROUP BY t.TitleName
HAVING SUM(oi.Quantity) >= @MinimumQuantitySold;
```

TitleName	TotalQuantitySold
Pride and Predicates	25
The Join Luck Club	13
Catcher in the Try	11
The DateTime Machine	13

Рис. 13.6. Выборка по столбцам TitleName и TotalQuantitySold всех книг, проданных в количестве не менее 10 экземпляров

Результирующая выборка содержит четыре книги, удовлетворяющие установленному порогу в 10 проданных экземпляров (`Quantity`). Если мы хотим изменить порог фильтра на другое значение, достаточно изменить значение переменной `@MinimumQuantitySold`.

13.3. Присвоение переменной неизвестного значения

Одно из полезных свойств переменных — возможность создавать и использовать их даже тогда, когда мы не знаем точного значения для фильтрации. Скажите честно: спроси я вас, какой `TitleID` у книги «The Sum Also Rises», вы вспомните? Даже я, автор всей этой базы, не удержу в голове такие детали.

Однако, несмотря на то что мы не помним `TitleID` наизусть, мы постоянно задействуем их в запросах — ведь именно через эти идентификаторы таблицы `orderheader` и `title` связаны между собой.

13.3.1. Разберем, как работает запрос

Уделим немного времени рассмотрению того, как в первом запросе из раздела 13.2 применяется TitleID и как можно задействовать одну переменную для получения значения другой переменной. Вот этот самый запрос, который находит даты заказов книги «The Sum Also Rises»:

```
SET @TitleName = 'The Sum Also Rises';

SELECT
    oh.OrderDate
FROM orderheader oh
INNER JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
INNER JOIN title t
    ON oi.TitleID = t.TitleID
WHERE t.TitleName = @TitleName;
```

Для начала проанализируем последовательность соединений в запросе, двигаясь снизу вверх. Почему именно так? Скорее всего, СУБД начнет с того, что отфильтрует строки в таблице `title`. Применение фильтрации на раннем этапе позволяет сократить объем обрабатываемых данных, поскольку уменьшается количество строк, подлежащих последующему соединению с другими таблицами. Такой подход эффективнее, чем сначала читать все строки во всех таблицах, соединять их, а уже потом применять фильтрацию.

Здесь мы прибегли к переменной `@TitleName`, чтобы найти в таблице `title` все строки, соответствующие нашему условию (здесь это одна строка). Затем мы соединили эту строку со связанными строками в таблице `orderitem` по совпадающим значениям `TitleID`, а далее — со связанными строками в таблице `orderheader` по совпадающим значениям `OrderID`. Получив связанные данные из всех таблиц, можно выбрать значения `OrderDate`, чтобы определить, когда именно были оформлены заказы на это издание.

13.3.2. Присвоение неизвестного значения переменной посредством SELECT

Запрос этот является относительно простым и содержит лишь несколько соединений. Однако важно помнить, что каждое соединение — это дополнительная нагрузка на СУБД, ведь ей приходится читать и связывать данные из разных таблиц. К счастью, в запросе из раздела 13.3.1 мы можем уменьшить количество

соединений, создав переменную для значения TitleID, поскольку нам доподлинно известно, что в таблице title только одно значение соответствует наименованию «The Sum Also Rises».

Для получения неизвестного значения TitleID переменную можно инициализировать немного иначе: вместо ключевого слова SET задействовать SELECT, так как требуемое значение переменной будет извлекаться непосредственно из существующей таблицы.

Но сначала сделаем шаг назад. Допустим, мы хотим просто вернуть значение TitleID, а не присваивать его переменной @TitleID. Тогда наш SQL-запрос с использованием переменной @TitleName для названия книги мог бы выглядеть так:

```
SET @TitleName = 'The Sum Also Rises';

SELECT TitleID
FROM title
WHERE TitleName = @TitleName;
```

К практике!

Раз уж я продолжаю говорить об этом значении, выполните этот запрос, чтобы выбрать значение TitleID для произведения «The Sum Also Rises».

Можно применить логику такого запроса, чтобы присвоить значение этого TitleID новой переменной @TitleID. На рис. 13.7 показан вывод присвоенного значения:

```
SET @TitleName = 'The Sum Also Rises';

SELECT @TitleID := TitleID
FROM title
WHERE TitleName = @TitleName;
```

@TitleID := TitleID
108

Рис. 13.7. Выбор неизвестного значения для TitleID в переменной @TitleID приведет к отображению этого значения в выводе — в данном случае 108

Предложения FROM и WHERE здесь полностью совпадают с предыдущим запросом, но предложение SELECT выглядит иначе — такого мы еще не делали. С помощью конструкции SELECT @TitleID := TitleID мы одновременно выполняем две операции: выбираем значение TitleID и присваиваем его переменной @TitleID.

И еще один важный нюанс: в этом SQL-запросе мы воспользовались оператором присваивания := вместо привычного =. Выше мы уже узнали, что с ключевым

словом SET работают оба варианта. Однако при присвоении значения переменной через SELECT в MySQL допустим только оператор :=.

> **ВНИМАНИЕ** Как я отмечал ранее при рассмотрении команды SET, этот метод присвоения переменной неизвестного значения практически в каждой СУБД реализован по-разному. Общий принцип единый, однако синтаксис следует уточнять для каждой системы.

Еще один любопытный побочный эффект присвоения значения переменной с помощью SELECT заключается в том, что результаты выводятся на панель результатов. Команды SELECT генерируют вывод в MySQL, при этом отображается присвоенное переменной значение, а заголовком столбца служит SQL-код из SELECT.

13.3.3. Производительность при работе с переменными

Теперь, когда мы научились присваивать значение переменной с помощью SELECT, рассмотрим полную версию запроса для определения дат заказов книги «The Sum Also Rises» (результаты представлены на рис. 13.8):

```
SET @TitleName = 'The Sum Also Rises';

SELECT @TitleID := TitleID
FROM title
WHERE TitleName = @TitleName;

SELECT
    oh.OrderDate
FROM orderheader oh
INNER JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
WHERE oi.TitleID = @TitleID;
```

OrderDate
2021-11-13 00:00:00
2021-11-28 00:00:00
2022-03-10 00:00:00

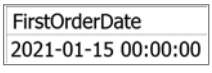
Рис. 13.8. Значения OrderDate для всех заказов, содержащих название «The Sum Also Rises». На этот раз использовалось ключевое слово SELECT вместо SET для получения результатов, приведенных на рис. 13.4

> **НА ЗАМЕТКУ** Если приглядеться к панели результатов, вы обнаружите внизу панели две отдельные вкладки с двумя различными наборами данных. Результаты (вкладка Result 2), показанные на рис. 13.8, соответствуют целевому выводу, запрошенному в нашем SQL-сценарии. А на «скрытой» вкладке (Result 1), как вы, наверное, уже догадались, отображается вывод первой команды SELECT, в которой присваивалось значение переменной.

Запрос задействует две переменные, но в последнем операторе обходится одним соединением вместо двух. Пусть сейчас это и выглядит несущественно — вычислительная нагрузка на СУБД остается невысокой, — при обработке значительных объемов данных подобная оптимизация (сокращение количества операций соединения) способна обеспечить ощутимый прирост производительности.

13.3.4. Диагностика и отладка при работе с переменными

Рассмотрим такую задачу: требуется найти и вывести название, количество и цену позиции из первого заказа в определенном (скажем, 2021) году. Для этого сначала необходимо установить дату первого заказа в 2021 году, а затем, оперируя этим значением, извлечь сведения о заказе, сделанном в эту дату. В нашей базе данных так мало заказов, что больше одного заказа в день не бывает, и это упрощает задачу. Начнем с определения даты первого заказа в 2021 году, как показано на рис. 13.9.



FirstOrderDate
2021-01-15 00:00:00

Рис. 13.9. Значение OrderDate для первого заказа, сделанного в 2021 году, которое выводится исключительно при помощи инструкции SELECT. Мы задействуем его позже для получения дополнительной информации о заказе, сделанном в этот день

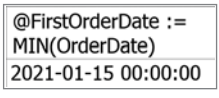
Я еще не затрагивал этот аспект, но вместо SELECT для присвоения неизвестного значения переменной можно применить метод SET. Впрочем, для этого потребуется применить подзапрос, к примеру, следующего вида:

```
SET @FirstOrderDate = (
    SELECT MIN(OrderDate)
    FROM orderheader
    WHERE OrderDate BETWEEN '2021-01-01' AND '2021-12-31');

SELECT @FirstOrderDate AS FirstOrderDate;
```

Запрос формирует корректный результат, однако значение, присвоенное переменной, можно визуализировать лишь при явном использовании отдельной инструкции SELECT. Однако если изначально задействовать SELECT вместо SET, значение переменной сразу будет отображено в результатах (рис. 13.10):

```
SELECT @FirstOrderDate := MIN(OrderDate)
FROM orderheader
WHERE OrderDate BETWEEN '2021-01-01' AND '2021-12-31';
```



@FirstOrderDate := MIN(OrderDate)
2021-01-15 00:00:00

Рис. 13.10. Значение OrderDate для первого заказа, сделанного в 2021 году, отображается без применения второй инструкции SELECT. Заголовком столбца выступает SQL-код из предложения SELECT

Какой метод выбрать — решать вам. Все зависит от того, хотите ли вы видеть присвоенное значение переменной прямо в результатах: в отличие от SELECT, команда SET по умолчанию не отображает значение переменной. Пользоваться SELECT-методом удобно на этапе отладки, поскольку он позволяет проследить, правильно ли заполнились переменные, и впоследствии избежать ошибочных результатов. Визуальное подтверждение значений в панели результатов повышает достоверность и надежность SQL-кода.

Пока остановимся на SELECT: с его помощью найдем первый заказ 2021 года и выберем из него название, количество и цену. Следующий запрос возвращает результаты, приведенные на рис. 13.11:

```
SELECT @FirstOrderDate := MIN(OrderDate)
FROM orderheader
WHERE OrderDate BETWEEN '2021-01-01' AND '2021-12-31';

SELECT
    t.TitleName,
    oi.Quantity,
    oi.ItemPrice
FROM orderheader oh
INNER JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
INNER JOIN title t
    ON oi.TitleID = t.TitleID
WHERE oh.OrderDate = @FirstOrderDate;
```

TitleName	Quantity	ItemPrice
The DateTime Machine	1	7.95

Рис. 13.11. Значения заглавия (TitleName), количества (Quantity) и стоимости (ItemPrice) для позиций в первом заказе, оформленном в 2021 году

Возможность выбора между SET и SELECT при работе с переменными в SQL — это несомненное преимущество. Теперь вам известны плюсы и минусы каждого подхода к присвоению значений пользовательским переменным.

13.4. Еще несколько слов о переменных

Прежде чем мы перейдем к упражнениям, стоит упомянуть еще ряд важных моментов, связанных с переменными.

13.4.1. Присвоение литерального значения с помощью SELECT

Хотя мы не обсуждали этот аспект ранее, оператор SELECT допустимо использовать для присвоения литерального (константного) значения переменной вместо SET. Выбор между ними — дело вкуса, но на протяжении всей главы мы

видели, что SELECT предоставляет больше возможностей за счет совместимости с предложениями FROM, WHERE и др. Для присвоения значения даты можно задействовать такой синтаксис:

```
SELECT @SomeDate := '2021-11-30';
```

13.4.2. Присвоение переменной значения NULL

Иногда бывает полезно с самого начала инициализировать переменную значением NULL, чтобы потом проверить, изменилось ли оно в ходе выполнения сценария. Тип переменной не определен до тех пор, пока ей не присвоено конкретное значение; в этот момент тип переменной автоматически становится таким же, как у присваиваемого значения. Значение NULL может быть присвоено переменной посредством как SET, так и SELECT, при помощи любой из следующих строчек SQL-кода:

```
SET @NullVariableWithSET = NULL;

SELECT @NullVariableWithSELECT;
```

13.4.3. Изменение типа данных переменной

В MySQL одной и той же переменной в ходе выполнения SQL-сценария можно последовательно присваивать разные значения. Тип данных переменной может меняться: если сначала ей присвоено значение NULL, а позже — строка, целое число или значение другого типа, переменная примет тип последнего присвоенного значения. Случаев, когда действительно нужно менять тип переменной, немного, но при желании вы можете присваивать ей значения разных типов в рамках одного сценария. В следующем примере сначала переменной задается числовое значение, а затем — строковое:

```
SET @SomeVariable = 1;

SELECT @SomeVariable AS FirstValue;

SET @SomeVariable = 'The Sum Also Rises';

SELECT @SomeVariable AS SecondValue;
```

Хотя технически такая смена типа переменной возможна, она относится к разряду «можно, но не нужно». Упоминаю об этом только потому, что, если вы случайно задействуйте одну переменную дважды с разными типами данных, система не выдаст никакого сообщения об ошибке или предупреждения о подобном действии.

13.5. Практическое занятие

1. В этой главе отмечалось, что в MySQL при присвоении значения посредством `SELECT` обязательно следует использовать оператор `:=`. Что произойдет, если вместо него задействовать `=`?
2. Я также упоминал, что переменной можно присвоить лишь одно значение. Что произойдет, если выполнить следующий запрос? Будет ли переменной присвоено значение, и если да, то какое?

```
SELECT @TitleID := TitleID
FROM title;
```

3. Просмотрите итоговый запрос из раздела 13.3.4 и измените его, используя переменные для начальной и конечной даты.
4. Составьте запрос для расчета общей суммы продаж в виде произведения количества (`Quantity`) на цену (`Price`) для любого покупателя, задав имя (`FirstName`) и фамилию (`LastName`) покупателя через переменные.

13.6. Ответы

1. Если в команде `SELECT` применять `=` вместо `:=` для присвоения значения переменной, то она получит значение NULL. Объясняется это тем, что в СУБД MySQL оператор `=` служит исключительно для проверки равенства, что неоднократно демонстрировалось при формулировке условий фильтрации и соединений. В контексте `SELECT` система сравнивает два значения и находит, что они не равны, поскольку переменная изначально не содержит никакого значения. Помните: NULL означает отсутствие данных.
2. Значение переменной будет равно 108. Несмотря на то что в результатах вы увидите весь список TitleID, переменная запомнит только последнее из них, так как она может хранить лишь одно значение. Прибегая к такому способу присвоения значений переменным, будьте внимательны и удостоверьтесь в том, что выбираете ровно одно значение.
3. Подобная методика применения SQL обеспечивает повышенную гибкость запроса, позволяя без особого труда вносить изменения в начале кода для анализа различных диапазонов данных:

```
SET
    @DateStart = '2021-01-01',
    @DateEnd = '2021-12-31';

SELECT @FirstOrderDate := MIN(OrderDate)
FROM orderheader
```

```
WHERE OrderDate BETWEEN @DateStart and @DateEnd;

SELECT
    t.TitleName,
    oi.Quantity,
    oi.ItemPrice
FROM orderheader oh
INNER JOIN orderitem oi
    ON oh.OrderID = oi.OrderID
INNER JOIN title t
    ON oi.TitleID = t.TitleID
WHERE oh.OrderDate = @FirstOrderDate;
```

4. Существует ряд способов сделать это. Вот один из них:

```
SET @FirstName = 'Chris';
SET @LastName = 'Dixon';

SELECT @CustomerID := CustomerID
FROM Customer
WHERE FirstName = @FirstName
    AND LastName = @LastName;

SELECT
    @FirstName AS FirstName,
    @LastName AS LastName,
    SUM(oi.Quantity * oi.ItemPrice) AS TotalSalesDollars
FROM orderheader oh
INNER JOIN orderitem oi
     ON oh.OrderID = oi.OrderID
WHERE oh.CustomerID = @CustomerID;
```

14 Работа с функциями в запросах

В главе 12 мы рассмотрели отдельные встроенные функции — программные конструкции, реализующие заранее определенные вычислительные операции. В частности, речь шла об основных агрегатных функциях, позволяющих без особого труда вычислять сумму диапазона значений, а также минимальное, максимальное и среднее значения для заданного набора данных.

В этой главе мы изучим новый набор функций, расширяющих возможности SQL, включая те, что позволяют выбирать и фильтровать конкретные строковые значения, даты и время, а также другие виды информации. Но начнем мы с более общего вопроса: когда следует применять функции, а когда — избегать их.

14.1. Минусы применения функций

Функции — мощнейший инструмент для выборки по отдельным фрагментам значений, вычисления новых значений и преобразования данных в SQL. Они подобны магическим заклинаниям, которые можно применить, просто добавив одно волшебное слово в ваш SQL-запрос. В то же время использование функций сопряжено с двумя существенными проблемами, которые необходимо обозначить, прежде чем вы начнете активно внедрять их в ваши запросы.

14.1.1. Набор команд функций варьируется в каждой СУБД

Основные ключевые слова и предложения, которыми мы пользовались до сих пор (такие, как `SELECT`, `FROM`, `WHERE` и `GROUP BY`), в целом являются универсальными. Запрос, написанный с их помощью, будет работать не только в MySQL, но и в любой другой системе управления реляционными базами данных. А вот

функции универсальными не являются: многие из функций, рассматриваемых в этой главе, могут называться по-другому или работать иначе в других СУБД.

Это не означает, что изучаемые вами функции применимы исключительно в MySQL — многие из них поддерживаются и другими системами. Но если при переносе кода в другую СУБД вы столкнетесь с синтаксической ошибкой, скорее всего, вам придется покопаться в документации, чтобы выяснить правильное название функции для этой конкретной системы. Я постараюсь отмечать такие различия по ходу главы, поскольку они могут стать серьезной преградой при адаптации ваших навыков работы с SQL к другой платформе.

14.1.2. Применение функций может быть неэффективным

Как мы отметили в конце главы 12, употребление ключевого слова `DISTINCT` сопряжено с дополнительной вычислительной нагрузкой: для исключения дубликатов СУБД должна считать все значения в указанном диапазоне. Я также подчеркнул, что словом `DISTINCT` следует пользоваться с особой осторожностью при работе с большими объемами данных во избежание неоправданных расходов серверных ресурсов.

То же самое касается почти всех функций, рассматриваемых в этой главе: в большинстве случаев им тоже приходится сканировать весь диапазон данных. Несмотря на высокую эффективность и кажущуюся простоту применения, такие функции могут оказывать ощутимое влияние на быстродействие при работе с массивными наборами данных. И тогда «простота» рискует обернуться долгим ожиданием и повышенной нагрузкой на сервер. Безусловно, функции в SQL всегда дадут верный ответ, однако не всегда самым экономичным способом.

Но не пугайтесь! Я вовсе не призываю вас отказываться от функций. Просто важно изначально понимать не только их плюсы, но и минусы.

14.2. Строковые функции

Строковые функции (string functions) позволяют извлекать или преобразовывать фрагменты строковых данных, что особенно актуально, когда требуется представить информацию иначе, чем она хранится в базе данных. Примеры таких задач: вывод имен клиентов заглавными буквами для почтовой рассылки или удаление лишних пробелов в начале и конце строкового значения.

14.2.1. Функции для работы с регистром

Первой строковой функцией, с которой мы познакомимся, будет `UPPER`. Она преобразует все символы строки в верхний регистр. Применим ее для выборки

покупателей, проживающих в Калифорнии (они обозначены как «CA» в столбце State, рис. 14.1). Для сравнения включим в запрос исходные значения столбцов FirstName и LastName:

```
SELECT
    FirstName,
    LastName,
    UPPER(FirstName),
    UPPER(LastName)
FROM customer
WHERE State = 'CA';
```

Синтаксис функции UPPER, как и большинства других функций, сводится к указанию имени функции с последующим размещением в круглых скобках ее аргумента — имени столбца, переменной или другого значения, к которому применяется эта функция. Поэтому в документации имя функции часто приводят вместе со скобками — к примеру, UPPER().

FirstName	LastName	UPPER(FirstName)	UPPER(LastName)
Cora	Daly	CORA	DALY
Tara	Di Silvestro	TARA	DI SILVESTRO
Margaret	Montoya	MARGARET	MONTOYA

Рис. 14.1. Имена и фамилии всех покупателей из Калифорнии: сначала в исходном формате из таблицы customer, а затем в верхнем регистре с применением функции UPPER

Как видите, третий и четвертый столбцы на рис. 14.1, как и предполагалось, содержат текст в верхнем регистре. Также обратите внимание на заголовки возвращаемых столбцов. В них просто повторяется содержимое предложения SELECT — так СУБД называет столбцы по умолчанию, если вы не задали им псевдонимы. Запустим тот же запрос снова, но теперь присвоим столбцам понятные имена с префиксом Upper (результат отображен на рис. 14.2):

```
SELECT
    FirstName,
    LastName,
    UPPER(FirstName) AS UpperFirstName,
    UPPER(LastName) AS UpperLastName
FROM customer
WHERE State = 'CA';
```

Результат стал удобнее для восприятия. По аналогии можно преобразовать текст в строчные буквы. Хотя потребность в приведении к нижнему регистру возникает куда реже, соответствующая функция тоже всегда под рукой. Для этой задачи вместо UPPER мы задействуем функцию-антоним LOWER (результаты показаны на рис. 14.3), указав подходящие псевдонимы столбцов:

```
SELECT
    FirstName,
    LastName,
    LOWER(FirstName) AS LowerFirstName,
    LOWER(LastName) AS LowerLastName
FROM customer
WHERE State = 'CA';
```

FirstName	LastName	UpperFirstName	UpperLastName
Cora	Daly	CORA	DALY
Tara	Di Silvestro	TARA	DI SILVESTRO
Margaret	Montoya	MARGARET	MONTOYA

Рис. 14.2. Имена и фамилии всех покупателей из Калифорнии: сначала в исходном формате из таблицы customer, а затем в верхнем регистре с применением функции UPPER с соответствующими именами столбцов

FirstName	LastName	LowerFirstName	LowerLastName
Cora	Daly	cora	daly
Tara	Di Silvestro	tara	di silvestro
Margaret	Montoya	margaret	montoya

Рис. 14.3. Имена и фамилии всех покупателей из Калифорнии: сначала в исходном формате из таблицы customer, а затем в нижнем регистре с применением функции LOWER с соответствующими именами столбцов

14.2.2. Функции удаления пробельных символов (TRIM-функции)

Еще одна полезная задача при обработке строковых данных — это удаление начальных или конечных пробелов. Для ее решения MySQL предоставляет три отдельные функции: `RTRIM`, `LTRIM` и `TRIM`. Функция `RTRIM` удаляет все *конечные пробелы* (trailing spaces) — то есть все пробелы справа от последнего непробельного символа в строке. Функция `LTRIM` удаляет все *начальные пробелы* (leading spaces) — все пробелы слева от первого непробельного символа строки. Функция `TRIM` эквивалентна одновременному применению `LTRIM` и `RTRIM`: она удаляет и начальные, и конечные пробелы.

> **НА ЗАМЕТКУ** Допускается применение двух функций к одному и тому же значению, к примеру: `SELECT RTRIM(LTRIM(SomeValue))`. При этом не забывайте проследить за соблюдением правильного логического порядка: первой выполняется та функция, что внутри. Такой код иногда встречается у тех разработчиков, кто либо не знал о существовании универсальной функции `TRIM`, либо не имел возможности задействовать ее для удаления пробелов с обеих сторон.

Протестируем данные функции на переменной, содержащей пробелы в начале и в конце. Пример этот может показаться надуманным, но уверяю вас: если вам когда-нибудь доведется разрабатывать интерфейс для ввода данных вручную, вы неизбежно столкнетесь с проблемой начальных и конечных пробелов.

Наша переменная будет содержать три пробела в начале строки и два в конце. В запросе мы также зададим информативные имена столбцам (результаты после удаления пробелов продемонстрированы на рис. 14.4):

```
SET @Word = '   word  ';
SELECT
    @Word AS WordAsEntered,
    LTRIM(@Word) AS WordLTRIM,
    RTRIM(@Word) AS WordRTRIM,
    TRIM(@Word) AS WordTRIM;
```

WordAsEntered	WordLTRIM	WordRTRIM	WordTRIM
word	word	word	word

Рис. 14.4. Результаты выборки строки ‘ word ‘, содержащей три начальных и два конечных пробела, с использованием трех TRIM-функций. LTRIM удаляет пробелы слева, RTRIM — справа, а TRIM — с обеих сторон

Признаться, просто взглянув на результаты, трудно различить пробелы. Однако можно воспользоваться еще одной функцией, чтобы удостовериться, что начальные и конечные пробелы действительно удалены: это функция `LENGTH`. Она возвращает длину строки в символах, включая пробелы.

В следующем примере нам придется вспомнить арифметику начальной школы —сложение и вычитание. Слово *word* («слово») состоит из 4 символов; добавив три пробела в начале и два в конце, получим строку общей длиной 9 символов.

После удаления трех начальных пробелов длина строки, возвращаемой функцией `LTRIM`, должна составить 6 символов (9 минус 3). А после удаления двух конечных пробелов функция `RTRIM` должна вернуть строку длиной 7 символов (9 минус 2). Наконец, при полном удалении пробелов в начале и в конце длина результирующей строки со словом *word* составит ровно 4 символа.

Проверим это при помощи SQL и функции `LENGTH`, примененной к результатам функций удаления пробельных символов. Все верно, функции можно вкладывать друг в друга! Но не забывайте: сначала выполняется самая глубокая, вложенная функция — та, что внутри. Результаты выполнения кода проиллюстрированы на рис. 14.5:

```
SET @Word = '   word  ';
SELECT
    LENGTH(@Word) AS WordAsEnteredLength,
    LENGTH(LTRIM(@Word)) AS WordLTRIMLength,
    LENGTH(RTRIM(@Word)) AS WordRTRIMLength,
    LENGTH(TRIM(@Word)) AS WordTRIMLength;
```

WordAsEnteredLength	WordLTRIMLength	WordRTRIMLength	WordTRIMLength
9	6	7	4

Рис. 14.5. Выборка длины строк посредством функции LENGTH после применения различных TRIM-функций к строке 'word'. Удаление начальных и конечных пробелов нередко сложно отследить визуально, и функция LENGTH помогает убедиться в корректности результатов

Еще раз подчеркну: убирать лишние пробелы — не прихоть, а необходимость, поскольку хранение и отображение строк с ведущими пробелами, как правило, нежелательно. Эти пробелы не только занимают лишнее место в базе данных, но и могут вызывать проблемы при выполнении стандартных операций, таких как фильтрация и сортировка.

> **НА ЗАМЕТКУ** В SQL Server отсутствует функция `LENGTH`. Вместо нее для определения длины строки следует задействовать функцию `LEN`.

14.2.3. Прочие строковые функции

Несмотря на то что каждая СУБД предоставляет собственный набор строковых функций, существуют функции, получившие столь широкое распространение, что они реализованы практически во всех системах управления реляционными базами данных. Ряд наиболее универсальных функций приведен в табл. 14.1.

Таблица 14.1. Универсальные строковые функции, доступные в большинстве СУБД, и их определения

Имя функции	Описание
`LEFT`	Извлекает указанное количество символов с начала строки
`REPLACE`	Выполняет поиск и замену подстроки значений в строке
`RIGHT`	Извлекает указанное количество символов с конца строки
`SUBSTRING`	Извлекает подстроку заданной длины, начиная с указанной позиции

14.3. Функции для работы с датой и временем

С помощью функций можно не только парсинговать строковые значения, но и извлекать отдельные компоненты из значений даты и времени. Большинство реляционных СУБД предоставляют функции с прозрачными названиями — YEAR (год), MONTH (месяц), DAY (день), HOUR (час), MINUTE (минута), SECOND (секунда), — которые позволяют получить соответствующие составляющие даты или времени. Это особенно полезно, когда требуется найти данные на основе одного или нескольких компонентов даты.

14.3.1. Функции даты, возвращающие числовые значения

Допустим, нам нужно получить список всех идентификаторов заказов за 2015 год, подобный показанному на рис. 14.6. Для этого можно воспользоваться функцией YEAR в запросе, который проверяет год для всех заказов в таблице orderheader и возвращает запрошенные данные. Включим в запрос столбцы OrderID и OrderDate:

```
SELECT
    OrderID,
    OrderDate
FROM orderheader
WHERE YEAR(OrderDate) = 2015;
```

То же самое можно проделать с любой из шести других функций, связанных с датой и временем. Скорее всего, вы уже думаете, как применять переменные в подобных запросах, чтобы сделать фильтрацию еще гибче. Или, вероятно, уже догадались, что функции эти допустимо использовать не только для фильтрации, но и для выборки отдельных частей значения даты и времени.

OrderID	OrderDate
1001	2015-06-01 00:00:00
1002	2015-06-15 00:00:00
1003	2015-07-03 00:00:00
1004	2015-08-12 00:00:00
1005	2015-09-05 00:00:00
1006	2015-11-02 00:00:00
1007	2015-11-15 00:00:00
1008	2015-11-22 00:00:00

Рис. 14.6. Результаты выборки всех значений OrderID и OrderDate из заказов, сделанных в 2015 году, полученные при помощи функции YEAR

Давайте применим все эти функции к значению OrderDate первого заказа. Идентификатор (OrderID) этого заказа равен 1001 (рис. 14.7):

```
SELECT
    OrderDate,
    YEAR(OrderDate),
    MONTH(OrderDate),
    DAY(OrderDate),
    HOUR(OrderDate),
    MINUTE(OrderDate),
    SECOND(OrderDate)
FROM orderheader
WHERE OrderID = 1001;
```

OrderDate	YEAR(OrderDate)	MONTH(OrderDate)	DAY(OrderDate)	HOUR(OrderDate)	MINUTE(OrderDate)	SECOND(OrderDate)
2015-06-01 00:00:00	2015	6	1	0	0	0

Рис. 14.7. Выборка всех компонентов даты и времени значения OrderDate первого заказа в таблице orderheader

Сейчас эти функции могут показаться не слишком полезными, тем не менее в главе 15 рассматриваются реальные сценарии применения компонентов даты и времени для выполнения различных вычислений. Кроме того, перечисленные функции — не единственные для работы с датой и временем. В разделе 14.3.2 описаны еще две функции, заслуживающие вашего внимания.

14.3.2. Функции даты, возвращающие строковые значения

Функции `DAYNAME` и `MONTHNAME` предоставляют дополнительную информацию о датах, возвращая строковые названия дня недели и месяца для заданной даты. Хотя название месяца легко определить по его числовому значению, маловероятно, что вы вспомните, в какой именно день недели был оформлен заказ. Измените предыдущий запрос, чтобы задействовать эти функции и получить нужные сведения (результаты приведены на рис. 14.8):

```
SELECT
    OrderDate,
    YEAR(OrderDate),
    MONTHNAME(OrderDate),
    DAY(OrderDate),
    DAYNAME(OrderDate)
FROM orderheader
WHERE OrderID = 1001;
```

OrderDate	YEAR(OrderDate)	MONTHNAME(OrderDate)	DAY(OrderDate)	DAYNAME(OrderDate)
2015-06-01 00:00:00	2015	June	1	Monday

Рис. 14.8. Выборка, демонстрирующая применение функций MONTHNAME и DAYNAME для определения названий месяца и дня недели первого заказа в таблице orderheader

СОВЕТ Держите эти функции в уме при подготовке отчетов — они помогут красиво оформить даты и указать названия месяцев или дней недели, как того требует ваш формат отчетности.

14.3.3. Прочие функции для работы с датой и временем

В табл. 14.2 собраны наиболее востребованные функции для работы с датой и временем, которые поддерживаются подавляющим большинством реляционных СУБД.

Таблица 14.2. Универсальные функции для даты и времени, доступные в большинстве СУБД

Имя функции	Что она возвращает
DATE	Только дату из значения даты и времени
DAYOFWEEK	Числовое представление дня недели для заданной даты
DAYOFYEAR	Числовое представление дня в году для заданной даты
LAST_DAY	Последний день месяца для заданной даты
QUARTER	Квартал в году для заданной даты
TIME	Только время из значения даты и времени
WEEKOFYEAR	Номер недели в году для заданной даты

Можете себе представить, каким широким спектром применения обладают эти функции при работе со значениями даты и времени, хранящимися в базе данных. Но что делать, если вам необходимо получить сведения о текущем моменте времени — к примеру, кто, откуда и когда инициировал запрос? Хорошая новость: для таких случаев в современных СУБД также предусмотрены специальные функции.

14.4. Информационные функции

Каждая СУБД реализует собственный набор средств для ответа на вопросы «кто», «где» и «когда» в контексте выполнения SQL-запроса, хотя в большинстве случаев применяются и универсальные функции. Начнем с «когда» — то есть с установления точного момента времени, в который выполняется запрос.

14.4.1. Сведения о дате и времени

Для определения текущего момента времени («сейчас») в SQL-запросах, как правило, используется функция `CURRENT_TIMESTAMP`. Функция возвращает текущее время на сервере, где размещена ваша база данных; это, скорее всего, ваш собственный компьютер с локальной установкой базы данных `sqlnovel`. Формат

даты и времени стандартный, к которому мы уже привыкли: [год-месяц-день час:минута:секунда].

Обратите внимание: при вызове функции CURRENT_TIMESTAMP после нее необходимо ставить круглые скобки, несмотря на отсутствие аргументов, — это правило для всех функций в SQL без исключения:

```
SELECT CURRENT_TIMESTAMP() AS RightNow;
```

Не стану приводить иллюстрацию, поскольку мои результаты будут отличаться от ваших, да и у вас при каждом запуске этой функции будут генерироваться разные значения.

К практике!

Определите текущее время на вашем сервере базы данных, прибегнув к функции CURRENT_TIMESTAMP.

CURRENT_TIMESTAMP — не единственная функция для работы с текущим временем. Для получения только даты или только времени большинство реляционных СУБД предоставляют функции CURRENT_DATE и CURRENT_TIME соответственно. Можете протестировать эти функции при помощи следующего запроса:

```
SELECT
    CURRENT_DATE() AS CurrentDate,
    CURRENT_TIME() AS CurrentTime;
```

Поскольку CURRENT_TIMESTAMP звучит длинновато для имени функции, многие СУБД предоставляют краткий синоним. В MySQL это функция NOW. Можете запустить следующий SQL-запрос и удостовериться в том, что обе функции возвращают одинаковое значение:

```
SELECT
    CURRENT_TIMESTAMP() AS RightNow,
    NOW() AS AlsoRightNow;
```

Последнее, что следует отметить об этих функциях — их можно применять в сочетании с другими функциями, рассмотренными в нашем руководстве. В частности, если необходимо получить название текущего дня недели, то можно выяснить его посредством такого запроса:

```
SELECT DAYNAME(NOW()) AS CurrentDayOfWeek;
```

14.4.2. Сведения о подключении

Перейдем к последнему набору функций, которые предназначены для идентификации пользователя и контекста подключения. Занимаясь по этой книге, вы работаете лишь с одной базой данных, однако в профессиональной практике вам, скорее всего, придется переключаться между разными БД, чтобы запрашивать данные из различных источников. Если вдруг вам потребуется уточнить, к какой именно базе данных вы сейчас подключены, просто вызовите функцию `DATABASE` — она подскажет (результат показан на рис. 14.9):

```
SELECT DATABASE();
```

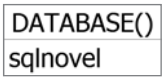


Рис. 14.9. Активная база данных, используемая сейчас, — sqlnovel

Кроме того, можно задействовать несколько учетных записей для подключения к базе данных. Потребность в этом нередко возникает, когда вы переключаетесь между своей личной учетной записью и учетной записью, используемой конкретным приложением или системой отчетности, — к примеру, в целях тестирования. Определить имя пользователя, связанное с актуальным подключением, можно при помощи функции `CURRENT_USER`:

```
SELECT CURRENT_USER();
```

> **НА ЗАМЕТКУ** MySQL поддерживает две синонимичные функции — `USER` и `CURRENT_USER`, которые возвращают информацию об имени пользователя. Большинство СУБД включают `CURRENT_USER`, но не `USER`.

Наконец, как мы помним из инструкций по установке в главе 1, программное ядро СУБД MySQL регулярно обновляется, и каждое обновление фиксируется в номере версии. При подключении к базе данных номер версии, как правило, не отображается. Для получения сведений о текущей версии следует воспользоваться функцией `VERSION`:

```
SELECT VERSION();
```

Версия выводится в виде строки из трех чисел, разделенных точками; первое число обозначает основную версию. Для выполнения упражнений, представленных в этом учебном пособии, рекомендуется пользоваться MySQL версии 8 или выше.

Что ж, для этого урока новых функций вполне достаточно. В главе 15 вас ждет еще больше функций, позволяющих преобразовывать значения и выполнять различные вычисления во множестве практических сценариев.

14.5. Практическое занятие

1. Ранее мы отметили, что большинство функций принимают какой-либо аргумент, однако с CURRENT_TIMESTAMP параметры не применялись. Что произойдет, если передать этой функции значение — скажем, число 2?
2. Как посчитать количество заказов, оформленных в понедельник, при помощи функций для работы с датой, рассмотренных в этой главе?
3. Какие две функции можно задействовать для определения самого длинного заглавия книги в таблице title? И как составить запрос, который выведет это название?

14.6. Ответы

1. В качестве аргументов функция CURRENT_TIMESTAMP принимает целые числа; они задают точность возвращаемого значения даты и времени. Указание значения 2 включает в результат миллисекунды:

```
SELECT CURRENT_TIMESTAMP(2);
```

2. Для подсчета заказов, оформленных в понедельник, можно прибегнуть к функции DAYNAME, отфильтровав с ее помощью строки в таблице orderheader по условию, при котором день недели даты заказа равен Monday:

```
SELECT COUNT(OrderID) AS MondayOrders
FROM orderheader
WHERE DAYNAME(OrderDate) = 'Monday';
```

3. Чтобы определить самое длинное название в таблице title, можно воспользоваться функциями MAX и LENGTH:

```
SELECT MAX(LENGTH(titlename))
FROM title;
```

   Теперь внимание: чтобы вывести именно это название (а не только его длину), понадобится подзапрос. Можно отфильтровать заглавия книг по длине, определенной предыдущим запросом, используя подзапрос в предикате для нахождения самого длинного названия книги с максимальной длиной, соответствующей значению MAX:

```
SELECT TitleName
FROM title
WHERE LENGTH(TitleName) =
    (SELECT MAX(LENGTH(TitleName))
    FROM title);
```

15 Объединение и вычисление значений при помощи функций

В предыдущей главе мы рассмотрели ряд функций, позволяющих извлекать отдельные фрагменты данных. Здесь же речь пойдет еще о нескольких функциях, дающих возможность объединять и комбинировать значения различными способами и даже выполнять вычисления. Вне зависимости от специфики ваших задач вы обязательно найдете в этой главе что-нибудь полезное. В частности, если вы работаете с адресной базой, то откроете для себя, как объединить в один столбец название улицы, города и другие компоненты адреса, взятые из отдельных столбцов. А если вы занимаетесь финансовой отчетностью, то узнаете, как проследить за тем, чтобы все суммы выводились с нужным количеством знаков после запятой.

В этой главе мы обратимся к этим и другим подобным сценариям. А начнем с функций, предназначенных для объединения значений.

15.1. Объединение строковых значений

Я еще не касался этой темы, но пора отметить, что язык SQL позволяет осуществлять базовые арифметические операции, такие как сложение чисел. Вот пример простого сложения (рис. 15.1):

```
SELECT 1 + 1;
```

1 + 1
2

Рис. 15.1. Результат вычисления 1+1 в SQL

Если вы работаете исключительно с числовыми данными, такая возможность, безусловно, пригодится. Но что делать, если нужно объединить строковые значения? К сожалению, как видно по рис. 15.2, применение знака «плюс» (+) не позволяет объединить строки так, чтобы получить желаемый результат:

```
SELECT 'Я' + ' ' + 'люблю' + ' ' + 'книги!';
```

'Я' + ' 'люблю' + ' '+ 'книги!'
0

Рис. 15.2. Результат объединения строк оператором «плюс», который не соединяет все значения в одну строку

Вместо ожидаемого результата «Я люблю книги!» возвращается 0. Это означает, что операция не может быть выполнена, поскольку MySQL не знает, как «складывать» слова с математической точки зрения.

> **НА ЗАМЕТКУ** В SQL Server знак «плюс» действительно можно применять для объединения строк, однако такой прием не работает в большинстве реляционных СУБД.

Для описания действия, которое мы пытаемся выполнить, — соединения двух или более строковых значений в одно — в английском языке имеется специальный глагол: concatenate (*конкатенировать*, «последовательно соединять», «склеивать»). Это важно знать, поскольку функция, которую мы задействуем для объединения строк, называется `CONCAT`.

15.1.1. Функция CONCAT

Для конкатенации, «склеивания» строковых значений в MySQL применяется функция `CONCAT`, принимающая в качестве аргументов список строк, разделенных запятыми. Применительно к приведенному выше запросу функцию эту следует использовать следующим образом — результат отображен на рис. 15.3:

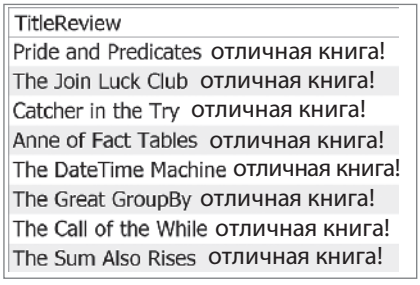
```
SELECT CONCAT('Я', ' ', 'люблю', ' ', 'книги!');
```

CONCAT('Я', ' ', 'люблю' ' ', 'книги!'
Я люблю книги!

Рис. 15.3. Результат применения функции CONCAT для получения единого строкового значения «Я люблю книги!» из нескольких строк

Возможно, пример выглядит несколько надуманным, но, как вы вскоре убедитесь, функция `CONCAT` куда мощнее, чем кажется. Ведь строковые значения существуют не только в виде литералов, использованных в этом запросе. Они также хранятся в столбцах таблиц или в переменных. Любые из этих строковых значений можно так же легко объединять с помощью `CONCAT`. Для примера создадим переменную для отзыва, присвоив ей значение `' — отличная книга!'`, и объединим ее со всеми заглавиями произведений из таблицы `title` (результаты показаны на рис. 15.4):

```
SET @Review = ' отличная книга!';
SELECT CONCAT(TitleName, @Review) AS TitleReview
FROM title;
```



TitleReview
Pride and Predicates отличная книга!
The Join Luck Club отличная книга!
Catcher in the Try отличная книга!
Anne of Fact Tables отличная книга!
The DateTime Machine отличная книга!
The Great GroupBy отличная книга!
The Call of the While отличная книга!
The Sum Also Rises отличная книга!

Рис. 15.4. Результат объединения значений столбца TitleName таблицы title со строковой переменной для формирования единого выходного столбца

Функцию `CONCAT` можно использовать даже с числами или датами. Однако при этом следует помнить, что все значения автоматически преобразуются в строковый тип, обеспечивая тем самым возможность объединения данных разных типов. Такое преобразование иногда приводит к неожиданным результатам при сортировке конкатенированных значений, содержащих числа или даты.

Проверим это на практике: последовательно объединим значения столбцов Price и `TitleName` из таблицы `title`, как показано на рис. 15.5. Для удобства чтения разделим их пробелами. Поскольку мы хотим отсортировать результат от наименьшего значения к наибольшему, явно укажем сортировку по возрастанию посредством ключевого слова `ASC`:

```
SELECT CONCAT(Price, ' ', TitleName) AS PriceAndTitle
FROM title
ORDER BY PriceAndTitle ASC;
```



PriceAndTitle
10.95 The Great GroupBy
12.95 Anne of Fact Tables
7.95 The DateTime Machine
7.95 The Sum Also Rises
8.95 Catcher in the Try
8.95 The Call of the While
9.95 Pride and Predicates
9.95 The Join Luck Club

Рис. 15.5. Результаты объединения значений Price и TitleName с пробелом в качестве разделителя, отсортированные по возрастанию результирующей строки

В чем тут дело? Численно значения 10.95 и 12.95 должны находиться в конце при сортировке по возрастанию, однако здесь они оказались в начале. Происходит это потому, что числовые значения преобразованы в строки для конкатенации и их упорядочивание осуществляется лексикографически, то есть по кодам символов. В этом случае первым символом в этих соединенных строках является «1», который в алфавитном порядке идет раньше первых символов остальных значений — «7», «8» и «9».

Чтобы получить желаемый порядок сортировки в выдаче, следует применять предложение ORDER BY с указанием конкретного столбца, по которому требуется выстраивать результаты, — в нашем случае это столбец Price. Даже если сам столбец Price не входит в выборку, его можно задействовать для сортировки, как показано в полученных строках (рис. 15.6):

```
SELECT CONCAT(Price, ' ', TitleName) AS
     PriceAndTitle
FROM title
ORDER BY Price;
```

PriceAndTitle
7.95 The DateTime Machine
7.95 The Sum Also Rises
8.95 Catcher in the Try
8.95 The Call of the While
9.95 Pride and Predicates
9.95 The Join Luck Club
10.95 The Great GroupBy
12.95 Anne of Fact Tables

Рис. 15.6. Результаты объединения значений Price и TitleName с пробелом в качестве разделителя, отсортированные по возрастанию значения Price

Напомню на всякий случай, о чем уже говорил в главе 4: отсутствие столбца в результирующем наборе вовсе не означает, что по нему нельзя сортировать в предложении ORDER BY. Допускается упорядочение по любому значению или сочетанию значений из таблицы, указанной в предложении FROM, при условии что вы не применяете агрегацию с предложением GROUP BY. Следовательно, конкатенированные значения можно скомпоновать не только по цене (Price) или названию (TitleName), но также по идентификатору названия или дате публикации.

К практике!

Возьмите приведенный выше запрос, выполняющий выборку и конкатенацию значений Price и TitleName, и добавьте символ '$' перед ценой для указания валюты. Если вы по-прежнему сортируете по Price, а не по объединенной строке, порядок останется ожидаемым — по возрастанию. При желании скомпонуйте выборку по TitleName или любому другому столбцу таблицы title, и порядок выдачи результатов тут же изменится.

Вам нечасто придется конкатенировать значения разных типов, что мы только что продемонстрировали, но если вы работаете с данными клиентов, то, скорее всего, вам понадобится формировать вывод, в котором имя и фамилия объединены в один столбец. Подобные составные значения могут применяться, в частности, в рассылках, шаблонах электронных писем, бейджах и т. п.

Как вы, вероятно, уже догадались, соединить имя и фамилию при помощи функции CONCAT проще простого. Объединим значения столбцов FirstName и LastName из таблицы author, разделив их пробелом и присвоим полученному столбцу псевдоним AuthorName (результат отображен на рис. 15.7):

```
SELECT CONCAT(FirstName, ' ', LastName) AS AuthorName
FROM author;
```

AuthorName
Paul Tripp
Doug Li
Jen Strong
Jorge Guerra
Robert Davidson
Gail Shawn
Rebecca Miller
Andy Melkin
Buck Fernandez
Chris Walenski
Deepthi Mahadevan

Рис. 15.7. Результаты конкатенации столбцов FirstName и LastName таблицы author в объединенное значение с разделителем-пробелом

15.1.2. Функция CONCAT_WS

Функция CONCAT — замечательный инструмент, однако для последовательного соединения множества значений с одинаковым разделителем есть еще одна удобная функция — CONCAT_WS. Большинство (хотя не все) СУБД поддерживают ее, чтобы немного упростить конкатенацию строк. Функция CONCAT_WS похожа на CONCAT с тем лишь отличием, что первое указанное значение является разделителем, который ставится между всеми последующими значениями. Выборку, приведенную на рис. 15.7, можно получить при помощи следующего запроса, использующего функцию CONCAT_WS с пробелом в качестве первого значения:

```
SELECT CONCAT_WS(' ', FirstName, LastName) AS AuthorName
FROM author;
```

Применение CONCAT_WS не делает данный конкретный SQL-запрос короче. Тем не менее, если требуется «склеить» более двух значений при помощи пробела или иного разделителя, функция CONCAT_WS предпочтительнее CONCAT, так как она позволяет указать разделитель всего лишь раз.

Поскольку в таблице author есть столбец для второго имени (MiddleName), попробуем при помощи CONCAT_WS добавить и его, чтобы вывести полное имя каждого автора (результаты представлены на рис. 15.8):

```
SELECT CONCAT_WS(' ', FirstName, MiddleName, LastName) AS AuthorName
FROM author;
```



AuthorName
Paul K Tripp
Doug Li
Jen Strong
Jorge Armando Guerra
Robert Grant Davidson
Gail Anne Shawn
Rebecca Miller
Andy Melkin
Buck Fernandez
Chris Walenski
Deepthi Mahadevan

Рис. 15.8. Выборка конкатенированных значений FirstName, MiddleName и LastName для всех строк таблицы author с применением функции CONCAT_WS

Функция `CONCAT_WS` обеспечивает эффективную конкатенацию имен авторов, что особенно важно при наличии NULL-значений в столбце `MiddleName` — `CONCAT_WS` автоматически обрабатывает эти значения и заменяет их пустыми строками при объединении, чего обычный `CONCAT`, как правило, не умеет.

При использовании `CONCAT` значения null не преобразуются в пустые строки, что может вызвать определенные сложности. Как мы узнали из главы 7, пустые значения представляют собой отсутствие данных, поэтому при конкатенации любого значения, не являющегося NULL, с NULL-значением результат всегда будет NULL.

Рассмотрим этот случай на практике. Если мы попытаемся получить те же результаты, что отображены на рис. 15.8, при помощи `CONCAT`, то будем разочарованы результатами. Как показывает рис. 15.9, любая объединенная строка с пустым значением для `MiddleName` вернет результат NULL:

```
SELECT CONCAT(FirstName, ' ', MiddleName, ' ', LastName) AS AuthorName
FROM author;
```



AuthorName
Paul K Tripp
NULL
NULL
Jorge Armando Guerra
Robert Grant Davidson
Gail Anne Shawn
NULL
NULL
NULL
NULL
NULL

Рис. 15.9. Выборка конкатенированных значений FirstName, MiddleName и LastName для всех строк таблицы author с применением функции CONCAT. Строки с результатом NULL вызваны пустыми значениями в столбце MiddleName

15.1.3. Функция COALESCE

Если необходимо учитывать потенциально возможные NULL-значения при последовательном соединении строк посредством функции `CONCAT` или любой другой функции, не преобразующей такие значения, язык SQL предлагает дополнительную функцию `COALESCE`. Она поддерживается всеми СУБД и может применяться для обработки NULL-значений в операциях конкатенации. Функция `COALESCE` принимает список значений и возвращает из него первое значение, отличное от NULL.

В примере на рис. 15.10 мы задействуем `COALESCE` со значением `MiddleName` в качестве первого параметра и пустой строкой в качестве второго параметра. Поскольку известно, что пустая строка не является NULL, `COALESCE` гарантированно вернет либо существующие значения `MiddleName`, либо пустую строку для NULL-значений.

В соответствии с вышеизложенным, заменим выбор `MiddleName` в предыдущем запросе на функцию `COALESCE`, чтобы исключить NULL-значения из результатов (рис. 15.10):

```
SELECT CONCAT(
    FirstName, ' ', COALESCE(MiddleName, ''), ' ', LastName
    ) AS AuthorName
FROM author;
```

AuthorName
Paul K Tripp
Doug Li
Jen Strong
Jorge Armando Guerra
Robert Grant Davidson
Gail Anne Shawn
Rebecca Miller
Andy Melkin
Buck Fernandez
Chris Walenski
Deepthi Mahadevan

Рис. 15.10. Выборка конкатенированных значений FirstName, MiddleName и LastName для всех строк таблицы author с применением функции CONCAT. Функция COALESCE заменяет NULL-значения для MiddleName на пустые строки

Хотя `COALESCE` устранила проблемы, вызванные NULL-значениями столбца `MiddleName`, при внимательном рассмотрении можно заметить, что в исправленных строках теперь присутствуют два пробела между именем и фамилией. В таком результате нет ничего принципиально ошибочного, однако `CONCAT_WS` в этом плане оказалась аккуратнее.

Если вы работаете с СУБД, не поддерживающей `CONCAT_WS`, и вам необходимо соединить несколько строк, заменив двойные пробелы на одинарные, это вполне

осуществимо. Вам просто потребуется другая распространенная функция, обеспечивающая преобразование строковых значений.

15.2. Преобразование значений

Теперь, когда мы научились последовательно соединять несколько значений для создания нового, рассмотрим ряд других функций, предназначенных для модификации значений.

15.2.1. Функция REPLACE

Функция REPLACE позволяет заменить любую последовательность из одного или нескольких символов в строке на другую. Такая последовательность называется *подстрокой* (substring), поскольку она является частью всей оцениваемой строки.

Функция REPLACE принимает три аргумента в следующем порядке: строка, в которой выполняется поиск, подстрока, которую необходимо заменить, и подстрока, которая служит заменой.

Приведем простой пример. Чтобы исправить написание американского слова *check* («чек») на британский вариант *cheque*, достаточно в слове *check* заменить буквы *ck* на *que*, как показано на рис. 15.11. Мы можем сформировать следующий запрос, задействовав функцию REPLACE:

```
SELECT REPLACE('check', 'ck', 'que');
```

REPLACE('check', 'ck', 'que')
cheque

Рис. 15.11. Результат замены сочетания ck на que для приведения американского варианта написания слова check к британскому

Возвращаясь к проблеме замены двойных пробелов на один в соединенных именах, обозначенной в разделе 15.1.3, мы можем добавить в запрос функцию REPLACE, чтобы заменить любое вхождение двойного пробела на одиночный пробел (результаты проиллюстрированы на рис. 15.12):

```
SELECT REPLACE(
    CONCAT(
        FirstName, ' ', COALESCE(MiddleName, ''), ' ', LastName
    )
    , '  ', ' ') AS AuthorName
FROM author;
```

Для решения этой конкретной задачи потребовалось три функции. По мере накопления опыта работы с SQL такая стратегия перестанет казаться вам чем-то необычным; каждая функция выполняет определенную задачу, и, возможно,

вам нередко придется творчески подходить к применению нескольких функций в одном запросе.

AuthorName
Paul K Tripp
Doug Li
Jen Strong
Jorge Armando Guerra
Robert Grant Davidson
Gail Anne Shawn
Rebecca Miller
Andy Melkin
Buck Fernandez
Chris Walenski
Deepthi Mahadevan

Рис. 15.12. Результаты объединения имен авторов с заменой NULL-значений на пустую строку функцией COALESCE и последующей заменой двойных пробелов на одинарные функцией REPLACE

15.2.2. Функции CONVERT и CAST

Для преобразования значений из одного типа данных в другой, как правило, служат две функции: `CAST` и `CONVERT`. Обе приводят значение одного типа данных к другому указанному типу. В главе 13 мы обнаружили, что MySQL умеет делать это автоматически, однако далеко не все реляционные СУБД могут похвалиться тем же. Во многих системах для выполнения любых преобразований типов данных необходимо воспользоваться одной из этих двух функций. Кроме того, хотя многие СУБД предлагают и `CAST`, и `CONVERT`, некоторые поддерживают лишь одну из них. К счастью, MySQL поддерживает обе функции, поэтому вы можете попрактиковаться в написании запросов для них обеих.

Возьмем простой пример. Текущие значения даты хранятся вместе со временем, однако для нашей БД время не важно, поскольку мы не фиксируем данные о часах, минутах или секундах. Все временные значения `PublicationDate` содержат только нули. Для отображения исключительно даты из `PublicationDate` в таблице `title` нужно задействовать одну из означенных функций.

Сначала рассмотрим функцию `CONVERT`, которая принимает два аргумента: значение, подлежащее изменению, и тип данных, к которому требуется его привести. В нашем случае мы хотим преобразовать тип данных из `DATETIME` (в котором данные хранятся в таблице `title`) в `DATE`. Для сравнения выполним выборку как исходного `PublicationDate`, так и преобразованного значения (результаты представлены на рис. 15.13):

```
SELECT
    PublicationDate,
    CONVERT(PublicationDate, DATE) AS PublicationDateNoTime
FROM title;
```

PublicationDate	PublicationDateNoTime
2015-04-30 00:00:00	2015-04-30
2016-02-06 00:00:00	2016-02-06
2017-04-03 00:00:00	2017-04-03
2018-01-12 00:00:00	2018-01-12
2019-02-04 00:00:00	2019-02-04
2019-12-23 00:00:00	2019-12-23
2020-03-14 00:00:00	2020-03-14
2021-11-12 00:00:00	2021-11-12

Рис. 15.13. Результаты выборки PublicationDate из таблицы title с преобразованием в целях удаления из значения времени

Доступные для преобразования типы данных зависят от конкретной СУБД. Однако имейте в виду, что преобразование строковых значений, таких как имена, в числовые или даты в большинстве случаев не дает полезного результата и может даже привести к ошибке.

К практике!

Воспользуйтесь SQL-кодом из этого раздела, чтобы преобразовать `PublicationDate` таблицы title в тип `DATE`; затем попробуйте также привести названия книг (`TitleName`) к типу `DATE`. В MySQL для преобразованных названий вы получите значения null, поскольку строку нельзя корректно преобразовать в дату.

Функция `CAST` работает так же, как `CONVERT`, но несколько отличается в части синтаксиса. Так, вместо двух отдельных параметров мы записываем в скобках своего рода фразу, где вместо запятой употребляется слово `AS`. Вот как будет выглядеть последний пример с функцией `CAST`:

```
SELECT
    PublicationDate,
    CAST(PublicationDate AS DATE) AS PublicationDateNoTime
FROM title;
```

Выполнение запроса возвращает те же результаты, что приведены на рис. 15.13.

Какую функцию выбрать? Вне зависимости от личных пристрастий, следует, по всей видимости, отдать предпочтение `CAST`, поскольку она считается стандартной функцией SQL, тогда как `CONVERT` таковой не является. Поэтому `CAST` вы почти наверняка встретите в списке поддерживаемых функций любой СУБД, что обеспечивает лучшую переносимость SQL-кода между системами.

НА ЗАМЕТКУ Хотя по существу обе эти функции выполняют одно и то же действие, `CONVERT` нередко располагает расширенными возможностями, включая передачу третьего аргумента для форматирования выходных данных. Проверьте документацию вашей СУБД, чтобы уточнить, поддерживается ли сама функция `CONVERT` и доступны ли дополнительные параметры.

15.3. Функции для математических вычислений

В главе 12 мы познакомились с агрегатными функциями, такими как MIN, MAX, AVG и SUM. Эти агрегатные функции являются разновидностью математических функций, применимых к числовым значениям для выполнения различных расчетов.

Существует великое множество функций для работы с числами, при этом большинство из них выполняет специфические математические задачи, такие как вычисление квадратных корней, логарифмов или тангенса. В настоящий момент давайте сосредоточимся на одной математической функции, которую большинство пользователей сочтет весьма полезной применительно к ряду сценариев из реальной практики.

Функция ROUND обеспечивает решение типовых задач, требующих округления значений до меньшего количества десятичных разрядов — в частности, для преобразования десятичных денежных значений в целочисленные. К примеру, коммерческие организации обычно не оперируют в публичной отчетности точными величинами вроде 1 000 000.32 доллара, предпочитая округлять цифры (1 000 000 долларов).

И хотя база данных sqlnovel не может похвастаться продажами на миллион долларов, мы тем не менее можем применить функцию ROUND для получения общего объема продаж за год в долларах без указания центов (долей доллара). Давайте посмотрим, как воспользоваться функцией ROUND для вычисления целочисленного значения общего объема продаж по всем заказам. В главе 12 мы уже выполняли расчет этого значения посредством функции SUM. Здесь же мы добавим второй столбец, который округлит эту сумму с помощью функции ROUND. Для сравнения выберем точную и округленную сумму (результаты отображены на рис. 15.14):

```
SELECT
    SUM(Quantity * ItemPrice) AS TotalOrderValue,
    ROUND(SUM(Quantity * ItemPrice)) AS TotalOrderValueRounded
FROM orderitem;
```

TotalOrderValue	TotalOrderValueRounded
573.50	574

Рис. 15.14. Совокупная стоимость всех продаж в долларах и центах наряду с тем же показателем, округленным до целочисленного значения

Обратите внимание, что в этом примере округление увеличило значение, поскольку любое число, равное или большее 0.50, округляется в большую сторону, тогда как любое число меньше 0.50 округляется в меньшую сторону. В этом легко

убедиться, прибегнув к простой инструкции SELECT, которая округляет значение 573.49 до меньшего значения.

К практике!

Выполните SELECT ROUND(573.49) и удостоверьтесь в том, что это значение округляется до 573.

Еще одна деталь, на которую стоит обратить внимание при работе с функцией ROUND: она принимает два аргумента. Первый аргумент — это числовое значение, которое мы использовали в этой главе. Второй аргумент необязательный — он задает количество знаков после запятой, до которых должно выполняться округление. Если значение второго параметра не указано, число округляется до целого (то есть до нуля знаков после запятой).

При работе с денежными значениями вам скорее всего придется задействовать ROUND с обоими параметрами. Допустим, требуется рассчитать сумму налога с продаж для книги, цена которой составляет 9.95 доллара. При ставке налога 5 % можно просто умножить 9.95 на 0.05. Однако результат такого вычисления даст значение налога с количеством знаков больше двух. Так как покупателям нельзя выставлять счет за доли цента, целесообразно передать значение 2 во второй параметр функции ROUND, чтобы округлить налог в соответствии с форматом денежных значений. Для проверки результатов выполните следующий запрос, который покажет как рассчитанный, так и округленный налог (рис. 15.15).

```
SELECT
    9.95 * .05 AS CalculatedTax,
    ROUND(9.95 * .05, 2) AS CalculatedTaxRounded
```

CalculatedTax	CalculatedTaxRounded
0.4975	0.50

Рис. 15.15. Результат применения ROUND для округления числа 0.4975 с четырех десятичных знаков до двух, что приводит к незначительному увеличению значения

Для работы с более сложными вычислениями в вашей СУБД наверняка имеются десятки дополнительных функций для математических операций. В табл. 15.1 перечислены некоторые распространенные математические функции, доступные практически в каждой системе управления реляционными БД.

НА ЗАМЕТКУ В SQL Server функция CEIL заменена на CEILING.

На протяжении предыдущих глав мы изучали способы применения SQL для выборки данных из таблиц БД и представления результатов различными способами. В главе 16 мы начнем рассматривать методы изменения содержимого таблиц средствами управления данными.

Таблица 15.1. Универсальные математические функции, доступные в большинстве СУБД

Имя функции	Что она производит
ABS	Абсолютное значение числа
CEIL	Округление вверх до ближайшего целого
FLOOR	Округление вниз до ближайшего целого
MOD	Остаток от деления одного числа на другое (модуль)
SQRT	Квадратный корень числа

15.4. Практическое занятие

1. Сформируйте выборку из таблицы `author`, содержащую единственный столбец с псевдонимом `AuthorName`, в котором представлены все имена и фамилии авторов в формате «Фамилия, Имя» (например, «Iannucci, Jeff»).
2. Составьте запрос, выводящий фразу о дате публикации первой книги, скажем: «Первая книга опубликована 2001-01-30», где значение `PublicationDate` оформлено как дата без времени.
3. Стандартной практикой в библиографии является игнорирование артиклей (к примеру, слова *The*) при алфавитной сортировке заглавий публикаций. Напишите запрос, который возвращает названия (`TitleName`) всех книг из таблицы `title`, отсортированные в алфавитном порядке с учетом данного правила.

15.5. Ответы

1. Можно задействовать функцию `CONCAT_WS` для форматирования имен, составив следующий запрос:

```
SELECT CONCAT_WS(', ', LastName, FirstName) AS AuthorName
FROM author;
```

2. Достичь этого результата можно несколькими способами, в зависимости от того, какой функцией вы предпочтете воспользоваться. С применением CAST ваш запрос может выглядеть так:

```
SELECT CONCAT(
    'Первая книга опубликована ', CAST(PublicationDate AS DATE),
    '.'
    ) AS FirstPublicationDate
FROM title
WHERE TitleID = 101;
```

   Тот же результат можно получить с помощью CONVERT:

```
SELECT CONCAT(
    'Первая книга опубликована ', CONVERT(PublicationDate, DATE),
     '.'
    ) AS FirstPublicationDate
FROM title
WHERE TitleID = 101;
```

   Также можно задействовать функцию DATE, рассмотренную в главе 14:

```
SELECT CONCAT(
    'Первая книга опубликована ', DATE(PublicationDate), '.'
    ) AS FirstPublicationDate
FROM title
WHERE TitleID = 101;
```

3. Это задание чуть сложнее предыдущих, поскольку требует применения функции REPLACE в предложении ORDER BY, что вы еще не делали. Такой прием позволит заменить артикль *The* в значении TitleName на пустую строку для сортировки. Не забудьте включить пробел в подстроку ('The '), чтобы обеспечить надлежащий порядок сортировки:

```
SELECT TitleName
FROM title
ORDER BY REPLACE(TitleName, 'The ', '');
```

16 Вставка данных

В предыдущих 15 главах мы изучали способы считывания данных при помощи команды `SELECT`. Однако все эти значения должны были сначала каким-то образом попасть в таблицы. В этой главе мы наконец узнаем, как добавлять данные.

В ходе освоения материала мы не раз убеждались в том, что синтаксис SQL во многом схож с английским языком. Это справедливо и для операции вставки строк данных, реализуемой через ключевое слово `INSERT` («вставить»). Далее мы рассмотрим ряд способов применения `INSERT` для заполнения таблиц нашей БД новыми данными.

16.1. Вставка конкретных значений

Начнем с самого прямого способа — вставки конкретных значений. Помните: вставляя данные, мы добавляем новую строку в уже существующую таблицу.

В главе 2 мы говорили о строках, столбцах и значениях. Все таблицы в нашей реляционной СУБД содержат строки данных, а каждая строка имеет набор свойств, определенный столбцами таблицы. Каждое свойство в столбцах представлено значением определенного типа данных, иногда с применением NULL для обозначения отсутствия значения в конкретном столбце.

Надеюсь, что к настоящему моменту предыдущий абзац полностью вам понятен, учитывая все SQL-запросы, что вы уже написали. Если же у вас остались какие-то сомнения, рекомендую перечитать главу 2, чтобы закрепить понимание строк, столбцов и значений. Если же все предельно ясно — вперед, к добавлению новых данных!

16.1.1. Вставка новой строки

Для вставки данных мы воспользуемся ключевым словом `INSERT`, а для указания конкретных значений — ключевым словом `VALUES`. В качестве первого запроса мы добавим новую строку в таблицу `title`, однако сначала давайте снова окинем взором все данные в ней.

В главе 3 мы подчеркнули нежелательность применения команды `SELECT *`, но поскольку таблица у нас небольшая, мы все же задействуем эту команду, чтобы не вводить все имена столбцов в рабочем запросе. Для просмотра строк данных в таблице мы неоднократно прибегнем в этой главе к «звездочке выбора» (рис. 16.1):

```
SELECT *
FROM title;
```

TitleID	TitleName	Price	Advance	Royalty	PublicationDate
101	Pride and Predicates	9.95	5000.00	15.00	2015-04-30 00:00:00
102	The Join Luck Club	9.95	6000.00	12.00	2016-02-06 00:00:00
103	Catcher in the Try	8.95	5000.00	10.00	2017-04-03 00:00:00
104	Anne of Fact Tables	12.95	10000.00	15.00	2018-01-12 00:00:00
105	The DateTime Machine	7.95	5500.00	15.00	2019-02-04 00:00:00
106	The Great GroupBy	10.95	0.00	20.00	2019-12-23 00:00:00
107	The Call of the While	8.95	2500.00	15.00	2020-03-14 00:00:00
108	The Sum Also Rises	7.95	5000.00	12.00	2021-11-12 00:00:00

Рис. 16.1. Выборка всех строк и столбцов в таблице title

Если мы собираемся вставить новую строку в таблицу, представленную на рис. 16.1, нам потребуются значения соответствующих типов данных для всех столбцов. Для удобства чтения укажем в запросе столбцы в порядке их следования в таблице: сначала `TitleID`, затем `TitleName` и т. д.

Ниже приведен запрос, который мы применим для вставки новой строки с данными о книге «David Emptyfield»[1]. Мы будем называть такой запрос командой `INSERT`:

```
INSERT INTO title (
    TitleID,
    TitleName,
    Price,
    Advance,
    Royalty,
    PublicationDate
```

[1] Здесь обыгрывается название романа Чарльза Диккенса «Жизнь Дэвида Копперфилда, рассказанная им самим» (*David Copperfield*). — *Примеч. пер.*

```
    )
VALUES (
    109,
    'David Emptyfield',
    9.95,
    0.00,
    10.00,
    '2022-01-16'
    );
```

В запросе мы указываем очередной порядковый идентификатор `TitleID` (109), название книги (`TitleName`), а также значения для столбцов `Price` (9.95), `Advance` (0.00), `Royalty` (10.00) и `PublicationDate` (2022-01-16). При успешном выполнении запроса в панели вывода отображается сообщение: 1 row(s) affected («Изменения коснулись 1 строки»).

К практике!

Составьте и выполните последний запрос. Возможно, ввод всех имен столбцов таблицы `title` покажется вам утомительным, но не забывайте, что можно воспользоваться клавишей Shift для выбора всех имен столбцов на панели навигации, а затем перетащить их на панель запросов. Забыли, как это сделать? Загляните в главу 3!

Еще одна деталь, на которую стоит обратить внимание в нашем запросе: и названия столбцов, и значения взяты в круглые скобки. При помощи этих скобок мы задаем порядок столбцов и присваиваемых им значений, которые применяются в нашей команде `INSERT`.

Примечательно, что тот же запрос можно сформировать с другим порядком столбцов. Главное, чтобы значения располагались в такой же последовательности, в какой указаны соответствующие им столбцы.

Вот пример с обратным порядком столбцов. Не запускайте его, если вы уже выполнили предыдущую команду `INSERT`:

```
INSERT INTO title (
    PublicationDate,
    Royalty,
    Advance,
    Price,
    TitleName,
    TitleID
    )
VALUES (
    '2022-01-16',
    10.00,
    0.00,
    9.95,
```

```
    'David Emptyfield',
    109
    );
```

К чему вообще менять порядок столбцов в предложении `INSERT`? Как правило, в этом нет никакой необходимости — так вы лишь запутаете тех, кто будет разбирать ваш SQL-код. Однако если таблица содержит десятки или даже сотни столбцов, ваш запрос может стать более удобочитаемым, если указать имена столбцов в алфавитном порядке, а не в порядке их расположения в таблице. В остальных случаях при написании команды `INSERT` лучше придерживаться порядка столбцов, заданного в структуре таблицы.

16.1.2. Вставка нескольких строк

Еще один любопытный аспект, связанный с заключением значений в круглые скобки: каждая пара скобок обозначает набор значений для одной строки. Это значит, что при необходимости можно одновременно добавить несколько строк. Подобно тому как запятая используется для разделения столбцов и значений внутри команды `INSERT`, ее можно задействовать для разделения наборов значений, которые вставляются как отдельные строки. Ниже приведен пример добавления еще двух книг[1] в таблицу `title` с применением нескольких наборов данных в разделе `VALUES` команды `INSERT`:

```
INSERT INTO title (
    TitleID,
    TitleName,
    Price,
    Advance,
    Royalty,
    PublicationDate
    )
VALUES (
    110,
    'Red Badge of Cursors',
    7.95,
    0.00,
    15.00,
    '2022-03-29'
    ),
    (
    111,
    'Of Mice and Metadata',
    8.95,
    0.00,
```

[1] В двух новых названиях вымышленных книг о SQL угадываются роман Стивена Крейна «Алый знак доблести» (*The Red Badge of Courage*) и повесть Джона Стейнбека «О мышах и людях» (*Of Mice and Men*). — *Примеч. пер.*

```
12.00,
'2022-05-17'
);
```

В этом примере указаны лишь две дополнительные строки. Тем не менее не существует никакого ограничения на количество строк, которые можно вставить в таблицу. Единственным реальным ограничением является объем дискового пространства базы данных, который сложно определить с помощью какого-либо универсального SQL-запроса (и что выходит за рамки этой книги). Впрочем, если вы не добавляете тысячи или миллионы строк, вам вряд ли стоит беспокоиться о месте на диске.

НА ЗАМЕТКУ Раз уж речь зашла об универсальных SQL-инструкциях, стоит отметить, что при работе с другими реляционными базами данных можно столкнуться с синтаксисом команды `INSERT`, в котором опускается ключевое слово `INTO`. То есть запрос выглядит примерно так: `INSERT [имя_таблицы] VALUES (...)`. Однако пропуск `INTO` допустим не во всех СУБД, и потому я рекомендую всегда включать его в код, что позволит избежать синтаксических ошибок при переносе SQL-кода между различными платформами.

До сих пор все команды `INSERT` предполагали вставку полной строки (или строк) данных. Однако при работе с SQL иногда может потребоваться добавить значения не для всех столбцов, хотя программные ограничения могут этому препятствовать. Давайте рассмотрим, как можно вставить неполную строку.

16.1.3. Вставка неполной строки

Термин *неполная строка* (partial row) может сбивать с толку, поскольку в главе 2 я указал на обязательность наличия значений для всех столбцов в каждой строке таблицы. В целом это утверждение остается верным, однако существуют два условия, позволяющие вставлять строки данных частично.

Условие первое — когда таблица включает столбцы, допускающие пустые значения. В качестве примера обратимся к таблице `author` (см. рис. 16.2):

```
SELECT *
FROM author;
```

Столбец `MiddleName` допускает NULL-значения, поскольку не у всех авторов есть второе имя. Хотя столбец этот доставил нам немало хлопот при работе с функциями и конкатенацией в главе 15, зато его можно пропускать при вставке строк.

Если требуется вставить строку с пустым значением для второго имени, можно просто не указывать этот столбец (`MiddleName`) в команде `INSERT`. Это приведет к автоматическому присвоению NULL-значения, например:

```
INSERT INTO author (
    AuthorID,
    FirstName,
    LastName,
    PaymentMethod
    )
VALUES (
    12,
    'Whitney',
    'Miller',
    'Cash'
    );
```

AuthorID	FirstName	MiddleName	LastName	PaymentMethod
1	Paul	K	Tripp	Cash
2	Doug	NULL	Li	Check
3	Jen	NULL	Strong	Check
4	Jorge	Armando	Guerra	Check
5	Robert	Grant	Davidson	Check
6	Gail	Anne	Shawn	Check
7	Rebecca	NULL	Miller	Check
8	Andy	NULL	Melkin	Direct Deposit
9	Buck	NULL	Fernandez	Cash
10	Chris	NULL	Walenski	Direct Deposit
11	Deepthi	NULL	Mahadevan	Direct Deposit

Рис. 16.2. Выборка всех строк и столбцов в таблице author

Запрос выполнится без ошибок — ведь столбец `MiddleName` допускает значение null и автоматически устанавливает его по умолчанию для новой строки. В этом легко убедиться, выполнив запрос и затем выбрав все строки в таблице `author` (рис. 16.3).

Отсутствие значения в этом запросе может показаться проявлением лености, но в действительности вы нередко столкнетесь с таблицами, в которых требуется вставлять строки, содержащие пустые значения для некоторых столбцов. Классический пример — таблица, включающая столбцы «Дата изменения» и «Пользователь, внесший изменения», где фиксируется, когда и кем вносились правки. При добавлении новых записей мы, напротив, хотим, чтобы эти поля сохраняли значения null, показывая тем самым, что данные не изменялись с момента их первоначального добавления в таблицу. Подробнее о модификации данных мы поговорим в главе 17.

AuthorID	FirstName	MiddleName	LastName	PaymentMethod
1	Paul	K	Tripp	Cash
2	Doug	NULL	Li	Check
3	Jen	NULL	Strong	Check
4	Jorge	Armando	Guerra	Check
5	Robert	Grant	Davidson	Check
6	Gail	Anne	Shawn	Check
7	Rebecca	NULL	Miller	Check
8	Andy	NULL	Melkin	Direct Deposit
9	Buck	NULL	Fernandez	Cash
10	Chris	NULL	Walenski	Direct Deposit
11	Deepthi	NULL	Mahadevan	Direct Deposit
12	Whitney	NULL	Miller	Cash

Рис. 16.3. Выборка всех строк и столбцов в таблице author, включая новую строку с AuthorID 12, в которой MiddleName присвоено NULL-значение

Возвращаясь к теме вставки неполных строк: второе условие, при котором можно не заполнять все столбцы, заключается в наличии *ограничений по умолчанию* (default constraints), заданных для одного или нескольких столбцов таблицы. Создатель таблицы может определить значение по умолчанию для столбца, чтобы при вставке данных это значение подставлялось автоматически. В результате данные для таких столбцов заполняются системой без их явного указания в команде `INSERT`.

Вернемся к таблице `author`. Обратите внимание, что первый столбец, `AuthorID`, содержит возрастающую числовую последовательность от 1 до 12. Эти значения должны быть строго уникальными для каждой строки, поскольку они выступают в качестве ключей, служащих для связи с данными в других таблицах (о ключах мы говорили в главе 8).

Чтобы значения эти не совпадали, разработчики баз данных обычно задают для первого столбца специальное ограничение по умолчанию, которое автоматически генерирует уникальный идентификатор для каждой строки. Это гарантирует, что мы с вами случайно не продублируем идентификационное значение для разных строк в таблице `author`. Если бы два автора получили один и тот же идентификатор, это привело бы к неоднозначности связей между таблицами, использующими столбец `AuthorID`.

Сейчас в нашей таблице `author` такого ограничения нет: мы сами вводили значение 12 при добавлении новой строки. Если бы для `AuthorID` применялся автоматически генерируемый числовой идентификатор, система не позволила бы

нам самостоятельно присваивать значение для этого столбца. В таких случаях, какие нередко встречаются в таблицах с ID-полями, `INSERT` следует записывать, опуская столбец `AuthorID`, например, так:

```
INSERT INTO author (
    FirstName,
    MiddleName,
    LastName,
    PaymentMethod
    )
VALUES (
    'Whitney',
    NULL,
    'Miller',
    'Cash'
    );
```

Отмечу еще раз: в нашей базе данных не установлено ограничение по умолчанию для таблицы author, поэтому не выполняйте этот запрос. Просто имейте в виду, что во многих реальных базах данных вам потребуется учитывать подобные столбцы при частичной вставке данных.

16.1.4. Предостережение относительно пропуска столбцов

При работе с реальным SQL-кодом других разработчиков встречается еще один вид пропуска в команде `INSERT`: пропуск названий столбцов таблицы в разделе `INSERT INTO`. К примеру, можно составить такую команду `INSERT` и успешно выполнить ее:

```
INSERT INTO author
VALUES (
    12,
    'Whitney',
    NULL,
    'Miller',
    'Cash'
    );
```

Такой подход может показаться заманчивым, однако если учесть ряд аспектов, уже обсуждавшихся ранее, можно догадаться, чем опасен этот вариант.

- *Указанный прием работает только в тех случаях, когда присутствуют значения для всех столбцов таблицы и все они указаны в надлежащем порядке.* Если одно значение лишнее или, напротив, одного не хватает, запрос завершится ошибкой. Если значения указаны в неверном порядке, в лучшем случае вы добавите строку с некорректными данными, а в худшем — столкнетесь с ошибкой. (Пожалуй, ошибка выполнения предпочтительнее — так вы хотя бы сохраните целостность данных.)

- *Запросы такого вида дают сбой при изменении базовой таблицы*. Расширение таблицы путем добавления столбцов — распространенная практика. Если вы добавите новый столбец в таблицу `author`, запрос перестанет корректно выполняться, поскольку количество столбцов превысит количество значений.
- *В таблице могут быть ограничения по умолчанию, не позволяющие вводить значения для определенных столбцов*. Будь в нашей таблице какие-либо ограничения по умолчанию, команда `INSERT` без указания столбцов не выполнилась бы.

По вышеизложенным причинам всегда следует прописывать имена столбцов в команде `INSERT`. Обнаружив SQL-код без указания столбцов, постарайтесь осуществить его рефакторинг, включив имена столбцов.

Разобравшись с этим вопросом, рассмотрим альтернативные способы вставки данных. До сих пор для добавления строк мы пользовались ключевым словом `VALUES`, однако есть и другие варианты. Отрадно то, что вам они уже знакомы.

16.2. Вставка строки посредством подзапроса

Когда мы применяем ключевое слово `VALUES`, как делали это до сих пор в командах `INSERT`, мы тем самым говорим системе: «Смотри, СУБД, вот набор значений, и я хочу вставить их в эту таблицу». Как на языке SQL, так и в обычном языке такое высказывание состоит из двух смысловых частей: указания места, куда производится вставка, и перечисления вставляемых значений.

Чтобы вместо `VALUES` задействовать другой метод в команде `INSERT`, достаточно просто заменить вторую часть выражения на подзапрос, который возвращает столбцы в таком же порядке и с теми же типами данных, что и столбцы таблицы, в которую осуществляется вставка, как это задано в первой части запроса.

Поскольку мы уже добавили новую книгу «David Emptyfield» и нового автора по имени Уитни Миллер (Whitney Miller), то можем связать эту книгу с конкретным автором через таблицу `titleauthor`. Я еще не рассказывал об этой таблице, поэтому давайте познакомимся с ней, выполнив рабочий запрос `SELECT *` (результаты показаны на рис. 16.4):

```
SELECT * FROM titleauthor;
```

В показанной таблице имеется три столбца: `TitleID`, `AuthorID` и `AuthorOrder`. Как вы, наверное, уже догадались, `TitleID` связан со столбцом `TitleID` в таблице `title`, а `AuthorID` связан со столбцом `AuthorID` в таблице `author`. Следует отметить, что значения в этих столбцах не обязательно являются уникальными. Таблица представляет отношение «многие ко многим», поскольку любая книга может иметь более одного автора, и любой автор мог участвовать в создании нескольких книг.

TitleID	AuthorID	AuthorOrder
101	2	1
102	3	1
103	4	1
104	5	1
105	6	1
106	7	1
107	11	1
107	1	2
108	8	1
108	9	2
108	10	3

Рис. 16.4. Выборка всех строк и столбцов в таблице titleauthor

Третий столбец, `AuthorOrder`, указывает порядок авторов на обложке произведения. Для одного автора значение в этом столбце всегда равно `1`, что важно для новой строки данных, которую мы собираемся добавить.

Начнем с команды `SELECT`, которая позволит нам запросить необходимые таблицы. Поскольку данные эти еще не связаны, допустимо использовать подзапросы в соответствии с методикой, рассмотренной нами в главе 11:

```
SELECT
    (
    SELECT TitleID
    FROM title
    WHERE TitleName = 'David Emptyfield'
    ) AS TitleID,
    (
    SELECT AuthorID
    FROM author
    WHERE FirstName = 'Whitney'
        AND LastName = 'Miller'
    ) AS AuthorID;
```

НА ЗАМЕТКУ Применение подзапросов в командах `SELECT`, как правило, не рекомендуется, однако в команде `INSERT` использование подзапроса вполне оправданно, поскольку это простой и эффективный способ добавления значений в таблицу `titleauthor`.

Этот запрос служит отправной точкой для нашей вставки. Обратите внимание, что, хотя псевдонимы столбцов не обязательны, они добавлены для удобства чтения и помогают нам сопоставить результирующие значения со столбцами таблицы `titleauthor`.

Добавим необходимые компоненты, чтобы превратить запрос в команду `INSERT`, включая добавление значения `1` для столбца `AuthorOrder`. Наш окончательный запрос будет выглядеть так:

```
INSERT INTO titleauthor (
    TitleID,
    AuthorID,
    AuthorOrder
    )
SELECT
    (
    SELECT TitleID
    FROM title
    WHERE TitleName = 'David Emptyfield'
    ) AS TitleID,
    (
    SELECT AuthorID
    FROM author
    WHERE FirstName = 'Whitney'
        AND LastName = 'Miller'
    ) AS AuthorID,
    1 AS AuthorOrder;
```

Вновь подчеркну: псевдонимы в предложении `SELECT` присвоены столбцам исключительно для удобства восприятия. При этом важно, что столбцы в обоих разделах запроса — `INSERT` и `SELECT` — расположены в соответствующем порядке.

К практике!

Если вы этого еще не сделали, выполните SQL-запрос для вставки произведения «David Emptyfield» в таблицу `title` (раздел 16.1.1) и автора Whitney Miller в таблицу author (раздел 16.1.3) с последующим выполнением команды `INSERT` для их связывания в таблице `titleauthor`. Эти данные пригодятся нам в главе 17.

16.3. Вставка строки при помощи переменных

Теперь обратимся еще к одному способу вставки данных — с применением переменных. В главе 13 были представлены приемы работы с переменными в командах `SELECT`, значение которых при создании многократно используемого SQL-кода трудно переоценить. Допустимо объявлять и задействовать переменные в запросе `INSERT...SELECT` для формирования легко модифицируемых запросов, требующих лишь изменения значений переменных. Воспользуемся следующим примером, чтобы добавить в таблицу `title` книгу «A Table of Two Cities»[1]:

```
SET
    @TitleID = 112,
    @TitleName = 'A Table of Two Cities',
    @Price = 9.95,
```

[1] Здесь обыгрывается название исторического романа Чарльза Диккенса «Повесть о двух городах» (*A Tale of Two Cities*). — *Примеч. пер.*

```
    @Advance = 0.00,
    @Royalty = 15.00,
    @PublicationDate = '2022-08-07';

INSERT INTO title (
    TitleID,
    TitleName,
    Price,
    Advance,
    Royalty,
    PublicationDate
    )
VALUES (
    @TitleID,
    @TitleName,
    @Price,
    @Advance,
    @Royalty,
    @PublicationDate
    );
```

Рассмотренный пример иллюстрирует лишь один из возможных подходов к использованию переменных при добавлении данных. Если вы прочли главу 13, уверен, вам наверняка пришли на ум и другие способы применения переменных для вставки данных в таблицы БД.

Освоив добавление данных при помощи команды `INSERT`, мы сделали первый шаг в освоении SQL-команд, предназначенных для так называемого манипулирования данными. Под *манипулированием данными* (data manipulation) понимается набор операций по модификации содержимого таблиц, включающий не только его добавление, но также изменение и удаление. В главе 17 вы обогатите навыки манипулирования данными и узнаете, как изменять и удалять хранящиеся в базе данных значения.

16.4. Практическое занятие

1. Выполнится ли этот SQL-запрос успешно, несмотря на то что псевдонимы столбцов отличаются от имен столбцов таблицы?

```
INSERT INTO titleauthor (
    TitleID,
    AuthorID,
    AuthorOrder
    )
SELECT
    (
    SELECT TitleID
    FROM title
```

```
WHERE TitleName = 'David Emptyfield'
) AS TID,
(
SELECT AuthorID
FROM author
WHERE FirstName = 'Whitney'
    AND LastName = 'Miller'
) AS AID,
1 AS AO;
```

2. Будет ли данный SQL-запрос выполнен успешно при отсутствии явного указания имен столбцов таблицы?

```
INSERT INTO author
VALUES (
    13,
    'Jeff',
    'Iannucci'
    );
```

3. Выполните вставку новой строки в таблицу `promotions`, добавив следующие значения:
   - `PromotionID` — 13
   - `PromotionCode` — 2OFF2022
   - `PromotionStartDate` — 1 мая 2022 г.
   - `PromotionEndDate` — 15 мая 2022 г.
4. Добавьте новую строку в таблицу `customer` для покупателя по имени Джанлука Росси (Gianluca Rossi). Идентификатор `CustomerID` должен быть равен 21, тогда как вся остальная информация — быть такой же, как у Миа Росси (Mia Rossi) с CustomerID 20. В этом упражнении используйте для вставки `SELECT` вместо `VALUES`.

16.5. Ответы

1. Да. Имена столбцов, даже если они указаны в виде псевдонимов в предложении `SELECT`, не обязательно должны совпадать с именами столбцов в таблице, в которую вставляются данные. Важно, чтобы совпадало число столбцов и соответствующие им типы данных.
2. Нет. Выполнение запроса приведет к ошибке, указывающей на несоответствие количества столбцов, поскольку таблица `author` содержит четыре столбца, а запрос пытается вставить значения только для трех столбцов. Этот

случай демонстрирует одну из причин, по которой всегда следует указывать столбцы таблицы при добавлении данных.

3. Ваш SQL-код должен быть примерно таким:

```
INSERT INTO promotion (
    PromotionID,
    PromotionCode,
    PromotionStartDate,
    PromotionEndDate
    )
VALUES (
    13,
    '2OFF2022',
    '2022-05-01 00:00:00',
    '2022-05-15 00:00:00'
    );
```

4. Несмотря на возможность явного указания всех значений для новой строки с помощью литеральных значений, можно также скопировать существующие значения из строки с `CustomerID=20`, за исключением значений `CustomerID` и `FirstName`, запустив следующий программный код:

```
INSERT INTO customer (
    CustomerID,
    FirstName,
    LastName,
    Address,
    City,
    State,
    Zip,
    Country
    )
SELECT
    21,
    'Gianluca',
    LastName,
    Address,
    City,
    State,
    Zip,
    Country
FROM customer
WHERE CustomerID = 20;
```

17 Обновление и удаление данных

В главе 16 мы рассмотрели вставку новых строк данных в таблицы и впервые попрактиковались в применении SQL для операций, отличных от чтения данных. Там же отмечалось, что `INSERT` является одним из нескольких ключевых слов, предназначенных для манипулирования данными.

В этой главе мы изучим два других способа манипулирования данными: обновление и удаление. В связи с тем, что SQL проектировался как язык, интуитивно понятный англоговорящим пользователям, для выполнения вышеуказанных операций служат ключевые слова `UPDATE` («обновить») и `DELETE` («удалить») соответственно.

17.1. Обновление значений

Обновление данных несколько отличается от их вставки — в этом случае мы выполняем операции не на уровне строк, а на уровне столбцов. Как вы помните, таблицы состоят из строк, а строки представляют собой набор свойств, представленных отдельными столбцами. Когда мы обновляем данные в SQL, мы работаем со значениями этих свойств, то есть вносим изменения именно на уровне столбцов.

Такие изменения могут затрагивать или не затрагивать все столбцы для конкретной строки, а также включать или не включать обновление значений одного или нескольких столбцов для всех строк таблицы. В SQL предусмотрено множество способов обновления данных. Если мое объяснение кажется пока слишком абстрактным, не переживайте — в примерах, приведенных ниже, мы подробно разберем все существующие возможности.

17.1.1. Выполняем операции с данными в реальном времени

Прежде чем двигаться дальше, хочу обратить ваше внимание на одну особенность SQL и реляционных баз данных, которая нередко застает пользователей врасплох. Возможно, вы привыкли работать с текстовым редактором, электронной таблицей или другим приложением, где данные сохраняются лишь после того, как вы нажмете соответствующую кнопку или сочетание клавиш. И если вы допустили ошибку, то в таких программах любые правки легко отменить, вернувшись на шаг назад.

В реляционных базах данных все иначе: любая операция с данными выполняется в реальном времени и является необратимой. Отменить досадный промах невозможно — изменения фиксируются мгновенно. Да, звучит настораживающе, однако именно благодаря такому подходу системы управления реляционными базами данных способны оперативно и точно предоставлять актуальные данные сотням или тысячам пользователей, одновременно подключенным к базе.

> **ВНИМАНИЕ** Помните, что все операции с данными, которые мы будем производить, выполняются в реальном времени. В реляционных СУБД нет кнопки или сочетания клавиш «Сохранить».

Поскольку изменения в базе данных происходят мгновенно, а люди щедро наделены талантом ошибаться, MySQL Workbench оснащен встроенной функцией безопасности под названием Safe Updates (Безопасные обновления). Этот режим включен по умолчанию и снижает риск случайного изменения данных, которое могло бы затронуть все строки таблицы.

В частности, при включенном режиме Safe Updates невозможно выполнить обновление без указания ключевого поля — своего рода идентификатора нужной строки. Так как в наших таблицах в настоящий момент нет ключевых полей, нам необходимо отключить этот режим, чтобы иметь возможность обновлять содержимое БД. В противном случае любые SQL-инструкции из этой главы будут завершаться ошибкой.

Отключить режим можно разными способами, но самый надежный — через меню Edit ▸ Preferences в левом верхнем углу окна MySQL Workbench. Эта команда открывает диалоговое окно настроек Workbench Preferences (см. рис. 17.1). В панели слева выберите SQL Editor, затем снимите флажок с параметра Safe Updates в нижней части основной области окна. После этого нажмите OK для закрытия окна настроек, а затем перезапустите приложение Workbench. Теперь функция Safe Updates отключена, можно приступать к обновлению данных — и делать это следует предельно осторожно.

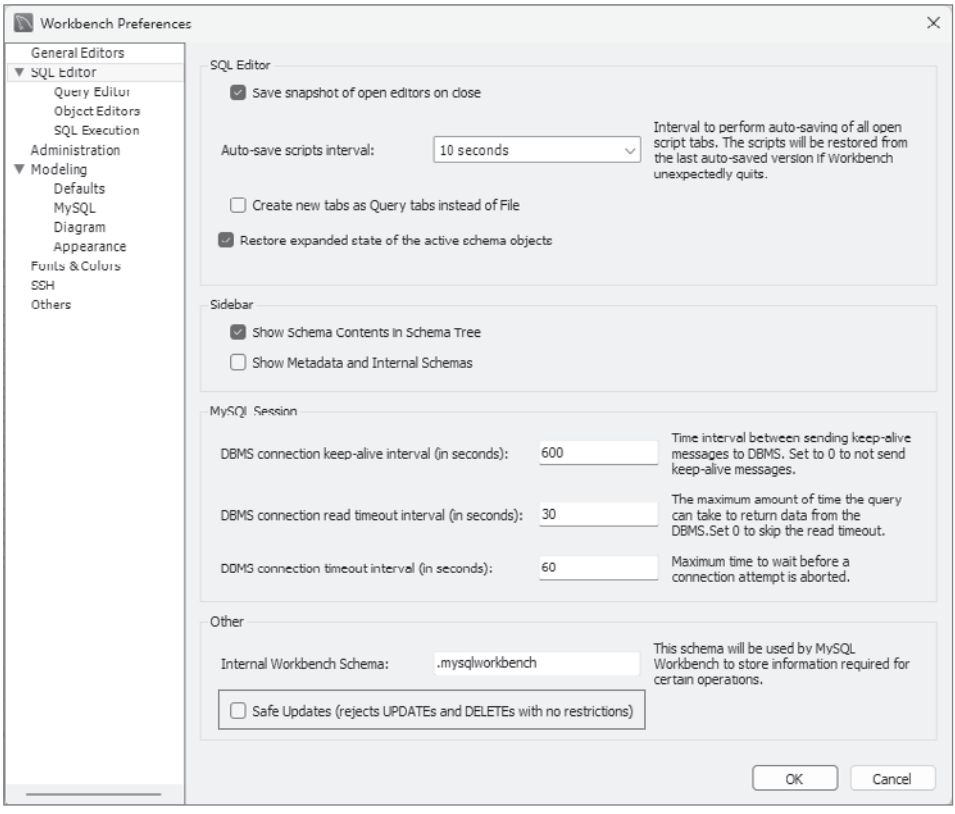


Рис. 17.1. Отключенный режим Safe Updates в диалоговом окне настроек Workbench Preferences

17.1.2. Обязательные компоненты операции обновления

В первом примере нам предстоит обновить ценник одной книги, «Pride and Predicates», с 9.95 до 8.95 доллара. Как и в случае со многими SQL-инструкциями, начнем с формулировки простого высказывания на английском языке: “I would like to update the price of Pride and Predicates to $8.95” («Мне нужно обновить цену Pride and Predicates до 8.95 доллара»).

Эта фраза служит неплохой отправной точкой, семантически близкой будущей SQL-команде, однако нам необходимо внести ряд изменений, указав имена таблиц и столбцов: “I would like to update the price to $8.95 where the title name is Pride and Predicates” («Мне нужно обновить цену до 8.95 доллара для книги, название которой — Pride and Predicates»).

Наша словесная формулировка приблизилась к заветному SQL-коду. У нас есть фильтр («для книги, название которой — Pride and Predicates»), но нам все еще

нужно указать целевую таблицу. Мы делаем это, уточнив, что хотим обновить таблицу, а затем присваиваем новые значения полям одного или нескольких столбцов: “I would like to update the title table. I would like to set the price to $8.95 where the title name is Pride and Predicates” («Хочу обновить таблицу title. Мне нужно установить цену в 8.95 доллара для книги, название которой — Pride and Predicates»).

Теперь наша словесная формулировка идеальна. Преобразуем ее в SQL-инструкцию:

```
UPDATE title
SET Price = 8.95
WHERE TitleName = 'Pride and Predicates';
```

В запросе фигурирует новое ключевое слово `UPDATE`, а вместе с ним и старый знакомый по переменным — `SET`. Команда `UPDATE` указывает таблицу для модификации данных, а `SET` задает новые значения — почти как при работе с переменными. Данный `UPDATE`-запрос иллюстрирует три обязательных компонента любого обновления и их последовательность для формирования корректного SQL-запроса:

1. Таблица, подлежащая обновлению (в предложении `UPDATE`).
2. Имя столбца (или столбцов) для присвоения новых значений (в предложении `SET`).
3. Условие фильтрации строк, подлежащих обновлению (в предложении `WHERE`).

> **ВНИМАНИЕ** Пусть с точки зрения синтаксиса это и не обязательно, но предложение `WHERE` — это, без преувеличения, краеугольный камень всей операции. Если при *выполнении* запроса его *проигнорировать*, волна обновлений прокатится по *каждой* строке таблицы без разбора. Учитывая, что изменения вносятся напрямую и в реальном времени, следует с предельной осмотрительностью составлять условие фильтрации, чтобы оно изменяло значения исключительно в нужных строках.

17.1.3. Обновляем значения в одном или нескольких столбцах

Как указано во втором обязательном компоненте SQL-инструкций обновления данных, значения можно изменять в одном или нескольких столбцах. Аналогично синтаксису команды `SELECT`, где запятые позволяют перечислять несколько столбцов, в `UPDATE` запятые применяются для обозначения двух или более целевых правок.

> **НА ЗАМЕТКУ** Все значения, подлежащие изменению в команде `UPDATE`, должны принадлежать одной и той же таблице и удовлетворять условию фильтра.

В следующем примере значения аванса (`Advance`) и авторского гонорара (`Royalty`) для книги «Pride and Predicates» изменяются на 0 и 10 соответственно:

```
UPDATE title
SET Advance = 0.00,
    Royalty = 10.00
WHERE TitleName = 'Pride and Predicates';
```

Несмотря на то что мы фильтруем строки по названию (`TitleName`), для `UPDATE`-запросов более типично применение ключевых полей или иных уникальных идентификаторов, что гарантирует обновление требуемой строки или строк. Отредактируем наш код, чтобы задействовать идентификатор `TitleID`, который для книги «Pride and Predicates» равен 101:

```
UPDATE title
SET Advance = 0.00,
    Royalty = 10.00
WHERE TitleID = 101;
```

> **НА ЗАМЕТКУ** Если вы выполнили две предыдущие команды, то заметили, что обе они завершились успешно, но выдали разные сообщения на панели вывода. Так, для первой команды отображается `(1) row(s) affected` («Затронута (1) строка») и `Changed: 1` («Изменена: 1»), а для второй — `(0) row(s) affected` («Затронуто (0) строк») и `Changed: 0` («Изменено: 0»). Все просто: второй `UPDATE`-команде уже не над чем было работать, ведь первое же обновление установило для Advance и Royalty именно те значения, что мы пытались задать повторно.

Как и в случае с `INSERT`-командами из главы 16, мы можем задействовать переменные, чтобы написать код, пригодный для многократного применения. В следующем примере мы восстановим исходные значения для книги «Pride and Predicates», отбирая строку по ее идентификатору (`TitleID`):

```
SET
    @TitleID = 101,
    @Price = 9.95,
    @Advance = 5000.00,
    @Royalty = 15.00;

UPDATE title
SET Price = @Price,
    Advance = @Advance,
    Royalty = @Royalty
WHERE TitleID = @TitleID;
```

К практике!

Если вы до сих пор этого не сделали, самое время выполнить все встретившиеся в главе команды `UPDATE`. Чтобы отслеживать изменения, используйте запрос вида `SELECT * FROM title WHERE TitleID = 101;` между выполнением инструкций.

Операция обновления позволяет совершить еще одно действие — удалить значение. Напомню, что в СУБД каждый столбец должен иметь какое-либо представление значения в каждой строке таблицы, поэтому единственный способ удалить значение — установить его в NULL, что укажет на отсутствие данных.

Далеко не все столбцы допускают нулевые значения, и тем не менее нам известен как минимум один такой столбец в нашей БД: второе имя (`MiddleName`) в таблице `author`. Чтобы установить `MiddleName` для первого автора в NULL, можно составить `UPDATE`-запрос следующего вида:

```
UPDATE author
SET MiddleName = NULL
WHERE AuthorID = 1;
```

Таковы возможности модификации данных для обновления значений с помощью базовой команды `UPDATE`. Далее рассмотрим, как применять `UPDATE` для запроса нескольких таблиц в составе условия фильтрации.

17.1.4. Обновляем значения при помощи многотабличного запроса

В главе 8 мы узнали, что *предикат* — это любая часть SQL-команды, которая проверяет, является ли некое условие истинным, ложным или неопределенным. До этого момента предикаты применялись преимущественно для фильтрации данных — в условиях, заданных в предложениях `FROM`, `WHERE` и `HAVING`.

В процессе работы с SQL вам, вероятно, придется писать `UPDATE`-команды, в которых фильтрация выполняется при помощи предиката, обращающегося к двум и более таблицам. Тут начинается самое интересное: оператор `UPDATE` позволяет изменять значения только в одной таблице, а фильтровать мы планируем по данным из нескольких. Поэтому важно правильно построить запрос. Так, для задания условий фильтра потребуется оператор `FROM`, который следует разместить в надлежащем разделе SQL-команды.

К сожалению, в разных системах управления реляционными базами данных размещение предиката в `UPDATE` может отличаться. Хотя использование большинства ключевых слов в SQL стандартизировано, единого стандарта для обновления данных с участием нескольких таблиц не существует.

> **ВНИМАНИЕ** Несмотря на то что в этом разделе приведены примеры работы с другими популярными СУБД, отличными от MySQL, они не являются исчерпывающими. Обязательно сверьтесь с документацией вашей конкретной СУБД — там вы найдете точный синтаксис для такого рода `UPDATE`-команд.

Допустим, нам нужно изменить стоимость всех книг определенного автора в базе данных `sqlnovel` в MySQL. Нам известен идентификатор автора (`AuthorID`),

который равен 12, но мы не знаем названий (TitleName) или идентификаторов (TitleID) конкретных книг. Наша цель — установить новую цену в $8.95. Для выполнения такой операции потребуется связать между собой как минимум две таблицы.

Напомню, что таблицы title и author связаны через таблицу titleauthor. Уникальная идентификация строк в таблице title осуществляется посредством TitleID, а в таблице author — при помощи AuthorID. Таблица titleauthor содержит оба столбца, TitleID и AuthorID, реализуя отношение «многие ко многим», поскольку один автор может написать несколько книг и одна книга может быть написана несколькими авторами.

Прежде чем приступить к построению UPDATE-команды, составим запрос SELECT, чтобы вывести значение, которое мы собираемся обновить. Для написания SQL-кода, отображающего цену (Price) всех книг автора с AuthorID, равным 12, можно сформировать следующий запрос с соединениями в предложении FROM:

```
SELECT title.Price
FROM title
INNER JOIN titleauthor
    ON title.TitleID = titleauthor.TitleID
INNER JOIN author
    ON titleauthor.AuthorID = author.AuthorID
WHERE author.AuthorID = 12;
```

Запрос возвращает значение, которое мы собираемся изменить. Можно сделать его более наглядным и лаконичным, задействовав псевдонимы таблиц, как обсуждалось в главе 8:

```
SELECT t.Price
FROM title t
INNER JOIN titleauthor ta
    ON t.TitleID = ta.TitleID
INNER JOIN author a
    ON ta.AuthorID = a.AuthorID
WHERE a.AuthorID = 12;
```

Вот так — стало гораздо понятнее. Вдумчивые читатели наверняка заметят, что таблица author здесь вообще не нужна — ведь идентификатор автора (AuthorID) также присутствует в связующей таблице titleauthor. Следовательно, можно упростить запрос, убрав все упоминания таблицы author и фильтруя по AuthorID непосредственно в таблице titleauthor:

```
SELECT t.Price
FROM title t
INNER JOIN titleauthor ta
    ON t.TitleID = ta.TitleID
WHERE ta.AuthorID = 12;
```

Теперь, имея в качестве основы сформированную команду SELECT, мы можем без особого труда преобразовать ее в команду UPDATE, удалив предложение SELECT и заменив ключевое слово FROM на UPDATE. Затем, после UPDATE, мы задействуем SET для установки значений перед предложением WHERE. Поскольку мы задали псевдонимы таблиц в предыдущем предложении FROM, то можем использовать те же псевдонимы в нашем запросе, чтобы указать, какая именно таблица в запросе подвергается обновлению данных. Вот как будет выглядеть наша UPDATE-команда:

```
UPDATE title t
INNER JOIN titleauthor ta
    ON t.TitleID = ta.TitleID
SET t.Price = 8.95
WHERE ta.AuthorID = 12;
```

Этот пример может отчасти сбить с толку, поскольку мы разделили предикат на две части нашего запроса: соединение таблиц указано в операции обновления до SET, а дополнительные условия фильтрации — после SET в предложении WHERE. Еще раз подчеркну: такой подход к обновлению данных — особенность синтаксиса MySQL.

> **НА ЗАМЕТКУ** Далее представлены примеры не для MySQL, а для других популярных СУБД. В этих фрагментах SQL-кода наглядно демонстрируются синтаксические различия, однако запускать их в MySQL не стоит — при попытке их выполнения система просто вернет сообщения об ошибках.

Если бы мы работали с нашей базой данных в PostgreSQL, то по-прежнему бы указывали таблицу для обновления и ее псевдоним в разделе UPDATE нашего запроса, а вот соединение с дополнительными таблицами поместили бы в предложение FROM. Приведенный ниже запрос корректен для PostgreSQL, но несовместим с MySQL:

```
UPDATE title t
SET t.Price = 8.95
FROM titleauthor ta
WHERE t.TitleID = ta.TitleID
    AND ta.AuthorID = 12;
```

Применительно к инстансу SQL Server преобразование нашей инструкции выглядело бы даже логичнее, если брать за основу запрос SELECT. В SQL Server мы бы просто заменили SELECT на разделы UPDATE и SET, оставив предложения FROM и WHERE без изменений. Следующий запрос должным образом выполняется в SQL Server, но не поддерживается в MySQL:

```
UPDATE title t
SET t.Price = 8.95
```

```
FROM title t
INNER JOIN titleauthor ta
    ON t.TitleID = ta.TitleID
WHERE ta.AuthorID = 12;
```

Вывод прост: несмотря на возможность задействовать соединения для фильтрации в команде UPDATE, у каждой СУБД свой синтаксис для подобных запросов. Увы, здесь мы сталкиваемся с одним из тех немногих аспектов освоения SQL, который выходит за рамки этой книги — вам придется углубиться в документацию по конкретной СУБД, с которой вы работаете.

17.2. Удаление строк

Операция удаления в SQL — это своего рода антипод команды INSERT. Если INSERT добавляет одну или несколько строк в таблицу, то DELETE, напротив, безвозвратно удаляет их из таблицы. Важно помнить, что, как и в случае с INSERT и UPDATE, в одной инструкции DELETE можно изменить данные лишь из одной таблицы.

17.2.1. Удаляем одну или несколько строк

По уже привычной нам методике начнем со словесной формулировки запроса. Допустим, нам требуется удалить строку из таблицы title с идентификатором TitleID, равным 110. Постановка задачи простым языком могла бы звучать так: “I would like to delete any rows from the title table where the TitleID is 110” («Мне нужно удалить все строки из таблицы title, в которых TitleID равен 110»). Наша SQL-инструкция, весьма схожая с этой фразой, задействует новое ключевое слово DELETE:

```
DELETE
FROM title
WHERE TitleID = 110;
```

Как и в случае с INSERT и UPDATE, мы начинаем нашу инструкцию с ключевого слова для манипуляции данными — DELETE. Далее мы указываем таблицу, из которой намерены удалить данные. И в завершение прописываем условие отбора, чтобы удалить только целевые строки.

> **ВНИМАНИЕ** Как и в командах UPDATE, предложение WHERE не является здесь обязательным, но, пожалуй, это краеугольный камень всей операции. Если при *выполнении* запроса *проигнорировать* условие фильтрации, вы удалите *все* строки в таблице. Нетрудно представить, к каким катастрофическим последствиям это приведет. Поскольку все изменения вносятся напрямую и в реальном времени, следует с предельной осмотрительностью составлять условие фильтрации, чтобы оно применялось лишь к тем строкам, которые требуется удалить.

Разумеется, мы и здесь можем задействовать переменные для создания SQL-инструкций многократного использования в запросах. Для последнего примера это будет выглядеть так:

```
SET @TitleID = 110;

DELETE
FROM title
WHERE TitleID = @TitleID;
```

> **СОВЕТ** Некоторые СУБД разрешают не указывать ключевое слово `FROM` в подобных `DELETE`-инструкциях, и тем не менее лучше его использовать. Спору нет, всегда похвально стремиться к лаконичности кода, однако выигрыш от пропуска одного слова минимален. Более того, если впоследствии вы решите перенести такой запрос в другую СУБД, отсутствие слова `FROM` может сделать его неработоспособным.

17.2.2. Удаляем строки при помощи многотабличного запроса

Напомню, что синтаксис команды `UPDATE`, объединяющей несколько таблиц в предикате, зависит от управляющей СУБД. К сожалению, с командами `DELETE` ситуация аналогична. Хорошая новость заключается в том, что различия не столь существенны, поскольку MySQL, SQL Server и MariaDB придерживаются одного и того же синтаксиса.

И что еще приятнее: синтаксис этот удивительно напоминает команду `SELECT`. В разделе 17.1.4 мы подготовили такой `SELECT`-запрос, перед тем как составить `UPDATE`-команду для книги, связанной с идентификатором автора (`AuthorID`), равным 12:

```
SELECT t.Price
FROM title t
INNER JOIN titleauthor ta
  ON t.TitleID = ta.TitleID
WHERE ta.AuthorID = 12;
```

Если требуется не обновлять цену, а удалить запись из таблицы `title`, достаточно заменить предложение `SELECT` на `DELETE` с указанием таблицы, из которой нужно удалить строки, как показано ниже:

```
DELETE t
FROM title t
INNER JOIN titleauthor ta
  ON t.TitleID = ta.TitleID
WHERE ta.AuthorID = 12;
```

Следует еще раз подчеркнуть, что приведенный синтаксис корректно работает в MySQL и ряде других систем, но не является универсальным. Чтобы не столкнуться с ошибками, сверьтесь с документацией вашей СУБД — там вы найдете надлежащий способ удаления строк с использованием предиката, объединяющего несколько таблиц.

17.2.3. Удаляем все строки таблицы

Я уже предупреждал в разделе 17.2.1, что, если выполнить команду `DELETE` без предложения `WHERE`, она может удалить все строки таблицы. Я говорю «может», поскольку, как мы помним из главы 16, базы данных часто проектируются со специальными ограничениями, которые среди прочего либо разрешают, либо запрещают вставку определенных значений в таблицу. Эти ограничения могут также разрешать или запрещать удаление строк.

Если таких ограничений нет — а в наших таблицах БД их сейчас нет, — можно удалить все строки в таблице, выполнив команду `DELETE` без условия фильтра. Разумеется, удалять все строки в таблицах нашей базы данных мы не будем, но в учебных целях можно выполнить эту операцию для таблицы `myfirstquery`, с которой мы работали в главе 2, — это не повлияет на дальнейшие упражнения. Ниже приведен пример такого запроса:

```
DELETE
FROM myfirstquery;
```

Как и предполагалось, выполнение этого запроса приводит к удалению всех строк в таблице `myfirstquery`. Впрочем, для полной очистки таблицы существует куда более эффективное решение, поддерживаемое практически всеми СУБД, — команда `TRUNCATE TABLE`.

В отличие от `DELETE`, который кропотливо перебирает и стирает каждую строку, `TRUNCATE TABLE` действует молниеносно, не анализируя записи по отдельности. Эта команда создавалась именно для быстрой очистки.

Синтаксис `TRUNCATE TABLE` предельно прост. Наш SQL-запрос для очистки таблицы `myfirstquery` будет выглядеть так:

```
TRUNCATE TABLE myfirstquery;
```

`TRUNCATE TABLE` обладает куда меньшей гибкостью по сравнению с `DELETE`, поскольку она предназначена исключительно для быстрого удаления всех строк таблицы. Ее нельзя применить для удаления одной или нескольких строк. Кроме того, ограничения целостности данных, способные препятствовать выполнению `DELETE`, также могут блокировать и выполнение `TRUNCATE TABLE`.

К практике!

Воспользуйтесь приведенными выше командами DELETE и TRUNCATE TABLE для удаления всех строк из таблицы myfirstquery. Для тестирования процедуры удаления нескольких строк попробуйте выполнить одну или несколько команд INSERT для добавления строк в таблицу myfirstquery.

17.3. Один важный совет по манипулированию данными

В разделе 17.1.4 мы сформировали команду SELECT для идентификации строк, подлежащих обновлению. Хотя моей целью было продемонстрировать структурное сходство между командами SELECT и UPDATE, на практике подобным запросом рекомендуется пользоваться для того, чтобы заранее увидеть, какие именно строки попадут под изменение.

В этой главе я предупреждал уже дважды, как легко случайно изменить или удалить из таблицы все данные. Я мог бы добавить предостережение о риске изменения неверно отобранных данных из-за ошибки в условиях фильтрации команд UPDATE и DELETE. Напомню еще раз: все изменения в базе данных происходят в реальном времени, поэтому при обновлении или удалении данных следует соблюдать повышенную осторожность.

Самый надежный и широко распространенный способ избежать неприятностей — сначала составить и выполнить команду SELECT с той же логикой, что и в будущем коде UPDATE или DELETE. Такой подход позволяет заранее определить, какие строки будут затронуты предстоящей операцией, и тем самым предотвратить ошибочные изменения данных. Из собственного опыта могу сказать, что этот прием: сначала выбирать данные запросом, а уже потом вносить изменения — не раз спасал меня и многих других от катастрофических последствий.

Отменить уже выполненный запрос невозможно, зато вы всегда можете заранее просмотреть затронутые строки при помощи SELECT. Настоятельно рекомендую пользоваться отбором данных для проверки условий фильтрации вне зависимости от степени вашей квалификации в области SQL.

17.4. Практическое занятие

1. На основании материала, изученного в главе 16, составьте и выполните запрос на добавление вашего имени и адреса в таблицу customer. Для CustomerID укажите число 22. (Совет: в MySQL Workbench названия столбцов можно просто перетащить из панели навигатора в редактор запросов.)

2. Составьте и выполните запрос, чтобы поменять адрес в только что добавленной вами строке на адрес, по которому вы проживали ранее (или любой другой).
3. Составьте и выполните запрос на удаление строки, которую вы добавили и изменили в таблице `customer`, но сначала с помощью `SELECT` убедитесь, что ваш SQL-код даст корректные результаты.

17.5. Ответы

1. Приблизительно так может выглядеть ваш запрос на вставку строки в таблицу `customer`:

```
INSERT customer (
    CustomerID,
    FirstName,
    LastName,
    Address,
    City,
    State,
    Zip,
    Country
)
VALUES (
    22,
    'Jeff',
    'Iannucci',
    '1600 Pennsylvania Ave NW',
    'Washington',
    'DC',
    '20500',
    'USA'
    );
```

2. Приблизительно так может выглядеть ваш запрос на изменение адреса в таблице `customer`:

```
UPDATE customer
SET Address = '1700 W Washington St',
    City = 'Phoenix',
    State = 'AZ',
    Zip = '85007'
WHERE CustomerID = 22;
```

3. Приблизительно так может выглядеть ваш запрос на удаление строки из таблицы `customer`:

```
DELETE
FROM customer
WHERE CustomerID = 22;
```

18 Организация и хранение данных в таблицах

В главах 16 и 17 мы начали создавать данные и выполнять с ними различные операции, а в этой перейдем к рассмотрению способов создания и манипулирования уже самими таблицами. По существу, это и есть самое сердце SQL, ведь способ хранения информации в таблицах — основа любой реляционной БД. От того, как именно организованы данные в таблицах, зависит буквально все. Можно сказать, что правильный выбор структуры таблиц — *наиважнейшее* решение, которое принимается на этапе проектирования базы данных.

Впрочем, бояться тут нечего — глава не изобилует технически сложными подробностями. Материал по-прежнему остается простым и доступным, а поскольку вы уже работали с SQL-запросами, синтаксис которых во многом является отражением естественного английского языка, я уверен, что вы легко усвоите изложенные здесь концепции и команды. Кроме того, глава поможет освежить и закрепить ваши знания о таких понятиях, как первичные ключи и типы данных.

18.1. Создание таблицы

Создание таблицы в SQL — задача на удивление простая, в чем вы скоро убедитесь сами. Однако прежде чем переходить к написанию SQL-кода, стоит уделить внимание ряду ключевых аспектов, связанных с будущей таблицей.

18.1.1. Что следует учесть

Прежде чем приступить к проектированию таблицы, задайте себе три ключевых вопроса:

- Каким будет имя вашей таблицы?
- Какими будут имена столбцов, включенных в вашу таблицу?
- Данные какого типа будут храниться в этих столбцах?

К выбору имен и типов данных следует подходить вдумчиво, особенно принимая во внимание уже существующие таблицы. Названия должны быть информативными, однако нельзя использовать имя, если оно уже занято другой таблицей.

А вот имена столбцов могут повторяться — мы уже наблюдали это в базе данных `sqlnovel`: `OrderID`, `TitleID` и другие встречаются в разных таблицах. Впрочем, такой подход целесообразен лишь при наличии логической связи между одноименными столбцами.

> **СОВЕТ** Использование схожих имен — одна из причин, по которой я применил такие имена, как `OrderID` и `TitleID`, в базе данных `sqlnovel`. Возможно, когда-нибудь вы будете работать с базой данных, где в десятках таблиц встречаются столбцы с безликими именами вроде `ID` или `Name` — в них легко запутаться. При создании таблиц старайтесь делать имена столбцов информативными, однозначными, интуитивно понятными и легкими для восприятия всеми потенциальными пользователями ваших данных.

В качестве упражнения создадим таблицу для категорий книг. Конечно, мы могли бы обойтись без создания новой таблицы и просто добавить в таблицу `title` столбец `Category` («Категория») со строковыми значениями вроде `Mystery` («Детектив») или `Romance` («Романтика»). Однако если вспомнить наш разговор о связях между таблицами из главы 8, станет ясно: такой подход не является оптимальным для реляционных БД. (Позже в этой главе мы все же добавим столбец в таблицу `title`, вот только его значения будут ссылаться на первичный ключ новой таблицы.)

Для новой таблицы категорий нам хватит двух столбцов: идентификатора (ID), который станет первичным ключом, и поля для названия категории. Чтобы сохранить единообразие структуры всей базы данных, назовем таблицу `category`, а ее столбцы — `CategoryID` и `CategoryName`.

Столбец `ID` в таблице обычно определяется как целочисленный (integer) тип данных — `int`. Тип данных `int` позволяет применять уникальный числовой идентификатор для каждой строки, обычно в диапазоне до нескольких миллиардов. Вряд ли эта таблица будет содержать миллиарды категорий, поэтому для поля `CategoryID` тип данных `int` вполне подойдет. Существуют и другие целочисленные типы данных — такие, как `tinyint`, `smallint`, `mediumint` и `bigint`, — тем не менее, поскольку не все реляционные СУБД поддерживают их, мы воспользуемся универсальным типом `int`.

Столбцы, содержащие значения строкового типа разной длины, такие как `CategoryName`, обычно определяются как тип `varchar` (от variable character, «переменное количество символов»). Этот тип данных учитывает, что длина строк (количество символов) может отличаться, благодаря чему он обеспечивает более эффективный расход памяти по сравнению с типом `char`, который всегда резервирует фиксированное количество символов, заданное при определении столбца.

Хотя указание максимальной длины для `varchar` и не обязательно, лучше все же это сделать — иначе СУБД может использовать неоптимальное значение по умолчанию, которое к тому же различается от системы к системе. Значения в нашем столбце `CategoryName` не будут превышать 20 символов, поэтому определим тип данных как `varchar(20)`.

НА ЗАМЕТКУ В ходе общения с другими специалистами по SQL можно заметить отсутствие единого стандарта произношения термина *varchar*. Одни говорят «вар-чар», другие — «вар-кар», а кто-то даже — «вэр-кэр». Последний вариант, вероятно, наиболее правильный с точки зрения грамматики, поскольку гласные в нем совпадают с произношением первых слогов слов *variable character*, однако, по моим наблюдениям, он употребляется реже остальных. Впрочем, не ломайте голову — пусть каждый говорит, как привык. Как сказали бы французы, *vive la différence* — «Да здравствует разнообразие»!

18.1.2. Создаем пустую таблицу

Теперь, когда мы определились с названием таблицы, именами столбцов и типами данных, сформулируем нашу задачу на естественном языке: “I would like to create a table named category. I would like the table to have a column named CategoryID that is an int data type. I would like the table to also have a column named CategoryName that is a varchar(20) data type” («Хочу создать таблицу с именем category. В таблице должен быть столбец с именем CategoryID типа `int`. В ней также должен быть столбец с именем CategoryName типа varchar(20)»). В итоге SQL-запрос для создания такой таблицы обретет следующий вид:

```
CREATE TABLE category (
    CategoryID int,
    CategoryName varchar(20)
    );
```

Обратите внимание: после того, как мы задаем имя таблицы с помощью команды `CREATE TABLE`, мы перечисляем все столбцы, разделяя имена и типы данных запятыми. Такой формат должен показаться вам интуитивно понятным, ведь мы уже применяли запятые для разделения столбцов в предложениях `SELECT` и `ORDER BY`. Также заметьте, что весь список заключается в круглые скобки. Пропуск скобок при использовании `CREATE TABLE` — весьма распространенная ошибка начинающих SQL-разработчиков, поэтому не забывайте ставить их при создании таблицы.

После выполнения приведенного выше запроса панель результатов не отобразит никаких данных, но в панели вывода появится зеленый кружок с белой галочкой и сообщением 0 row(s) affected («Затронуто 0 строк»). Несмотря на лаконичность такого уведомления, оно свидетельствует об успешном создании таблицы.

Данный фрагмент кода представляет собой лишь базовую форму команды создания таблицы. Как мы увидим позже в этой и последующих главах, команда `CREATE TABLE` располагает целым рядом возможностей для задания дополнительных свойств таблицы. А пока двинемся дальше и применим знания о ключевом слове `INSERT` из главы 16.

18.1.3. Вносим значения в пустую таблицу

Поскольку таблица `customer` пока пустует, добавим в нее значения идентификаторов (`CategoryID`) и названий (`CategoryName`) для следующих категорий:

1. `Romance` («Романтика»).
2. `Humor` («Юмор»).
3. `Mystery` («Детектив»).
4. `Fantasy` («Фэнтези»).
5. `Science Fiction` («Научная фантастика»).

В главе 16 мы научились добавлению нескольких значений в таблицу посредством ключевых слов `INSERT` и `VALUES`. Применим тот же подход для вставки перечисленных значений в нашу новую таблицу `category`:

```
INSERT INTO category (CategoryID, CategoryName)
VALUES
    (1, 'Romance'),
    (2, 'Humor'),
    (3, 'Mystery'),
    (4, 'Fantasy'),
    (5, 'Science Fiction');
```

Выполнение этого фрагмента SQL-кода вновь не отобразит никаких данных на панели результатов, однако если оно прошло успешно, в столбце сообщений на панели вывода мы увидим 5 row(s) affected («Затронуто 5 строк»). Это означает, что в таблицу `category`, как мы и планировали, добавлено пять новых строк.

К практике!

Если вы еще не создали и не заполнили таблицу `category`, выполните SQL-код, представленный в разделах 18.1.2 и 18.1.3. Мы будем работать с этой таблицей в этой и последующих главах.

Чтобы убедиться, что требуемые значения внесены в таблицу `category`, выполним простой `SELECT`-запрос для просмотра ее содержимого (результат представлен на рис. 18.1):

```
SELECT
    CategoryID,
    CategoryName
FROM category;
```

CategoryID	CategoryName
1	Romance
2	Humor
3	Mystery
4	Fantasy
5	Science Fiction

Рис. 18.1. Все строки новой таблицы category

18.2. Модификация таблицы

Следующий шаг в присвоении категории каждому произведению — это добавление нового столбца в таблицу `title`, который будет ссылаться на нашу таблицу `category`. А именно мы хотим связать значения `CategoryID` из таблицы `title` с соответствующими значениями в таблице `category`. Мы сделаем это, добавив столбец `CategoryID` в таблицу `title`.

18.2.1. Добавляем столбец в таблицу

Подобно тому как мы задавали себе три ключевых вопроса перед созданием таблицы, при добавлении столбца тоже стоит задуматься о трех аспектах:

- К таблице с каким именем мы добавляем новый столбец?
- Какие имена будут у столбцов, которые мы добавим в эту таблицу?
- Какие типы данных будут у этих столбцов?

Нам известен ответ на первый вопрос — таблица `title`. Поскольку мы хотим сохранить согласованность в именах и типах данных, ответы на второй и третий вопросы очевидны: столбец должен называться `CategoryID` и иметь тип `int`, аналогично соответствующему столбцу в таблице `category`.

Для внесения этих и других изменений в таблицу в SQL служит новая команда: `ALTER TABLE`. Несмотря на сходство с `CREATE TABLE`, она не требует применения круглых скобок. Ниже приведен SQL-код для добавления столбца в таблицу `title`:

```
ALTER TABLE title
ADD CategoryID int;
```

Как и в случае с командой CREATE TABLE, этот запрос не вернет никаких результатов. Его успешное выполнение отмечено на панели вывода только белой галочкой в зеленом кружке и сообщением 0 row(s) affected («Затронуто 0 строк»).

Чтобы убедиться, что столбец действительно создан, выполним проверочный запрос. На рис. 18.2 видно, что к существующим столбцам добавился новый:

```
SELECT *
FROM title;
```

TitleID	TitleName	Price	Advance	Royalty	PublicationDate	CategoryID
101	Pride and Predicates	9.95	5000.00	15.00	2015-04-30 00:00:00	NULL
102	The Join Luck Club	9.95	6000.00	12.00	2016-02-06 00:00:00	NULL
103	Catcher in the Try	8.95	5000.00	10.00	2017-04-03 00:00:00	NULL
104	Anne of Fact Tables	12.95	10000.00	15.00	2018-01-12 00:00:00	NULL
105	The DateTime Machine	7.95	5500.00	15.00	2019-02-04 00:00:00	NULL
106	The Great GroupBy	10.95	0.00	20.00	2019-12-23 00:00:00	NULL
107	The Call of the While	8.95	2500.00	15.00	2020-03-14 00:00:00	NULL
108	The Sum Also Rises	7.95	5000.00	12.00	2021-11-12 00:00:00	NULL
109	David Emptyfield	9.95	0.00	10.00	2022-01-16 00:00:00	NULL
110	Red Badge of Cursors	7.95	0.00	15.00	2022-03-29 00:00:00	NULL
111	Of Mice and Metadata	8.95	0.00	12.00	2022-05-17 00:00:00	NULL
112	A Table of Two Cities	9.95	0.00	15.00	2022-08-07 00:00:00	NULL

Рис. 18.2. Таблица title с новым столбцом CategoryID в крайней правой позиции

Наш новый столбец пока не содержит значений, поэтому результаты запроса показывают NULL в каждой строке. Чтобы заполнить значения, мы задействуем команду UPDATE. Для вставки значений в таблицу category мы прибегли к INSERT, которое добавляет целые строки. Здесь так не получится; нам требуется добавить значение для одного столбца в существующие строки, что и позволяет сделать UPDATE.

Заполним значения для всех строк, указав категорию для каждой книги. Если вы не запомнили все значения CategoryID и CategoryName (или просто не хотите возвращаться к предыдущим страницам), воспользуйтесь поясняющими комментариями, которыми снабжен SQL-код :

```
/* 1 – Romance */
UPDATE title
SET CategoryID = 1
WHERE TitleID IN (101, 104);

/* 2 – Humor */
UPDATE title
SET CategoryID = 2
WHERE TitleID IN (106, 109);
```

```
/* 3 – Mystery */
UPDATE title
SET CategoryID = 3
WHERE TitleID IN (102, 103, 110);

/* 4 – Fantasy */
UPDATE title
SET CategoryID = 4
WHERE TitleID IN (107, 112);

/* 5 – Science Fiction */
UPDATE title
SET CategoryID = 5
WHERE TitleID IN (105, 108, 111);
```

Запустив все приведенные выше инструкции `UPDATE`, мы наконец увидим заполненный столбец `CategoryID` для всех строк. Выполним финальный запрос, чтобы вывести содержимое таблицы `title` — как следует из рис. 18.3, все значения `CategoryID` теперь добавлены:

```
SELECT *
FROM title;
```

TitleID	TitleName	Price	Advance	Royalty	PublicationDate	CategoryID
101	Pride and Predicates	9.95	5000.00	15.00	2015-04-30 00:00:00	1
102	The Join Luck Club	9.95	6000.00	12.00	2016-02-06 00:00:00	3
103	Catcher in the Try	8.95	5000.00	10.00	2017-04-03 00:00:00	3
104	Anne of Fact Tables	12.95	10000.00	15.00	2018-01-12 00:00:00	1
105	The DateTime Machine	7.95	5500.00	15.00	2019-02-04 00:00:00	5
106	The Great GroupBy	10.95	0.00	20.00	2019-12-23 00:00:00	2
107	The Call of the While	8.95	2500.00	15.00	2020-03-14 00:00:00	4
108	The Sum Also Rises	7.95	5000.00	12.00	2021-11-12 00:00:00	5
109	David Emptyfield	9.95	0.00	10.00	2022-01-16 00:00:00	2
110	Red Badge of Cursors	7.95	0.00	15.00	2022-03-29 00:00:00	3
111	Of Mice and Metadata	8.95	0.00	12.00	2022-05-17 00:00:00	5
112	A Table of Two Cities	9.95	0.00	15.00	2022-08-07 00:00:00	4

Рис. 18.3. Таблица title с новым столбцом CategoryID с заполненными значениями для всех строк

Мы удостоверились в том, что идентификаторы категорий (`CategoryID`) проставлены, но эти значения напрямую не указывают названия категорий (`CategoryName`) для каждого произведения. Давайте выведем названия категорий для книг (`TitleName`), написав запрос, который свяжет таблицы `title` и `category` по `CategoryID`. Эта связь устанавливается при помощи внутреннего соединения `INNER JOIN` (о котором говорилось в главе 8). Результаты на рис. 18.4, отсортированные по `TitleID`, отображают категорию для каждого произведения:

```
SELECT
    t.TitleID,
    t.TitleName,
    c.CategoryName
FROM title t
INNER JOIN category c
    ON t.CategoryID = c.CategoryID
ORDER BY t.TitleID;
```

TitleID	TitleName	CategoryName
101	Pride and Predicates	Romance
102	The Join Luck Club	Mystery
103	Catcher in the Try	Mystery
104	Anne of Fact Tables	Romance
105	The DateTime Machine	Science Fiction
106	The Great GroupBy	Humor
107	The Call of the While	Fantasy
108	The Sum Also Rises	Science Fiction
109	David Emptyfield	Humor
110	Red Badge of Cursors	Mystery
111	Of Mice and Metadata	Science Fiction
112	A Table of Two Cities	Fantasy

Рис. 18.4. Названия категорий (CategoryName) для каждой книги (TitleName) в таблице title, присоединенные через CategoryID

18.2.2. Что следует учесть при добавлении нового столбца

Добавление категории для каждого заголовка представляет собой сравнительно простую операцию. Тем не менее, как и при создании таблицы, перед добавлением новых столбцов следует учитывать ряд моментов.

Многим начинающим пользователям SQL кажется странным, что новый столбец появляется в самом конце таблицы. Так и задумано — новые столбцы всегда добавляются в конец. Впрочем, некоторые реляционные СУБД (такие, как MySQL) позволяют изменить порядок столбцов. Но делать это, как правило, не советуют, особенно если таблица уже содержит данные.

Существует две основные причины, по которым новые столбцы рекомендуется добавлять только в конец таблицы:

- Вставка столбца в середину структуры таблицы требует перестройки физического порядка хранения данных, что сопровождается значительными вычислительными затратами и использованием дополнительных ресурсов. Добавляя столбец в конец, мы минимизируем количество операций ввода-вывода, нагрузку на систему и общее время выполнения команды.

- В целом, нет необходимости располагать столбцы таблицы в какой-то фиксированной позиции. Как мы не раз убеждались ранее, отображать столбцы можно в любой последовательности, просто указав порядок их вывода в SQL-запросе.

Единственным случаем, когда добавление нового столбца в конец таблицы не является предпочтительным, считается создание столбца, предназначенного для использования в качестве *первичного ключа* (primary key). Обычно такой столбец размещают первым, поскольку первичный ключ зачастую определяет физический или логический порядок организации данных в таблице. Ранее, начиная с главы 8, первичные ключи мы уже не раз упоминали, а теперь, в разделе 18.3, подробно разберем, как их создавать и применять.

18.3. Первичные ключи

Наше обучение построено таким образом, чтобы вы постепенно осваивали написание SQL-запросов, и потому теме первичных ключей (primary keys) до сих пор я уделял не так много внимания. Однако теперь, когда мы перешли к проектированию таблиц и построению связей между данными, настало время вернуться к ней и рассмотреть все основательно.

18.3.1. Что важно знать о первичных ключах

Первичные ключи — опорный компонент любой реляционной БД, ведь именно они позволяют устанавливать связи между таблицами. При работе с первичными ключами необходимо соблюдать ряд правил:

- *Первичный ключ должен быть уникальным* — это главное правило. Если первичный ключ не уникален, мы не сможем установить однозначное соответствие с конкретной строкой в связанной таблице. Представьте, что мы присвоили значение 1 столбцу `CategoryID` для каждой из пяти строк в новой таблице `category`. Если у всех пяти строк одно и то же значение, невозможно определить, какое название категории (`CategoryName`) соответствует каждой строке, ссылающейся на `CategoryID = 1`.
- *Значение первичного ключа должно быть задано для каждой строки*. Пустые значения здесь недопустимы: связать NULL не получится ни с одним другим значением, даже с другим NULL. Поэтому строка, в которой первичный ключ равен NULL, окажется изолированной — с ней невозможно будет установить никакие связи.
- *Значения первичных ключей нельзя изменять*. Именно по ним таблицы узнают друг друга — к примеру, `CategoryID` связывает таблицу `category` с таблицей `title`. Если мы изменим значения `CategoryID` в таблице `category`, скажем,

прибавив к каждому 10, данные двух таблиц перестанут совпадать, и связь между ними попросту исчезнет.

Соблюдение этих трех правил позволяет поддерживать *ссылочную целостность* (referential integrity) связей, задействующих первичные ключи. Это означает, что любое значение из одной таблицы, ссылающееся на первичный ключ в другой таблице, неизменно будет указывать на то же самое конкретное значение в таблице, где задан этот первичный ключ. Хотя значения других столбцов в таблице могут подвергаться правке (так, мы можем поменять значение в столбце `CategoryName` с `Mystery` на `Mystery Thriller` — «Детективный триллер»), значения первичных ключей нельзя изменять ни при каких обстоятельствах.

18.3.2. Добавляем первичный ключ (PRIMARY KEY)

Итак, зная требования к первичным ключам, создадим такой ключ для таблицы `category`. В этих целях мы воспользуемся уже знакомой командой `ALTER TABLE`, но с некоторыми отличиями:

```
ALTER TABLE category
ADD CONSTRAINT PRIMARY KEY (CategoryID);
```

Между добавлением обычного столбца и добавлением первичного ключа при помощи `ALTER TABLE` есть два заметных различия. Во-первых, при создании первичного ключа применяется ключевое слово `CONSTRAINT`. Раньше я это не уточнял, но на самом деле первичный ключ — это тоже *ограничение* (constraint), то есть объект базы данных, контролирующий соблюдение логических правил, установленных в ее структуре. После того как ограничение создано, нарушить его уже невозможно. Таким образом, добавив новое ограничение `PRIMARY KEY`, мы обязаны следовать тем правилам, о которых говорилось в разделе 18.3.1 для таблицы `category`.

Во-вторых, хотя это и не обязательно, первичному ключу обычно присваивают логическое имя. Делают это потому, что в таблице могут быть и другие объекты с различными ограничениями, и имена этих объектов должны отражать их назначение. Подобно тому как мы называем таблицы в соответствии с типом данных (`customer`, `author`), первичным ключам также стоит давать понятные имена. Кроме того, как станет ясно в дальнейшем, наличие имени у первичного ключа (или любого другого ограничения) позволяет упростить обращение к нему при редактировании или удалении.

Наиболее распространенная схема именования первичных ключей — префикс `PK`, за которым следует символ подчеркивания и имя таблицы. Даже если вам ничего не известно о конкретной базе данных, увидев объект с именем `PK_category`, вы сразу поймете, что перед вами первичный ключ (то есть ограничение `PRIMARY`

KEY) таблицы category. По этой причине желательно и нам придерживаться подобного соглашения об именовании при создании первичного ключа:

```
ALTER TABLE category
ADD CONSTRAINT PK_category PRIMARY KEY (CategoryID);
```

К практике!

Выполнив представленный выше SQL-код, создайте ограничение PRIMARY KEY для таблицы category.

В нашем примере мы добавили первичный ключ к уже существующей таблице, однако в большинстве случаев его задают сразу при ее создании. Сделать это несложно — достаточно указать первичный ключ в инструкции CREATE TABLE после перечисления столбцов, как если бы вы добавляли еще один столбец. Таблицу category можно было изначально создать с первичным ключом при помощи следующего SQL-кода:

```
CREATE TABLE category (
    CategoryID int,
    CategoryName varchar(20),
    CONSTRAINT PK_category PRIMARY KEY (CategoryID)
    );
```

СОВЕТ Хотя создавать первичный ключ для каждой таблицы в каждой базе данных вовсе не обязательно, первичный ключ должен присутствовать в любой таблице, содержащей (за неимением лучшего определения) набор уникальных объектов. Таблицы со строками, представляющими уникальные объекты, такие как товары, заказы и клиенты, всегда должны содержать первичный ключ, гарантирующий, что каждая строка останется единственной в своем роде.

И последнее замечание о первичных ключах: они могут быть составными, то есть состоять из нескольких столбцов. К примеру, наша таблица orderitem должна иметь первичный ключ, включающий как идентификатор заказа (OrderID), так и идентификатор товара (ItemID), поскольку вместе эти столбцы образуют уникальный ключ. Несколько строк могут иметь одинаковый OrderID, но каждая из этих строк должна также иметь уникальный ItemID для каждого повторяющегося OrderID.

Для создания первичного ключа из нескольких столбцов в такой таблице, как orderitem, достаточно перечислить столбцы через запятую:

```
ALTER TABLE orderitem
ADD CONSTRAINT PK_orderitem PRIMARY KEY (OrderID, OrderItem);
```

Применение первичных ключей обеспечивает уникальность каждой строки в таблице. На эти строки часто ссылаются другие таблицы через связи, и мы можем явным образом задать эти связи посредством другого типа ограничения — внешнего ключа.

18.4. Внешние ключи и ограничения

Внешние ключи (foreign keys) создают для того, чтобы база данных сама следила за соблюдением правил взаимосвязей между двумя таблицами, объединенными по общему столбцу. Любая связь, основанная на внешнем ключе, включает *родительскую таблицу* (parent table), в которой задают значения ключа, и *дочерние таблицы* (child tables), содержащие значения, ссылающиеся на родительскую таблицу. Проще говоря, внешний ключ в дочерней таблице призван гарантировать наличие всех значений дочерней таблицы в родительской.

18.4.1. Диаграммы данных

Диаграммы данных (data diagrams) служат средством наглядного отображения отношений между таблицами в структуре базы данных. Они позволяют визуализировать объекты, связи и направления зависимостей между ними. На рис. 18.5 представлена диаграмма данных, включающая все таблицы базы данных `sqlnovel`, в том числе недавно добавленную таблицу `category`.

Каждый прямоугольный блок на диаграмме — это отдельная таблица, внутри которой приведен список всех ее столбцов и соответствующих типов данных. Основное внимание следует уделить линиям между таблицами — они обозначают реляционные связи. Если между двумя таблицами проведена линия, значит, между ними существует связь по внешнему ключу.

На диаграмме данных (рис. 18.5) не отмечено, какие именно столбцы участвуют в этих связях, однако это несложно выяснить, если внимательно изучить структуру таблиц. В процессе выполнения упражнений, представленных в этой главе и в практических заданиях, вы научитесь создавать собственную диаграмму данных.

И последнее любопытное наблюдение: таблица `myfirstquery` стоит особняком — к ней не ведет ни одной линии. Это потому, что она не связана ни с одной другой таблицей в БД и создана лишь для вашего первого запроса в главе 2. С тех пор мы продвинулись далеко вперед!

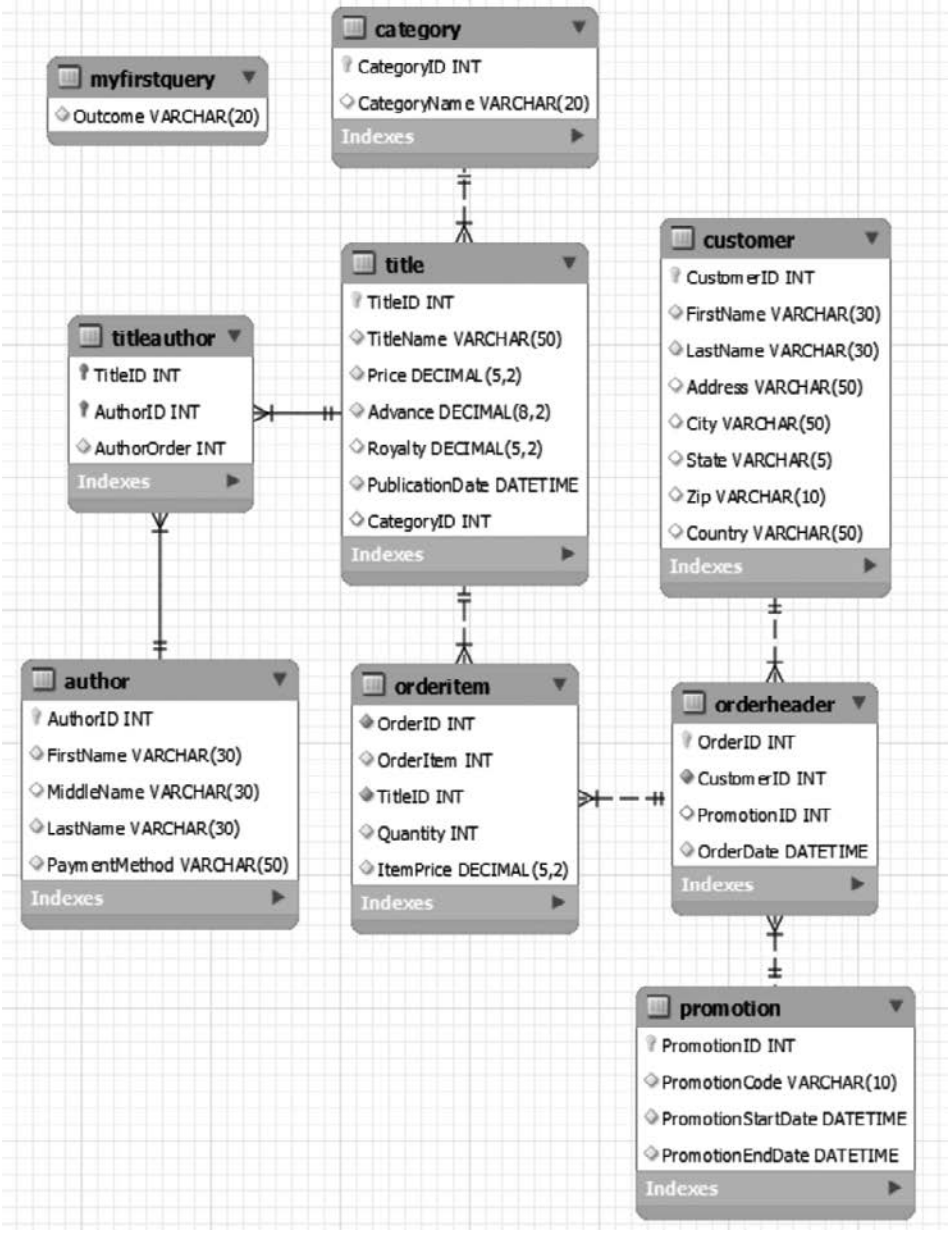


Рис. 18.5. Диаграмма данных, отображающая все таблицы в базе данных sqlnovel и существующие между ними реляционные связи

18.4.2. Добавляем ограничение внешнего ключа (FOREIGN KEY)

Поскольку внешний ключ — это еще один тип ограничения, мы снова воспользуемся командой ALTER TABLE, похожей на ту, что мы применяли для создания ограничения PRIMARY KEY. Как и в случае с первичным ключом, целесообразно присвоить ограничению FOREIGN KEY логическое имя. Общепринятая схема именования внешнего ключа — сначала префикс FK и символ подчеркивания, затем имя дочерней таблицы, символ подчеркивания и, наконец, имя родительской таблицы. Этот формат гарантирует уникальность имени нашего ограничения FOREIGN KEY в базе данных. Следуя соглашению об именовании, можно придать нашему ограничению внешнего ключа форму следующей инструкции:

```
ALTER TABLE title
ADD CONSTRAINT FK_title_category
FOREIGN KEY (CategoryID) REFERENCES category(CategoryID);
```

К практике!

Выполнив представленную выше SQL-инструкцию, создайте ограничение FOREIGN KEY для таблицы title.

Эта команда несколько отличается от применявшейся для создания ограничения PRIMARY KEY, поскольку мы создаем реляционную связь между двумя столбцами разных таблиц. Первый столбец, указанный после ключевых слов FOREIGN KEY, задает столбец в дочерней таблице, который ссылается на столбец в родительской таблице. Именно поэтому используется ключевое слово REFERENCES — оно указывает, на какую таблицу и какой столбец ведет ссылка.

Если таблица уже содержит данные, необходимо соблюдать особую осторожность при создании ограничений — будь то PRIMARY KEY, FOREIGN KEY или другие. Если существующие значения не соответствуют требованиям нового ограничения, при выполнении команды возникнет ошибка. Поэтому разумнее всего создавать все ограничения одновременно с таблицей, до вставки каких-либо строк данных.

СОВЕТ В этой главе мы придерживаемся общепринятых правил именования ограничений, тем не менее это далеко не единственный возможный подход. При работе с другими базами данных следует выяснить, применяются ли в них уже установленные соглашения об именовании. Если такие стандарты существуют, лучше следовать им, чтобы обеспечить согласованность со всеми остальными объектами БД.

18.5. Удаление таблицы, столбца или ограничения

Несмотря на то что уже созданные объекты нам еще пригодятся, при необходимости их можно удалить при помощи инструкций с ключевым словом DROP.

> **ВНИМАНИЕ** SQL-код, представленный ниже, приведен исключительно в ознакомительных целях. Не следует удалять таблицу `category` или любые связанные с ней столбцы и ограничения: нам понадобятся эти данные в последующих главах. Если же вы, невзирая на предупреждение, решите потренироваться в удалении этих объектов, то для их воссоздания вам придется повторно выполнить операции, описанные чуть выше.

Теперь, когда предупреждение озвучено, рассмотрим, как можно удалить объекты, созданные в этой главе.

18.5.1. Удаляем ограничение

Удаление ограничения, как и его добавление, выполняется командой `ALTER TABLE`. Но так как мы именно удаляем, а не создаем, достаточно указать только таблицу и имя самого ограничения. Соответствующий SQL-код будет выглядеть так:

```
ALTER TABLE title
DROP FOREIGN KEY FK_title_category;
```

Допустим, нам нужно удалить первичный ключ `PK_category` нашей таблицы `category`. Сделать это можно одним из двух способов. Способ первый — удалить его как ограничение, вот так:

```
ALTER TABLE category
DROP CONSTRAINT PK_category;
```

Впрочем, поскольку первичный ключ является особым типом ограничения, мы можем задействовать `ALTER TABLE`, чтобы удалить первичный ключ, не указывая имя ограничения:

```
ALTER TABLE category
DROP PRIMARY KEY;
```

> **НА ЗАМЕТКУ** В команде нам не нужно указывать имя первичного ключа, потому что в таблице может быть только один первичный ключ.

18.5.2. Удаляем столбец

В этой главе мы создали столбец в таблице `title`. Для его удаления понадобится связка `ALTER TABLE` и `DROP`:

```
ALTER TABLE title
DROP COLUMN CategoryID;
```

Как и в случае с ограничениями, для удаления столбца обычно достаточно знать лишь имя таблицы и самого столбца.

18.5.3. Удаляем таблицу

Для удаления таблицы существует самый лаконичный синтаксис среди всех операций удаления объектов — достаточно команды `DROP TABLE` и имени таблицы:

```
DROP TABLE category;
```

> **НА ЗАМЕТКУ** Если требуется удалить все эти объекты, действовать нужно именно в том порядке, который описан в разделе 18.5: сначала удалить ограничения. Если же мы попытаемся сначала удалить таблицу или столбец, большинство СУБД (включая MySQL) вернут сообщение об ошибке, указывающее, что в силу ограничений объект не может быть удален.

На этом мы завершим рассмотрение вопросов, связанных с удалением объектов, и перейдем к заданию, в рамках которого сможем попрактиковаться в создании таких ограничений, как первичные и внешние ключи.

18.6. Практическое занятие

1. В базе данных `sqlnovel` отсутствует несколько первичных ключей. С опорой на диаграмму данных из раздела 18.4.1 и схему именования из раздела 18.3.2 составьте и выполните SQL-команды для добавления ограничений `PRIMARY KEY` к следующим таблицам:
   - `author`
   - `customer`
   - `orderheader`
   - `promotion`
   - `title`
   - `titleauthor`
2. В предыдущем списке нет таблицы `orderitem`. Что произойдет, если попытаться создать ограничение `PRIMARY KEY` для столбцов `OrderID` и `OrderItem`? Каким образом можно решить такую проблему?
3. В базе данных `sqlnovel` также отсутствует ряд ограничений внешнего ключа. С опорой на диаграмму данных из раздела 18.4.1 и схему именования из раздела 18.4.2 составьте и выполните SQL-команды для добавления ограничений `FOREIGN KEY` для следующих таблиц и столбцов:
   - столбец `CustomerID` в таблице `orderheader`;
   - столбец `PromotionID` в таблице `orderheader`;
   - столбец `OrderID` в таблице `orderitem`;

- столбец `TitleID` в таблице `orderitem`;
- столбец `TitleID` в таблице `titleauthor`;
- столбец `AuthorID` в таблице `titleauthor`.

4. Если вы справились со всеми предыдущими заданиями, настал момент насладиться результатом! Создайте диаграмму данных. В меню MySQL Workbench выберите Database ▸ Reverse Engineer (База данных ▸ Обратное проектирование). Нажимайте Next (Далее) на всех последующих экранах и не забудьте выбрать флажок sqlnovel на экране Select Schemas to Reverse Engineer («Выбрать схемы для обратного проектирования»). По завершении у вас должна получиться диаграмма данных базы `sqlnovel`.

18.7. Ответы

1. Ограничения `PRIMARY KEY` для указанных таблиц можно создать при помощи следующих SQL-команд:

```
ALTER TABLE author
    ADD CONSTRAINT PK_author PRIMARY KEY (AuthorID);
ALTER TABLE customer
    ADD CONSTRAINT PK_customer PRIMARY KEY (CustomerID);
ALTER TABLE orderheader
    ADD CONSTRAINT PK_orderheader PRIMARY KEY (OrderID);
ALTER TABLE promotion
    ADD CONSTRAINT PK_promotion PRIMARY KEY (PromotionID);
ALTER TABLE title
    ADD CONSTRAINT PK_title PRIMARY KEY (TitleID);
ALTER TABLE titleauthor
    ADD CONSTRAINT PK_titleauthor PRIMARY KEY (TitleID, AuthorID);
```

2. SQL-код для создания ограничения `PRIMARY KEY` в таблице `orderitem` будет таким:

```
ALTER TABLE orderitem
    ADD CONSTRAINT PK_orderitem PRIMARY KEY (OrderID, ItemID);
```

Однако при его выполнении в окне вывода отображается сообщение об ошибке: Error Code: 1062. Duplicate entry '1022-1' for key 'orderitem.PRIMARY' («Ошибка 1062: Найден дубликат '1022-1' для ключа 'orderitem.PRIMARY'»). Это уведомление сигнализирует о противоречии в данных для формируемого первичного ключа и указывает, в чем именно оно состоит. Ошибка соответствует значениям `OrderID` 1022 и `OrderItem` 1. В наличии проблемы можно убедиться, выполнив следующий запрос, который должен возвращать одну строку, но вместо этого выводит две (рис. 18.6):

```
SELECT *
FROM orderitem
```

```
WHERE OrderID = 1022
    AND OrderItem = 1;
```

OrderID	OrderItem	TitleID	Quantity	ItemPrice
1022	1	101	1	7.95
1022	1	103	1	6.95

Рис. 18.6. Две строки, препятствующие созданию первичного ключа для таблицы orderitem

Итак, имеем две строки, приводящие к дублированию значений нашего первичного ключа, что недопустимо. Все значения ключа должны быть уникальными. К счастью, эти строки — не полные дубликаты, поскольку у них разные значения `TitleID`. Можно легко исправить эту ошибку в данных командой `UPDATE`, изменив значение `OrderItem` с 1 на 2 для одной из строк:

```
UPDATE orderitem
SET OrderItem = 2
WHERE OrderID = 1022
    AND OrderItem = 1
    AND TitleID = 103;
```

После выполнения указанной `UPDATE`-команды можно создавать первичный ключ для таблицы `orderitem`. А теперь смело добавьте ограничение `PRIMARY KEY`.

3. Ограничения `FOREIGN KEY` для указанных таблиц и столбцов можно создать при помощи следующих SQL-команд:

```
ALTER TABLE orderheader
    ADD CONSTRAINT FK_orderheader_customer FOREIGN KEY (CustomerID)
    REFERENCES customer(CustomerID);
ALTER TABLE orderheader
    ADD CONSTRAINT FK_orderheader_promotion FOREIGN KEY (PromotionID)
    REFERENCES promotion(PromotionID);
ALTER TABLE orderitem
    ADD CONSTRAINT FK_orderitem_orderheader FOREIGN KEY (OrderID)
    REFERENCES orderheader(OrderID);
ALTER TABLE orderitem
    ADD CONSTRAINT FK_orderitem_title FOREIGN KEY (TitleID)
    REFERENCES title(TitleID);
ALTER TABLE titleauthor
    ADD CONSTRAINT FK_titleauthor_title FOREIGN KEY (TitleID)
    REFERENCES title(TitleID);
ALTER TABLE titleauthor
    ADD CONSTRAINT FK_titleauthor_author FOREIGN KEY (AuthorID)
    REFERENCES author(AuthorID);
```

4. Успешное создание диаграммы данных указывает на правильный ответ, но единственного верного варианта не существует. Поэкспериментируйте с расположением таблиц, перемещая их, чтобы линии, представляющие реляционные связи, выглядели нагляднее. При этом обязательно наводите курсор на линии, чтобы увидеть, как подсвечиваются столбцы, задействованные в этих связях.

19 Создание ограничений и индексов

В главе 18 речь шла о двух важнейших ограничениях: первичных ключах (PRIMARY KEY) и внешних ключах (FOREIGN KEY). Теперь мы изучим ряд других ограничений, способствующих обеспечению целостности данных в таблицах.

Мы также рассмотрим *индексы* (indexes) — структурированные объекты, связанные с таблицами, заметно ускоряющие работу запросов. Подобно тому как предметный указатель в конце книги помогает быстро найти нужную тему, индекс в базе данных позволяет существенно сократить время выдачи запросом конкретных данных.

Надеюсь, вам понравилось создавать таблицы и применять к ним ограничения, ибо впереди нас ждет еще больше практики!

19.1. Ограничения

Работая с примерами из главы 18, вы освоили два способа создания ограничений: посредством ALTER TABLE для существующих таблиц и при помощи CREATE TABLE для новых.

С командой CREATE TABLE также существует два метода установки ограничения. Во-первых, можно объявить ограничение после перечисления всех столбцов, как мы делали в главе 18. Вот пример ограничения PRIMARY KEY для нашей таблицы category:

```
CREATE TABLE category (
    CategoryID int,
    CategoryName varchar(20),
    CONSTRAINT PK_category PRIMARY KEY (CategoryID)
    );
```

Во-вторых, ограничение можно задать и прямо при объявлении столбца — сразу после типа данных. Для первичного ключа таблицы `category` это выглядело бы так:

```
CREATE TABLE category (
    CategoryID int PRIMARY KEY,
    CategoryName varchar(20)
    );
```

Безусловно, второй метод проще, однако создание ограничения таким способом не позволяет разработчику присвоить ему осмысленное имя. СУБД сама сгенерирует ограничению некое техническое название из набора цифр и букв, которое ничего не говорит о его назначении.

Впрочем, не всем ограничениям нужно явное имя. Хотя я настоятельно рекомендую задействовать первый метод для создания ограничений `PRIMARY KEY` и `FOREIGN KEY`, другие ограничения, рассматриваемые в этой главе, в большинстве своем объявляются вторым методом.

19.1.1. Ограничения NOT NULL

Ограничение `NOT NULL` принудительно требует отсутствия в столбце NULL-значений — что, если задуматься, вполне справедливо для большинства столбцов любой таблицы. Ведь таблицы создаются для хранения информации, и если не все, то многие столбцы таблицы должны иметь значения.

Допустим, мы хотим добавить в базу данных `sqlnovel` таблицу для отслеживания отправки книг покупателям. Назовем ее `shipment` и добавим следующие шесть столбцов:

- `ShipmentID` — идентифицирует каждую уникальную строку в таблице;
- `OrderID` — идентифицирует заказ, к которому относится отправление;
- `ShipmentCost` — указывает стоимость отправления в долларах США;
- `ShipmentMethod` — определяет способ доставки: посылка (P) или экспресс-доставка (E);
- `TrackingNumber` — содержит номер для отслеживания (трек-номер), предоставленный компанией-перевозчиком;
- `ShipmentDate` — указывает дату отправки заказа.

Присвоим столбцам следующие типы данных:

- `ShipmentID` — `int`, уникальное целочисленное значение;
- `OrderID` — `int`, такой же, как в таблице `orderheader`;
- `ShipmentCost` — `decimal(5,2)` для сумм от 0.00 до 999.99;

- `ShipmentMethod` — `char(1)`, поскольку значение будет либо P, либо E;
- `TrackingNumber` — `varchar(20)` для любого значения, предоставляемого компанией-перевозчиком;
- `ShipmentDate` — `datetime`, значение даты и времени.

Также требуется создать ограничение `PRIMARY KEY` с именем `PK_shipment` для столбца `ShipmentID` и ограничение `FOREIGN KEY` с именем `FK_shipment_orderheader` для столбца `OrderID`, которое ссылается на значения `OrderID` в таблице `orderheader`. Располагая всей этой информацией, создадим таблицу `shipment` посредством следующей SQL-команды:

```
CREATE TABLE shipment (
    ShipmentID int,
    OrderID int,
    ShipmentCost decimal(5,2),
    ShipmentMethod char(1),
    TrackingNumber varchar(20),
    ShipmentDate datetime,
    CONSTRAINT PK_shipment PRIMARY KEY (ShipmentID),
    CONSTRAINT FK_shipment_orderheader FOREIGN KEY (OrderID)
        REFERENCES orderheader(OrderID)
    );
```

НА ЗАМЕТКУ Пока не выполняйте этот SQL-запрос. В ходе изложения материала главы мы его существенно доработаем.

Следующий шаг — решить, какие столбцы в нашей таблице могут оставаться пустыми (nullable), то есть *допускать пустые значения* (NULL). Для всех столбцов, которые не должны содержать NULL-значений, следует добавить ограничение `NOT NULL`, чтобы гарантировать заполнение этих столбцов во всех строках таблицы. Примером столбца, который может оставаться пустым, является `MiddleName` в таблице `author`: у некоторых авторов есть второе имя, а у других нет. Проанализируем, насколько обязательно заполнение каждого столбца таблицы `shipment`:

- `ShipmentID` не допускает NULL, поскольку в первичном ключе не может быть значений NULL;
- `OrderID` не допускает NULL, поскольку каждая отправка товара должна быть связана с заказом;
- `ShipmentCost` не допускает NULL, поскольку каждое отправление имеет стоимость 0.00 доллара или выше;
- `ShipmentMethod` не допускает NULL, поскольку нам необходимо знать способ доставки каждого заказа;

- `TrackingNumber` не допускает NULL, поскольку каждое отправление имеет номер для отслеживания;
- `ShipmentDate` не допускает NULL, поскольку нам необходимо знать, когда была совершена отправка.

При внимательном рассмотрении выясняется, что ни один из столбцов не допускает пустых значений, поэтому к каждому из них требуется добавить ограничение `NOT NULL`. Для этого доработаем наш SQL-запрос, указав `NOT NULL` после объявления типа данных для каждого столбца:

```
CREATE TABLE shipment (
    ShipmentID int NOT NULL,
    OrderID int NOT NULL,
    ShipmentCost decimal(5,2) NOT NULL,
    ShipmentMethod char(1) NOT NULL,
    TrackingNumber varchar(20) NOT NULL,
    ShipmentDate datetime NOT NULL,
    CONSTRAINT PK_shipment PRIMARY KEY (ShipmentID),
    CONSTRAINT FK_shipment_orderheader FOREIGN KEY (OrderID)
        REFERENCES orderheader(OrderID)
    );
```

Таким образом мы обеспечиваем заполнение всех значений для всех столбцов — именно этого мы и добивались. При попытке добавить строку без значения для какого-либо столбца система выдаст ошибку. Текст сообщения варьируется в зависимости от СУБД — MySQL обычно пишет, что для одного из столбцов отсутствует значение по умолчанию.

Как это понимать? И здесь мы подошли к рассмотрению еще одного класса ограничений — *значений по умолчанию* (default values).

19.1.2. Ограничения DEFAULT

Ограничения `DEFAULT` позволяют присвоить столбцу значение по умолчанию, которое будет подставлено, если при добавлении данных это поле оставить пустым. Значения по умолчанию, установленные этими ограничениями, применяются автоматически всякий раз, когда в инструкции `INSERT` не указано значение для столбца с ограничением `DEFAULT`.

Ограничение `DEFAULT` должно быть *литеральной константой* (literal constant), то есть выражением, идентичным для каждой новой строки. *Выражение* (expression) — это сочетание значений, операторов или функций, результатом вычисления которого является другое значение. Таким выражением может быть число, дата или строка символов. Впрочем, в MySQL и большинстве других реляционных СУБД в качестве значений по умолчанию можно также задействовать ряд функций даты и времени.

Наша таблица `shipment` может воспользоваться преимуществами такого ограничения с одной из этих функций. В главе 14 мы познакомились с функцией `CURRENT_DATE()`, которая возвращает текущую дату и время. Имеет смысл применить ее для настройки автоматического заполнения столбца `ShipmentDate`, чтобы при каждой вставке новой строки в таблицу `shipment` в него подставлялось актуальное значение `CURRENT_DATE()`.

> **ВНИМАНИЕ** Несмотря на широкое распространение, функция `CURRENT_DATE()` поддерживается не всеми СУБД. Вместо нее в SQL Server следует употреблять `GETDATE()`, в Oracle — `SYSDATE`, а в SQLite — `date('now')`.

Для ограничения `DEFAULT` после объявления типа данных мы задействуем ключевое слово `DEFAULT`, а затем укажем само значение по умолчанию. Вот как будет выглядеть наша команда `CREATE TABLE` с этим новым значением по умолчанию для `ShipmentDate`:

```
CREATE TABLE shipment (
    ShipmentID int NOT NULL,
    OrderID int NOT NULL,
    ShipmentCost decimal(5,2) NOT NULL,
    ShipmentMethod char(1) NOT NULL,
    TrackingNumber varchar(20) NOT NULL,
    ShipmentDate datetime NOT NULL DEFAULT (CURRENT_DATE()),
    CONSTRAINT PK_shipment PRIMARY KEY (ShipmentID),
    CONSTRAINT FK_shipment_orderheader FOREIGN KEY (OrderID)
        REFERENCES orderheader(OrderID)
    );
```

В связи с новым ограничением отметим два ключевых аспекта:

- *В команде* `CREATE TABLE` *значение по умолчанию* `CURRENT_DATE()` *необходимо заключать в круглые скобки*. Большинство других СУБД обходятся без них, а вот в MySQL скобки обязательны. Во избежание синтаксической ошибки сверьтесь с документацией вашей системы.
- *К одному столбцу можно применить несколько ограничений одновременно — как и в случае с ShipmentDate*. Разделитель в виде запятой между ограничениями не ставится, и не стоит пытаться это делать, поскольку в таком случае запятая обозначала бы новый столбец, а не второе ограничение для столбца `ShipmentDate`. В итоге этот столбец имеет ограничение `NOT NULL` и ограничение `DEFAULT`, что отнюдь не редкость для столбцов, автоматически указывающих время добавления строки.

> **НА ЗАМЕТКУ** Даже при наличии ограничения `DEFAULT` ограничение `NOT NULL` для столбца все равно необходимо. `DEFAULT` гарантирует подстановку значения, если оно не задано, тогда как `NOT NULL` страхует от явного указания NULL в качестве значения при вставке новой строки.

19.1.3. Ограничения UNIQUE

Рассмотрим далее еще один тип ограничения, применяемый к другой разновидности столбцов, — ограничение UNIQUE. Такое ограничение требует, чтобы каждое значение в заданном столбце было уникальным и не повторялось. Если при попытке вставить или изменить значение в столбце, где задано ограничение UNIQUE, в столбце окажутся одинаковые значения, произойдет ошибка.

Ограничения UNIQUE чем-то похожи на ограничение PRIMARY KEY, с которым мы уже работали, главное же их отличие состоит в том, что таблица может содержать только одно ограничение PRIMARY KEY, а вот ограничений UNIQUE может быть сколько угодно — для каждого столбца, где важно избежать повторов.

В отличие от PRIMARY KEY, ограничения UNIQUE могут включать значения NULL. Максимальное количество NULL-значений в столбце с ограничением UNIQUE зависит от управляющей СУБД: например, MySQL допускает несколько пустых значений, тогда как SQL Server и Oracle — только одно. Если для вашего проекта это ограничение имеет значение, обратитесь к документации вашей системы. Однако на практике случаи, когда столбцу требуется одновременно ограничение UNIQUE и возможность содержать NULL, встречаются довольно редко.

Поскольку номера отслеживания отправлений у каждой компании-перевозчика всегда уникальны, имеет смысл создать ограничение UNIQUE для столбца TrackingNumber, чтобы не допустить в нем повторяющихся значений. Добавить его так же просто, как и другие ограничения, рассмотренные в этой главе: достаточно указать слово UNIQUE в определении столбца:

```
CREATE TABLE shipment (
    ShipmentID int NOT NULL,
    OrderID int NOT NULL,
    ShipmentCost decimal(5,2) NOT NULL,
    ShipmentMethod char(1) NOT NULL,
    TrackingNumber varchar(20) NOT NULL UNIQUE,
    ShipmentDate datetime NOT NULL DEFAULT (CURRENT_DATE()),
    CONSTRAINT PK_shipment PRIMARY KEY (ShipmentID),
    CONSTRAINT FK_shipment_orderheader FOREIGN KEY (OrderID)
        REFERENCES orderheader(OrderID)
    );
```

19.1.4. Ограничения CHECK

Задействуем в нашей таблице последний тип ограничений — CHECK. Ограничение CHECK позволяет нам сужать круг допустимых в столбце значений путем их сравнения с заданным выражением. Возможность применения выражений в ограничении CHECK обеспечивает значительную гибкость при проверке корректности значений столбцов.

В таблице `shipment` следует добавить ограничения `CHECK` для столбцов `ShipmentCost` и `ShipmentMethod`. Способ доставки (ShipmentMethod) допускает лишь значения «P» и «E», поэтому выражение для нашего ограничения будет таким: `ShipmentMethod IN ('P', 'E')`.

> **ВНИМАНИЕ** Выражения, применяемые в ограничениях `CHECK`, могут включать сразу несколько столбцов, поэтому следует явно указывать столбец (или столбцы), участвующие в данных выражениях.

Для ограничения `CHECK` столбца `ShipmentCost` требуется задать значение в диапазоне от 0.00 до 999.99, следовательно, выражение будет иметь вид `ShipmentCost BETWEEN 0.00 AND 999.99`. Напомню, что оператор `BETWEEN` включает граничные значения, поэтому 0.00 и 999.99 также являются допустимыми. Ограничения `CHECK` добавляются аналогично ограничению `DEFAULT`, с помещением выражения ограничения в круглые скобки:

```
CREATE TABLE shipment (
    ShipmentID int NOT NULL,
    OrderID int NOT NULL,
    ShipmentCost decimal(5,2) NOT NULL
        CHECK (ShipmentCost BETWEEN 0.00 AND 999.99),
    ShipmentMethod char(1) NOT NULL CHECK (ShipmentMethod IN ('P', 'E')),
    TrackingNumber varchar(20) NOT NULL UNIQUE,
    ShipmentDate datetime NOT NULL DEFAULT (CURRENT_DATE()),
    CONSTRAINT PK_shipment PRIMARY KEY (ShipmentID),
    CONSTRAINT FK_shipment_orderheader FOREIGN KEY (OrderID)
        REFERENCES orderheader(OrderID)
    );
```

Несмотря на относительную простоту созданных нами ограничений `CHECK` для таблицы `shipment`, такие ограничения, как отмечалось ранее, могут охватывать несколько столбцов. Мы могли бы написать единое ограничение с составным выражением, проверяющим, находится ли `ShipmentCost` в указанном диапазоне числовых значений и имеет ли `ShipmentMethod` одно из двух допустимых значений. При использовании в выражении ограничения нескольких столбцов принято размещать такое ограничение после объявления всех столбцов, как это делалось при указании первичных и внешних ключей.

19.2. Автоматическое приращение значений в столбце

Нам осталось внести еще одно изменение в таблицу — это довольно типичная доработка, чем-то напоминающая ограничение `DEFAULT`. Мы уже решили, что столбец `ShipmentID` будет служить первичным ключом в нашей новой таблице `shipment`. Следовательно, его значения должны быть уникальными, чтобы мы могли однозначно идентифицировать каждую строку этой таблицы.

В главе 18 мы вручную прописывали значения первичного ключа (`CategoryID`) для таблицы `category`. Однако в большинстве случаев нежелательно самостоятельно задавать значения для столбца, который определен как первичный ключ. Вместо этого следует воспользоваться функцией, присутствующей во всех СУБД: функцией автоинкремента, или автоматического приращения значений.

В MySQL эта функция реализуется посредством ключевого слова `AUTO_INCREMENT`. Строго говоря, автоинкремент не является ограничением, однако он ведет себя подобно ограничению `DEFAULT`, позволяя вставлять строки в таблицу без необходимости явного указания значения для столбца.

При вставке строк оператором `INSERT` в таблицу `shipment` с включенным `AUTO_INCREMENT` мы пропускаем значения для столбца `ShipmentID`. Первая вставленная таким образом строка присваивает столбцу `ShipmentID` значение `1`, вторая — `2` и т. д. Автоматическое заполнение столбца увеличивающимися значениями обеспечивает уникальность и обязательное наличие (не NULL) первичного ключа.

Добавим `AUTO_INCREMENT` в наш SQL-запрос тем же способом, каким мы объявляли ограничения. Вот так выглядит теперь наша инструкция `CREATE TABLE` со столбцом `ShipmentID`, для которого установлен `AUTO_INCREMENT`:

```
CREATE TABLE shipment (
    ShipmentID int NOT NULL AUTO_INCREMENT,
    OrderID int NOT NULL,
    ShipmentCost decimal(5,2) NOT NULL
        CHECK (ShipmentCost BETWEEN 0.00 AND 999.99),
    ShipmentMethod char(1) NOT NULL CHECK (ShipmentMethod IN ('P', 'E')),
    TrackingNumber varchar(20) NOT NULL UNIQUE,
    ShipmentDate datetime NOT NULL DEFAULT (CURRENT_DATE()),
    CONSTRAINT PK_shipment PRIMARY KEY (ShipmentID),
    CONSTRAINT FK_shipment_orderheader FOREIGN KEY (OrderID)
        REFERENCES orderheader(OrderID)
    );
```

Таблица `shipment` со всеми заданными ограничениями готова — можно создавать!

К практике!

При помощи приведенного SQL-запроса создайте таблицу shipment. Вы поработаете с ней в упражнениях к этой главе.

19.3. Индексы

Каждое созданное нами ограничение предназначено для обеспечения целостности данных. Однако в MySQL некоторые из этих ограничений попутно генерируют объекты, которые мы пока не рассматривали, — индексы. Индексы

реализованы в любой современной реляционной СУБД, поэтому стоит присмотреться к ним повнимательнее.

Индекс (index) представляет собой объект, осуществляющий логическое упорядочивание данных для оптимизации выполнения распространенных запросов, тем самым ускоряя получение результатов. Существует два типа индексов: кластеризованные и некластеризованные. Начнем с первых.

19.3.1. Кластеризованные индексы

Кластеризованные индексы (clustered index) — это структуры данных, которые определяют физический порядок строк в таблице. Иными словами, этот порядок соответствует тому, как данные хранятся на дисках или других носителях информации. Если таблица создается без явно заданных индексов или ограничений, строки данных сохраняются в произвольном порядке. В силу данного обстоятельства любой запрос к такой таблице вынужден просматривать каждую строку, чтобы установить, нужно ли извлекать содержащиеся в ней значения, фильтровать их, объединять и т. п.

Чтобы было проще понять, что такое кластеризованный индекс, представьте себе телефонный справочник с именами и номерами телефонов жителей вашего города. Такой справочник обычно отсортирован по фамилии, а затем по имени. На каждой строке располагаются имя, фамилия, номер телефона и, возможно, адрес. Если бы этот справочник был таблицей, то кластеризованный индекс был бы построен по полям «фамилия» и «имя», потому что именно в этом порядке организованы строки справочника.

Нам нужно упорядочить данные по самым востребованным столбцам таблицы, поэтому к созданию кластеризованного индекса стоит подойти обдуманно. Поскольку строки в таблице можно отсортировать лишь одним способом, важно понимать, что любая таблица может содержать только один такой индекс. После его создания кластеризованный индекс фактически становится самой таблицей, а не отдельным объектом.

Если вы выполнили SQL-запрос из раздела 19.2, вы уже создали кластеризованный индекс для таблицы `shipment`, поскольку MySQL сгенерировал его автоматически при создании ограничения `PRIMARY KEY`. Обычно это оптимальный вариант, поскольку первичный ключ чаще всего и становится основой для кластеризации.

Как вы могли заметить, соединение таблиц в запросах выполнялось по связанным первичным и внешним ключам. Поскольку СУБД необходимо считывать эти значения ключей для соединения строк из разных таблиц, имеет смысл создавать кластеризованные индексы на столбцах первичных ключей.

НА ЗАМЕТКУ Даже если вы не задаете в таблице первичный ключ, сама СУБД «за кулисами» создает скрытый столбец, который часто называют *идентификатором строки* (row identifier). Столбец этот содержит уникальное значение для каждой строки. Будь у вас две или более строк с одинаковыми значениями, идентификатор строки позволил бы системе их различить. Но так как этот столбец скрыт, пользователи обычно его не видят. Поэтому, когда таблице требуется наличие уникального значения для каждой строки, принято явно создавать ограничение `PRIMARY KEY` и кластеризованный индекс.

Как отмечалось выше, в нашей таблице у нас уже есть кластеризованный индекс. В среде MySQL индексы таблицы можно просмотреть в MySQL Workbench: в панели навигации раскрываем сначала базу данных sqlnovel, затем Tables (Таблицы), таблицу shipment и, наконец, раздел Indexes (Индексы). Как показано на рис. 19.1, таблица shipment содержит три индекса.

Можно получить еще больше сведений о каждом индексе, выделив его и взглянув на информационную панель (Information), обычно расположенную под панелью навигации. Так, при выборе индекса с именем PRIMARY отображаются сведения об индексе (рис. 19.2).

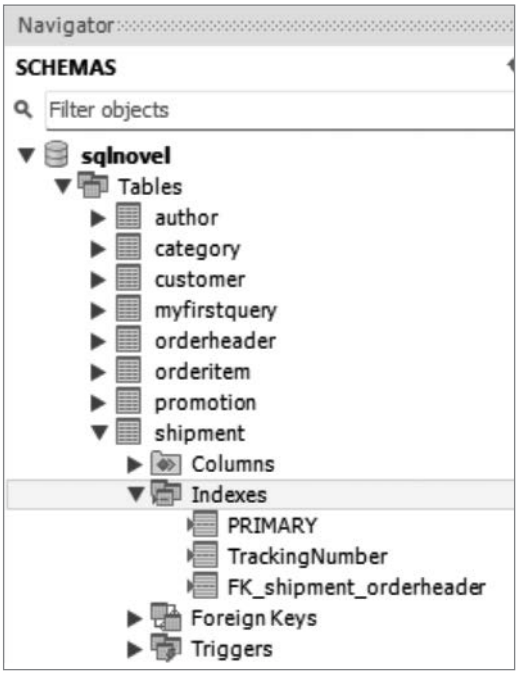


Рис. 19.1. Три индекса таблицы shipment в базе данных sqlnovel — вид в MySQL Workbench

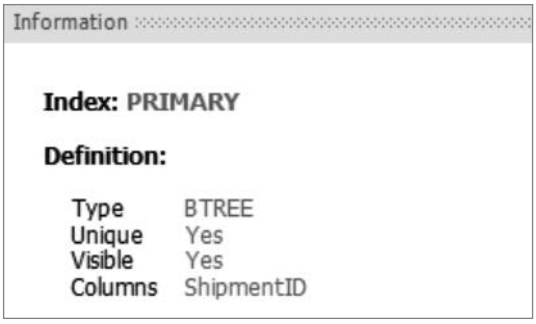


Рис. 19.2. Информация об индексе PRIMARY таблицы shipment — вид в MySQL Workbench

Хотя явное указание на кластеризованный индекс здесь отсутствует, в MySQL индекс с меткой `PRIMARY` является кластеризованным. Согласен, информационная панель содержит не так уж много информации, но для нас главное — значение в поле `Columns` (Столбцы). Из него мы видим, что кластеризованный индекс построен на столбце `ShipmentID` — том самом, для которого задано ограничение `PRIMARY KEY`.

> **ВНИМАНИЕ** Поскольку кластеризованные индексы не являются отдельными объектами, а скорее представляют собой свойства таблицы, зачастую связанные с ее первичным ключом, каждая система управления реляционными базами данных создает их по-своему. В некоторых СУБД, таких как SQL Server и DB2, кластеризованные индексы можно создать вручную при помощи операторов `ALTER TABLE` или `CREATE INDEX`. В других же системах, к примеру MySQL и PostgreSQL, такой возможности нет: система сама решает, какой индекс будет кластеризованным. Для выяснения доступных механизмов формирования кластеризованных индексов рекомендуется обратиться к официальной документации конкретной СУБД.

19.3.2. Некластеризованные индексы

Несмотря на распространенность и высокую эффективность кластеризованных индексов в части повышения производительности, некластеризованные индексы также позволяют существенно ускорить выполнение запросов. Некластеризованный индекс имеет два основных отличия от кластеризованного:

- В отличие от кластеризованных индексов, некластеризованные являются автономными объектами, отделенными от связанных с ними таблиц.
- Поскольку некластеризованные индексы являются отдельными объектами, таблица может содержать несколько таких индексов.

Хорошей аналогией для некластеризованного индекса может служить библиотечный каталог. В библиотеке нон-фикшен-литература обычно выстроена по

числовой системе классификации, например, по Десятичной классификации Дьюи. Однако большинство из нас ищет книгу не по ее числовому индексу, а по заглавию. Мы обращаемся к каталогу, чтобы найти нужное нам название, а каталог сообщает числовое значение (индекс) соответствующей книги. Затем мы переходим к стеллажам, где издания расположены по числовой системе, и находим требуемый экземпляр.

Некластеризованные индексы работают по тому же принципу, что и библиотечный каталог. Это своего рода способ упорядочения книг, в нашем случае по названию, который позволяет оперативно найти соответствие между названием и числовым значением, а затем применить это значение, чтобы перейти к месту фактического хранения книги. В этой аналогии числовое значение соответствует первичному ключу и кластерному индексу для нон-фикшен-литературы в библиотеке.

Как следует из приведенной аналогии, каталожная система (некластеризованный индекс) значительно ускоряет поиск конкретной книги. Если бы каталога не существовало, нам пришлось бы просматривать всю библиотеку, пока мы не нашли бы требуемое издание. Некластеризованные индексы создаются по той же причине — чтобы повысить производительность поиска строк данных в таблице без необходимости считывать ее целиком.

Хотя мы можем включить в некластеризованный индекс любые столбцы, которые сочтем нужными, на практике обычно индексируется лишь несколько столбцов, а порой всего один. Чаще всего поиск выполняется лишь по одному-двум полям, поэтому нет смысла делать некластеризованный индекс больше, чем необходимо. Чем больше столбцов он содержит, тем больше требуется места для его хранения и тем больше вычислительных ресурсов расходуется при выполнении команд `INSERT`, `UPDATE` и `DELETE`, связанных с таблицей.

Как видно из рис. 19.1, в нашей таблице `shipment` имеется два некластеризованных индекса, так что давайте их изучим. Для этого можно щелкнуть на них мышью в панели навигации и просмотреть сведения о них на информационной панели. Взглянем, к примеру, на данные об индексе `TrackingNumber`, представленные на рис. 19.3.

Поскольку мы добавили ограничение `UNIQUE` для столбца `TrackingNumber`, MySQL автоматически построил некластеризованный индекс для этого столбца. Как продемонстрировано на рис. 19.3, этот индекс имеет значение Yes для свойства Unique.

В этом плане MySQL выгодно отличается от многих других СУБД автоматическим созданием индекса для данного столбца. Поскольку столбец `TrackingNumber` должен содержать уникальные значения, создание для него некластеризованного индекса в любом случае представляется целесообразным.

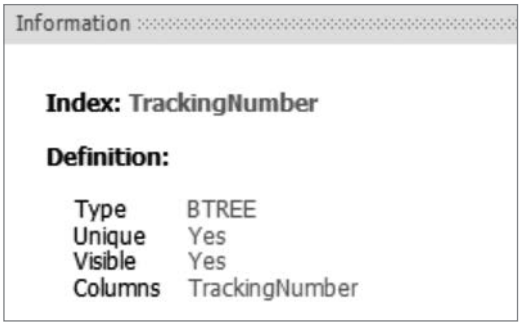


Рис. 19.3. Информация об индексе TrackingNumber таблицы shipment — вид в MySQL Workbench

Логика подсказывает: по столбцу `TrackingNumber` с номерами для отслеживания отправлений чаще всего будут искать информацию о конкретных посылках. Этот индекс и создавался для того, чтобы каждый раз не перебирать всю таблицу `shipment`, когда нужна информация по определенному трек-номеру. Иными словами, такие поля — идеальные кандидаты для некластеризованной индексации.

Аналогия, приведенная ранее, описывает библиотечный каталог, который работает как своего рода некластеризованный индекс для всех нон-фикшен-публикаций; здесь же столбец `TrackingNumber` будет выполнять такую же функцию применительно к номерам отслеживания наших отправлений. Если бы индекс для `TrackingNumber` не был сгенерирован автоматически (а во многих других СУБД этого не происходит), мы могли бы создать его при помощи следующей SQL-команды:

```
CREATE INDEX IX_shipment_TrackingNumber ON shipment (TrackingNumber);
```

Этот синтаксис универсален и работает практически в любой СУБД, так что никаких уточнений относительно его применения не требуется.

> **СОВЕТ** В представленной выше SQL-команде, создающей некластеризованный индекс для таблицы `shipment`, применяется общепринятая, но достаточно специфичная схема именования: `IX` для обозначения индекса, затем символ подчеркивания, имя таблицы, еще одно подчеркивание и имя индексируемого столбца. Помните, что следует целенаправленно подходить к выбору имен объектов БД и придерживаться единой системы именования, чтобы другим было проще ориентироваться в вашем коде.

Как видно на рис. 19.1, в таблице присутствует третий индекс: `FK_shipment_orderheader`. Он сгенерирован нашим ограничением `FOREIGN KEY`, что является особенностью MySQL, но не обязательно других СУБД. Впрочем, создание некластеризованных индексов для столбцов, входящих в ограничения `FOREIGN KEY`, является обычной практикой, поскольку по этим столбцам происходит связывание таблиц через отношения ключей. Такие индексы избавляют от

необходимости всякий раз полностью считывать эти таблицы при соединениях. Свойства индекса можно просмотреть на информационной панели, нажав на FK_shipment_orderheader в панели навигации (рис. 19.4).

Следует отметить одну характерную черту данного индекса: значение свойства Unique равно No, что отличает его от двух других индексов таблицы `shipment`. Несмотря на то что ограничение `UNIQUE` создало индекс с требованием неповторимости значений, важно понимать, что некластеризованные индексы могут допускать дубликаты. Для индекса `FK_shipment_orderheader` уникальные значения не требуются, поскольку между таблицами `orderheader` и `shipment` в контексте значений `OrderID` может существовать отношение «один ко многим».



Рис. 19.4. Информация об индексе FK_shipment_orderheader таблицы shipment — вид в MySQL Workbench

В этой главе мы рассмотрели многочисленные аспекты проектирования баз данных. Выделим ключевые моменты, связанные с ограничениями и индексами:

- Ограничения — это свойства одного или нескольких столбцов, которые обеспечивают целостность данных.
- Ограничения `NOT NULL` гарантируют отсутствие пустых значений в столбце.
- Ограничения `DEFAULT` задают значение по умолчанию, если в `INSERT` для столбца не указано явное значение.
- Ограничения `UNIQUE` обеспечивают отсутствие повторяющихся значений в столбце.
- Ограничения `CHECK` применяются для ограничения множества допустимых значений в столбце.
- Кластеризованный индекс определяет физический порядок сортировки таблицы, обычно по первичному ключу.
- У таблицы может быть только один кластеризованный индекс.
- Некластеризованный индекс является отдельным от таблицы объектом.

- Таблица может иметь несколько некластеризованных индексов, однако каждый такой индекс снижает производительность команд `INSERT`, `UPDATE` и `DELETE`.
- В MySQL (но не во всех СУБД) кластеризованный индекс создается автоматически при определении ограничения `PRIMARY KEY`.
- В MySQL (но не во всех СУБД) некластеризованный индекс создается автоматически при определении каждого ограничения `UNIQUE` или `FOREIGN KEY`.

Если есть силы и желание — самое время закрепить полученные навыки работы с ограничениями и индексами в практических упражнениях.

19.4. Практическое занятие

1. Напишите SQL-запрос для вставки строк в новую таблицу `shipment`, воспользовавшись следующими значениями:
   - `OrderID = 1001`
   - `ShipmentCost = 0.00`
   - `ShipmentMethod = 'P'`
   - `TrackingNumber = '1A2C3M4E'`
2. Учитывая, что таблица `shipment` в настоящий момент содержит не все заказы, каким образом следует составить запрос для отображения `OrderID`, `OrderDate` и `ShipmentDate` для каждого заказа?
3. Могли бы вы переписать выражение в разделе 19.1.4? Если да, то как?
4. Требуется создать отчет, отображающий общее число всех заказов, отправленных в определенную дату. Какое ограничение или индекс следует создать для оптимизации производительности данного отчета?

19.5. Ответы

1. Вам не нужно указывать значение для `ShipmentID`, так как это столбец с `AUTO_INCREMENT`, а также не нужно указывать значение для `ShipmentDate`, поскольку для него задано ограничение `DEFAULT`. Следовательно, ваш SQL-запрос будет выглядеть примерно так:

```
INSERT shipment (
    OrderId,
    ShipmentCost,
    ShipmentMethod,
    TrackingNumber
```

```
    )
VALUES (
    1001,
    0.00,
    'P',
    '1A2C3M4E'
    );
```

2. Поскольку у нас есть значения для каждого `OrderID` в таблице `orderheader`, но нет значений для каждого `OrderID` в таблице `shipment`, требуется применение левого внешнего соединения (`LEFT OUTER JOIN`), как показано ниже:

```
SELECT
    oh.OrderID,
    oh.OrderDate,
    s.ShipmentDate
FROM orderheader oh
LEFT OUTER JOIN shipment s
    ON oh.OrderID = s.OrderID;
```

   При использовании внутреннего соединения (`INNER JOIN`) в выборку попали бы только те заказы, которые имеют значение в столбцах `OrderID` обеих таблиц.

3. Это выражение можно переписать несколькими способами, включая следующий:

```
ShipmentCost >= 0.00 AND ShipmentCost <= 999.99.
```

4. SQL-код для отчета может иметь следующий вид:

```
SELECT
    ShipmentDate,
    COUNT(ShipmentDate)
FROM shipment
WHERE ShipmentDate = @ShipmentDate
GROUP BY ShipmentDate;
```

   Для оптимизации выполнения данного запроса можно создать некластеризованный индекс по столбцу `ShipmentDate`:

```
CREATE INDEX IX_shipment_ShipmentDate ON shipment (ShipmentDate);
```

   Этот индекс избавит от необходимости считывания всей таблицы при определении общего числа заказов, когда нужно найти лишь небольшую часть заказов, отправленных в конкретный день.

20 Повторное использование запросов: представления и хранимые процедуры

За 19 предыдущих глав мы написали множество SQL-запросов. Мы применяли фильтры, функции, агрегирование и другие приемы для поиска необходимых данных. Мы также научились добавлять, редактировать и удалять данные, а при помощи переменных добивались того, чтобы наши запросы могли выполнять одни и те же действия с разными значениями.

Теперь пришло время собрать все эти навыки воедино. В этой главе мы перейдем от запуска SQL-запросов к их сохранению в виде объектов базы данных — сценариев, которые может выполнять любой пользователь с соответствующими правами. В зависимости от выбранной реляционной СУБД существует несколько типов объектов, предназначенных для хранения подобного рода сценариев. Здесь мы сосредоточимся на двух наиболее универсальных объектах такого рода: представлениях и хранимых процедурах.

Представление (view) хранит команду `SELECT` и возвращает единый набор результатов, с которым можно работать так же, как с таблицей. *Хранимая процедура* (stored procedure) — это объект базы данных, содержащий один либо несколько SQL-запросов, которые могут выполняться совместно для реализации практически любой требуемой задачи в рамках системы.

20.1. Представления

Представления (views) — это логические объекты базы данных, формируемые на основе команды `SELECT`. Каждое представление возвращает единый результирующий набор данных, похожий на таблицу, поэтому их нередко называют *виртуальными таблицами* (virtual tables). Термин «виртуальные таблицы»

подчеркивает тот факт, что с представлениями в SQL-запросах можно работать так же, как с обычными таблицами.

Однако называть представления *виртуальными таблицами* не вполне корректно, поскольку они не являются таблицами и не содержат данных. Наверное, правильнее будет считать их командами SELECT, которым присвоено имя, хотя и такое определение не полностью отражает их функциональные возможности.

Представления позволяют многократно применять один и тот же запрос, а также свести сложный запрос к простому и интуитивно понятному объекту. При этом администратор БД может назначать пользователям права доступа к конкретным представлениям.

20.1.1. Создание представлений

Любое представление формируется на основе команды SELECT. Для создания представления, отображающего заглавия книг и названия их категорий, можно задействовать следующий запрос (результаты продемонстрированы на рис. 20.1):

```
SELECT
    t.TitleName,
    c.CategoryName
FROM title t
INNER JOIN category c
    ON t.CategoryID = c.CategoryID;
```

TitleName	CategoryName
Pride and Predicates	Romance
The Join Luck Club	Mystery
Catcher in the Try	Mystery
Anne of Fact Tables	Romance
The DateTime Machine	Science Fiction
The Great GroupBy	Humor
The Call of the While	Fantasy
The Sum Also Rises	Science Fiction
David Emptyfield	Humor
Red Badge of Cursors	Mystery
Of Mice and Metadata	Science Fiction
A Table of Two Cities	Fantasy

Рис. 20.1. Все значения TitleName из таблицы title и связанные значения CategoryName из таблицы category

Для создания представления на основе указанного запроса необходимо сформировать SQL-команду в следующей последовательности:

1. CREATE VIEW.
2. Имя представления.

3. AS.
4. Команда SELECT.

При помощи столь несложного синтаксиса можно создать представление с именем vw_TitleCategory:

```
CREATE VIEW vw_TitleCategory
AS
SELECT
    t.TitleName,
    c.CategoryName
FROM title t
INNER JOIN category c
    ON t.CategoryID = c.CategoryID;
```

Созданное нами представление сохраняет нашу команду SELECT, и теперь она может быть выполнена в любое время. Чтобы увидеть результаты работы представления, мы выбираем из него данные так же, как если бы это была обычная таблица:

```
SELECT *
FROM vw_TitleCategory;
```

Результаты выполнения предыдущего запроса, приведенные на рис. 20.1, полностью совпадают с результатами выполнения нашей исходной SQL-команды.

Для часто применяемых запросов представления экономят время — нужная выборка данных всегда под рукой и готова к работе. Напомню, что само по себе представление не содержит данных — оно лишь вызывает их при выполнении лежащего в его основе SQL-запроса. Однако этим возможности представлений не ограничиваются.

20.1.2. Фильтрация представлений

Представления можно фильтровать так же, как и обычные таблицы, прибегнув к предложению WHERE, что мы не раз делали в других запросах. Допустим, если мы хотим видеть в нашем представлении только произведения из категории Mystery (Детектив), как показано на рис. 20.2, достаточно отредактировать наш SELECT из представления, добавив соответствующее условие фильтра в предложение WHERE:

```
SELECT *
FROM vw_TitleCategory
WHERE CategoryName = 'Mystery';
```

С представлениями можно выполнять почти все операции, применимые к обычным таблицам, — фильтровать результаты, упорядочивать их и агрегировать данные. Допустимо даже соединять представления с другими таблицами и представлениями, но для этого потребуется определенная модификация нашего представления.

TitleName	CategoryName
The Join Luck Club	Mystery
Catcher in the Try	Mystery
Red Badge of Cursors	Mystery

Рис. 20.2. Выборка всех строк из представления vw_TitleCategory со значением CategoryName, равным 'Mystery'

20.1.3. Соединение представлений

Из многочисленных примеров соединения таблиц вы помните, что для успешного соединения таблиц необходимо установить реляционные связи. Представление `vw_TitleCategory` содержит два столбца, но ни один из них не связан с ключевыми значениями, формирующими отношения с другими таблицами. Так, ни одна таблица в БД не имеет отношений, задействующих столбцы `TitleName` или `CategoryName` в качестве ключей.

Для соединения нашего представления с другими таблицами требуются ключевые значения из исходных таблиц. В таком случае это означает добавление `TitleID` из таблицы `title` и `CategoryID` из таблицы `category`. Мы могли бы добавить `CategoryID` из таблицы `title` вместо таблицы `category`, однако в представлениях рекомендуется по возможности применять первичные, а не внешние ключи.

> **ВНИМАНИЕ** Как в таблицах, так и в представлениях имена столбцов должны быть уникальными. Если вы попытаетесь создать представление с одинаковыми именами столбцов, большинство СУБД, включая MySQL, вернут сообщение об ошибке, уведомляющее о наличии дубликатов в именах столбцов.

Модифицируем наше представление, прибегнув к синтаксису, похожему на применявшийся при его создании, однако теперь воспользуемся ключевым словом `ALTER` вместо `CREATE`, как это делалось при изменении таблицы в главе 18:

1. `ALTER VIEW`.
2. Имя представления.
3. `AS`.
4. Модифицированный SQL-запрос.

С помощью такой конструкции мы можем отредактировать наше представление `vw_TitleCategory`, включив в него два дополнительных столбца:

```
ALTER VIEW vw_TitleCategory
AS
SELECT
    t.TitleID,
    t.TitleName,
    c.CategoryID,
```

```
    c.CategoryName
 FROM title t
INNER JOIN category c
    ON t.CategoryID = c.CategoryID;
```

К практике!

Если вы еще не создавали и не изменяли представление `vw_TitleCategory`, просто выполните предыдущий запрос, заменив в нем `ALTER VIEW` на `CREATE VIEW`, — и оно будет создано. Это представление пригодится нам по ходу изучения главы.

После выполнения команды `ALTER VIEW` ознакомимся с результатами нашего представления при помощи следующего запроса (результаты приведены на рис. 20.3):

```
SELECT *
FROM vw_TitleCategory;
```

TitleID	TitleName	CategoryID	CategoryName
101	Pride and Predicates	1	Romance
102	The Join Luck Club	3	Mystery
103	Catcher in the Try	3	Mystery
104	Anne of Fact Tables	1	Romance
105	The DateTime Machine	5	Science Fiction
106	The Great GroupBy	2	Humor
107	The Call of the While	4	Fantasy
108	The Sum Also Rises	5	Science Fiction
109	David Emptyfield	2	Humor
110	Red Badge of Cursors	3	Mystery
111	Of Mice and Metadata	5	Science Fiction
112	A Table of Two Cities	4	Fantasy

Рис. 20.3. Выборка всех строк и столбцов из представления vw_TitleCategory, включающая теперь столбцы TitleID и CategoryID

Заметьте, результаты не упорядочены. И это правильно, ведь сортировка в самом представлении, без особой на то надобности, лишь замедлит работу. Как и в команде `SELECT`, добавление предложения `ORDER BY` к представлению может значительно снизить производительность запроса при обработке миллионов строк данных.

Теперь наше представление можно не только связывать с другими таблицами, но и применять для расчетов. К примеру, чтобы узнать, сколько книг продано в каждой категории, достаточно соединить наше представление с таблицей `orderitem`, задействовав связь между столбцами `TitleID`. Запрос будет иметь следующий вид (результат показан на рис. 20.4):

```
SELECT
    tc.CategoryName,
    SUM(oi.Quantity) AS TitlesOrdered
FROM vw_TitleCategory tc
LEFT OUTER JOIN orderitem oi
    ON tc.TitleID = oi.TitleID
GROUP BY tc.CategoryName;
```

CategoryName	TitlesOrdered
Romance	27
Mystery	24
Science Fiction	16
Humor	1
Fantasy	2

Рис. 20.4. Выборка общего количества книг, заказанных в каждой категории

20.1.4. Что важно знать о представлениях

Представления — необычайно удобный инструмент, хотя и здесь есть свои правила и оговорки. Вот лишь некоторые из ключевых факторов, которые необходимо учитывать при создании представлений и работе с ними:

- Имя представления не должно совпадать с именем другой таблицы или представления. Помните об этом, присваивая имена представлениям.
- При присвоении имен представлениям желательно следовать соглашению об именовании, которое позволяет отличать их от таблиц. В нашем примере в имени представления применяется префикс `vw_`. Последовательное соблюдение такой схемы особенно важно в тех случаях, когда к созданным вами представлениям обращаются другие пользователи, — это помогает им легко отличать таблицы от представлений.
- Рекомендуется включение столбцов с первичными и внешними ключами в команду `SELECT`, используемую при создании представления. Такой подход позволяет связать результаты представления с другими таблицами и представлениями.
- В главе 11 мы рассматривали *подзапросы* — запросы, вложенные в другие запросы, — при помощи которых мы извлекали нужные данные. Аналогичным образом представления могут обращаться к другим представлениям посредством подзапросов или соединений (`JOIN`). Такие представления, применяемые внутри других представлений, называются *вложенными представлениями* (nested views). Однако прибегать к ним следует с осторожностью, поскольку они существенно снижают производительность запросов.
- Если в вашем представлении есть вычисляемый столбец, как в примере выше, всегда задавайте для него псевдоним (alias). Большинство реляционных СУБД требуют, чтобы у каждого столбца представления было определенное имя.

- Возможно, вас это удивит, но многие СУБД позволяют править или даже вставлять данные прямо в представлениях. Однако делать это следует с осторожностью, поскольку такие изменения могут затронуть данные сразу в нескольких таблицах. В целом, от подобной практики лучше воздержаться.

НА ЗАМЕТКУ В свете последнего пункта очевидно, что представления не являются подходящим инструментом для модификации данных в SQL — для этих целей гораздо лучше подойдут хранимые процедуры. Они также представляют собой более эффективное средство для выборки данных.

20.2. Хранимые процедуры

Как и представления, хранимые процедуры позволяют сохранять SQL-команду непосредственно в базе данных для последующего многократного применения. Мы также можем назначить пользователям права доступа, определяя, могут ли они выполнять хранимые процедуры. Но этим их возможности не ограничиваются: в отличие от представлений, хранимые процедуры могут объединять несколько запросов в один процесс и обмениваться значениями через переменные, что делает их мощным средством управления данными.

20.2.1. Создание хранимых процедур

Начнем с преобразования нашей SQL-команды из раздела 20.1.1 в хранимую процедуру. Базовый синтаксис создания хранимых процедур практически универсален для всех СУБД:

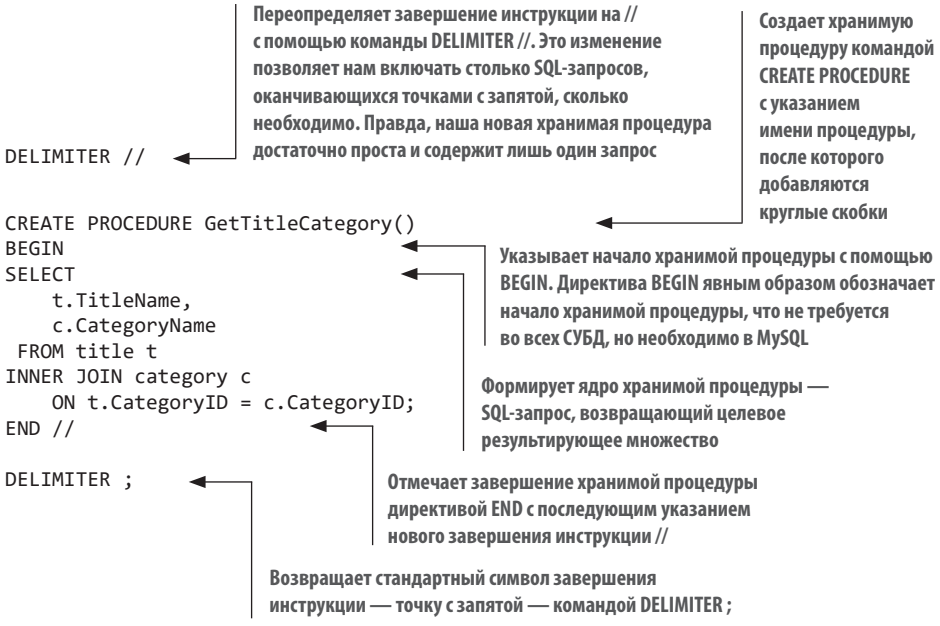
1. `CREATE PROCEDURE`.
2. Имя хранимой процедуры.
3. SQL-код, предназначенный для выполнения хранимой процедурой.

НА ЗАМЕТКУ К сожалению, в синтаксисе каждой СУБД при создании хранимых процедур обнаруживаются небольшие синтаксические различия. Но не стоит из-за этого отказываться от их изучения, поскольку, несмотря на специфику реализации, принцип работы с хранимыми процедурами одинаков во всех СУБД, за исключением SQLite, который вообще их не поддерживает.

Для создания хранимой процедуры в MySQL необходимо сначала изменить разделитель команд. Составляя наш первый запрос в главе 2, мы узнали, что в конце запроса следует ставить точку с запятой (*завершение команды*), указывающую СУБД, где заканчивается SQL-команда. Строгая политика MySQL в отношении

завершения команды создает определенные трудности при разработке хранимых процедур. Поскольку такие процедуры могут содержать несколько инструкций, первая встреченная точка с запятой будет воспринята системой как окончание всей процедуры.

В качестве обходного решения временно переопределим завершение команды, выбрав вместо точки с запятой двойную косую черту (//). Сделать это можно при помощи ключевого слова DELIMITER, характерного для MySQL. Далее создадим хранимую процедуру с применением стандартных точек с запятой внутри нее, а затем снова прибегнем к DELIMITER, чтобы вернуться к точке с запятой. Назовем процедуру GetTitleCategory. Ниже приведен SQL-код для создания хранимой процедуры, которая извлекает все значения TitleName и соответствующие значения CategoryName, с последующими пояснениями к его работе:



Созданная нами хранимая процедура готова к применению. Для ее выполнения используется ключевое слово CALL (результаты отображены на рис. 20.5):

```
CALL GetTitleCategory;
```

Пока что данная хранимая процедура довольно проста. Но возможности хранимых процедур гораздо шире — в разделе 20.2.2 мы добавим функциональность для передачи переменной и фильтрации наших результатов.

TitleName	CategoryName
Pride and Predicates	Romance
The Join Luck Club	Mystery
Catcher in the Try	Mystery
Anne of Fact Tables	Romance
The DateTime Machine	Science Fiction
The Great GroupBy	Humor
The Call of the While	Fantasy
The Sum Also Rises	Science Fiction
David Emptyfield	Humor
Red Badge of Cursors	Mystery
Of Mice and Metadata	Science Fiction
A Table of Two Cities	Fantasy

Рис. 20.5. Выборка всех значений TitleName из таблицы title и связанных с ними значений CategoryName из таблицы category, возвращенная хранимой процедурой GetTitleCategory

НА ЗАМЕТКУ Синтаксис хранимой процедуры зависит от конкретной СУБД, как и ее вызов. Так, MySQL, PostgreSQL и MariaDB задействуют ключевое слово `CALL`, а SQL Server и Oracle — `EXEC`.

20.2.2. Применение переменных в хранимых процедурах

Одним из главных преимуществ хранимых процедур по сравнению с представлениями является наличие параметров. *Параметр* (parameter) — это переменная, которая может передаваться в хранимую процедуру или из нее. В каждой процедуре таких параметров может быть несколько — и они открывают массу возможностей, среди которых фильтрация данных, изменение значений в таблицах или определение формата вывода результатов. При использовании параметров в хранимой процедуре для каждого из них необходимо задать три свойства:

- имя;
- тип данных;
- направление передачи (входной или выходной параметр).

Последнее свойство определяет, каким образом будет задействован параметр. Если параметр объявлен как входной, то в хранимую процедуру передается значение для использования внутри нее. Если же параметр объявлен как выходной, то его значение формируется во время выполнения процедуры и возвращается после ее завершения.

НА ЗАМЕТКУ В MySQL присутствует и третий тип параметров — `INOUT`. Он позволяет передать параметр внутрь, при необходимости изменить его, а затем вернуть. Этот тип параметров поддерживается не всеми СУБД.

Можно усовершенствовать нашу хранимую процедуру `GetTitleCategory`, добавив входной параметр для фильтрации по `TitleName`. Однако, чтобы изменить хранимую процедуру в MySQL, сначала требуется удалить ее, аналогично тому, как мы удаляли таблицы в главе 18:

```
DROP PROCEDURE GetTitleCategory;
```

Заново создадим нашу хранимую процедуру, теперь с входным параметром. Назовем параметр `_TitleName`, чтобы избежать путаницы с колонкой `TitleName`, а тип данных зададим таким же, как у колонки `TitleName` в таблице `title`. Узнать тип данных столбцов любой таблицы в MySQL можно при помощи оператора `SHOW COLUMNS`. Вот как задействовать `SHOW COLUMNS` для определения типов данных столбцов в таблице `title` (результаты показаны на рис. 20.6):

```
SHOW COLUMNS FROM title;
```

Field	Type	Null	Key	Default	Extra
TitleID	int	NO	PRI	NULL	
TitleName	varchar(50)	NO		NULL	
Price	decimal(5,2)	NO		NULL	
Advance	decimal(8,2)	NO		NULL	
Royalty	decimal(5,2)	YES		NULL	
PublicationDate	datetime	NO		NULL	
CategoryID	int	YES	MUL	NULL	

Рис. 20.6. Типы данных всех столбцов в таблице title, возвращенные командой SHOW COLUMNS

Как следует из результатов, тип данных столбца `TitleName` — `varchar(50)`, поэтому мы определим этот тип данных для нашего входного параметра. Осталось только добавить фильтрацию по этому параметру в самой процедуре с помощью условия `WHERE t.TitleName = _TitleName`. Следующий SQL-код создает нашу новую хранимую процедуру:

```
DROP PROCEDURE GetTitleCategory;

DELIMITER //

CREATE PROCEDURE GetTitleCategory(
    IN _TitleName varchar(50)
    )
BEGIN
SELECT
    t.TitleName,
```

```
    c.CategoryName
 FROM title t
INNER JOIN category c
    ON t.CategoryID = c.CategoryID
WHERE t.TitleName = _TitleName;
END //

DELIMITER ;
```

Можно теперь объявить переменную и передать ее в хранимую процедуру, чтобы вернуть только результаты для нужного значения `TitleName`. Выбрав в качестве такового «The Sum Also Rises», выполним хранимую процедуру с помощью следующего SQL-кода (результаты приведены на рис. 20.7):

```
SET @TitleName = 'The Sum Also Rises';
CALL GetTitleCategory (@TitleName);
```

TitleName	CategoryName
The Sum Also Rises	Science Fiction

Рис. 20.7. Результаты выполнения процедуры GetTitleCategory со значением «The Sum Also Rises», переданным через входной параметр _TitleName

После добавления входного параметра `_TitleName` в хранимую процедуру `GetTitleCategory` каждое ее выполнение требует указания значения для данного параметра. Если мы попытаемся выполнить `GetTitleCategory` без значения для `_TitleName`, то получим сообщение об ошибке Incorrect number of arguments («Неверное количество аргументов»). В контексте хранимой процедуры *аргумент* (argument) — это значение, передаваемое параметру. Теперь `GetTitleCategory` ожидает такой аргумент при каждом выполнении, и, если его не передать, процедура вместо одного получает ноль аргументов, а система закономерно выдает ошибку.

При написании хранимой процедуры нередко требуется предусмотреть вариант, когда аргумент может оказаться без значения. И вместо того чтобы вообще не передавать аргумент, в качестве значения можно просто указать NULL. Такой подход не является редкостью. Как мы не раз убеждались на протяжении всей книги, NULL — это тоже значение, с которым приходится считаться.

Одним из подходов к обработке ситуации, когда в параметр `_TitleName` передается значение NULL, является возврат всех строк результирующего набора. Как раз для таких случаев и пригодится функция `COALESCE` (о ней рассказывалось в главе 15). Чтобы хранимая процедура корректно обрабатывала NULL в параметре `_TitleName`, достаточно изменить условие фильтрации на `WHERE t.TitleName = COALESCE(_TitleName, t.TitleName)`.

Благодаря такой логике мы можем запускать хранимую процедуру с аргументом NULL. Если передано конкретное значение, наш результирующий набор

по-прежнему фильтруется по этому значению, а при передаче NULL возвращаются строки, в которых `TitleName` равен самому себе — то есть все строки.

Удалим хранимую процедуру и воссоздадим ее с новым алгоритмом фильтрации, применяющим `COALESCE`. Все эти операции можно объединить в одном SQL-коде:

```
DROP PROCEDURE GetTitleCategory;

DELIMITER //

CREATE PROCEDURE GetTitleCategory(
IN _TitleName varchar(50)
    )
BEGIN
SELECT
    t.TitleName,
    c.CategoryName
FROM title t
INNER JOIN category c
    ON t.CategoryID = c.CategoryID
WHERE t.TitleName = COALESCE(_TitleName, t.TitleName);
END //

DELIMITER ;
```

После такой доработки хранимую процедуру можно вызывать как с конкретным значением, так и с NULL. В последнем случае возвращаются все значения `TitleName`, как продемонстрировано на рис. 20.8:

```
SET @TitleName = NULL;
CALL GetTitleCategory (@TitleName);
```

TitleName	CategoryName
Pride and Predicates	Romance
The Join Luck Club	Mystery
Catcher in the Try	Mystery
Anne of Fact Tables	Romance
The DateTime Machine	Science Fiction
The Great GroupBy	Humor
The Call of the While	Fantasy
The Sum Also Rises	Science Fiction
David Emptyfield	Humor
Red Badge of Cursors	Mystery
Of Mice and Metadata	Science Fiction
A Table of Two Cities	Fantasy

Рис. 20.8. Выборка всех значений TitleName из таблицы title и связанных значений CategoryName из таблицы category, возвращенная хранимой процедурой GetTitleCategory с аргументом NULL для параметра _TitleName

При выполнении процедуры `GetTitleCategory` с аргументом, не равным NULL, таким как «The Sum Also Rises», мы получим отфильтрованные результаты только для указанного `TitleName`, как показано на рис. 20.9:

```
SET @TitleName = 'The Sum Also Rises';
CALL GetTitleCategory (@TitleName);
```

TitleName	CategoryName
The Sum Also Rises	Science Fiction

Рис. 20.9. Выборка всех значений TitleName из таблицы title и связанных значений CategoryName из таблицы category, возвращенная хранимой процедурой GetTitleCategory с аргументом 'The Sum Also Rises' для параметра _TitleName

К практике!

Создайте окончательную версию хранимой процедуры `GetTitleCategory` и протестируйте ее с различными значениями в качестве аргументов для `_TitleName`, включая NULL.

20.2.3. Что важно знать о хранимых процедурах

Надеюсь, теперь вы понимаете, почему хранимые процедуры являются таким популярным способом организации SQL-команды в реляционных СУБД. Однако, прежде чем вы броситесь преобразовывать все ваши запросы в хранимые процедуры, необходимо принять во внимание ряд существенных аспектов:

- Хранимые процедуры могут вызывать другие хранимые процедуры и даже передавать значения переменных между собой. Будьте осторожны с вложенностью хранимых процедур, так как это может затруднить отладку кода.
- Как и в случае с представлениями и другими объектами БД, придерживайтесь единой схемы именования в названиях хранимых процедур, чтобы ваши коллеги могли легко их идентифицировать и уяснить их назначение.
- При разработке хранимых процедур с параметрами необходимо следить, чтобы типы данных параметров соответствовали типам данных всех столбцов, к которым они будут применяться.
- Если вы задействуете переменные для передачи значений параметрам хранимой процедуры, удостоверьтесь, что тип данных ваших переменных совпадает с типом данных в хранимой процедуре.
- Поскольку хранимые процедуры могут содержать серию запросов, щедро снабжайте свой код понятными, содержательными комментариями, документирующими назначение каждого фрагмента вашей хранимой процедуры.

20.3. Различия между представлениями и хранимыми процедурами

В этой главе мы рассмотрели два наиболее популярных способа хранения SQL-кода для многократного применения. В частности, вы узнали, что представления и хранимые процедуры обладают различными наборами свойств. В табл. 20.1 обобщены ключевые различия между ними, что позволит вам выбрать подходящий инструмент в зависимости от конкретной ситуации или задачи.

Таблица 20.1. Ряд ключевых отличий между представлениями и хранимыми процедурами

Свойства	Представление	Хранимая процедура
Ввод	Не использует параметры	Может использовать параметры
Вывод	Может возвращать лишь один результирующий набор	Может возвращать ноль, один или несколько результирующих наборов или выходных параметров
Запросы	Может содержать лишь один запрос	Может содержать несколько запросов
Связи	Может соединяться с другими представлениями или таблицами посредством реляционных связей	Нельзя обращаться напрямую, как к таблице или представлению, — только через EXEC
Зависимости	Может содержать запрос, использующий таблицы или представления, но не хранимые процедуры	Может содержать запросы, использующие таблицы, представления или другие хранимые процедуры

Несмотря на то что эта глава освещает большинство аспектов, необходимых для работы с представлениями, она лишь приоткрывает дверь в мир хранимых процедур и их возможностей. В главе 21 вы узнаете, что можно создавать хранимые процедуры для чтения и записи данных с гибкой логикой, определяемой различными условиями.

20.4. Практическое занятие

1. Создайте представление с именем `vw_Order`, содержащее все столбцы из таблиц `orderheader` и `orderitem`, за исключением столбца `OrderID` из таблицы `orderitem`. Помните, что эти таблицы связаны через столбцы `OrderID`.
2. Почему, на ваш взгляд, следует исключить столбец `OrderID` из таблицы `orderitem`?

3. Составьте хранимую процедуру с именем `GetOrder`, соответствующую следующим требованиям:
   - задействует только что созданное вами представление `vw_Order`;
   - имеет параметр с именем `_OrderID` и отбирает результаты на основе соответствия значения этого параметра столбцу `OrderID` представления `vw_Order`;
   - соединяет с таблицей `title`, используя связь по столбцам `TitleID`;
   - возвращает следующие столбцы: `OrderID` (идентификатор заказа), `OrderDate` (дата заказа), `TitleName` (название книги), `Quantity` (количество) и `ItemPrice` (цена позиции).
4. Какой тип данных вы выбрали для параметра `_OrderID` в процедуре `GetOrder` и почему?
5. Что произойдет, если после создания процедуры `GetOrder` выполнить следующий код?

```
CALL GetOrder(1049)
```

20.5. Ответы

1. SQL-код вашего представления должен быть примерно таким:

```
CREATE VIEW vw_Order
AS
SELECT
    oh.OrderID,
    oh.CustomerID,
    oh.PromotionID,
    oh.OrderDate,
    oi.OrderItem,
    oi.TitleID,
    oi.Quantity,
    oi.ItemPrice
 FROM orderheader oh
INNER JOIN orderitem oi
    ON oh.OrderID = oi.OrderID;
```

2. Если не исключать столбец `OrderID` из таблицы `orderitem` в вашем представлении, получилось бы два столбца с именем `OrderID`. При попытке создать такое представление с двумя столбцами `OrderID` СУБД выдала бы ошибку, не понимая, какой столбец использовать.
3. SQL-код для создания хранимой процедуры `GetOrder` должен выглядеть приблизительно так:

```
DELIMITER //

CREATE PROCEDURE GetOrder(
    IN _OrderID int
    )
BEGIN
SELECT
    o.OrderID,
    o.OrderDate,
    t.TitleName,
    o.Quantity,
    o.ItemPrice
FROM vw_Order o
INNER JOIN title t
    ON o.TitleID = t.TitleID
WHERE o.OrderID = _OrderID;
END //

DELIMITER ;
```

4. Следовало использовать целочисленный тип данных (`int`) для параметра `_OrderID`, поскольку это тип данных столбца `OrderID`, с которым будет сравниваться параметр. Если вам неизвестен тип данных, можно определить его при помощи следующего запроса в MySQL:

```
SHOW COLUMNS FROM orderheader;
```

5. Команда вернет результаты, представленные на рис. 20.10. Выполнение хранимой процедуры таким способом демонстрирует, что при передаче аргументов параметру можно применять как переменные, так и литеральные значения, например 1049.

OrderID	OrderDate	TitleName	Quantity	ItemPrice
1049	2021-03-05 00:00:00	Pride and Predicates	1	6.95
1049	2021-03-05 00:00:00	The Join Luck Club	1	6.95
1049	2021-03-05 00:00:00	Catcher in the Try	1	5.95

Рис. 20.10. Результаты выполнения вашей новой хранимой процедуры GetOrder с аргументом в виде литерального значения 1049, переданным параметру _OrderID

21 Средства принятия решений в запросах

Освоив операции добавления, изменения и удаления данных в таблицах, перейдем к более «интеллектуальным» инструментам, предоставляемым SQL для логической проверки и автоматического принятия решений в запросах и хранимых процедурах. Как, к примеру, сгруппировать данные и вернуть значение 0, если результат функции `SUM` равен NULL? Или сделать так, чтобы вывод запроса зависел от заданных условий — будь то анализ параметров внутри хранимой процедуры или формирование различающегося по условиям результата? В этой главе мы подробно рассмотрим эти и многие другие сценарии.

21.1. Условные функции и выражения

Помните, мы уже не раз имели дело с одной из условных функций — `COALESCE`. Вы применяли ее в главе 15 для конкатенации полных имен авторов и в главе 20 для обработки NULL-значений в столбце `TitleName`. В первом примере функция `COALESCE` позволила избежать получения NULL в результате объединения, если у автора отсутствовало второе имя (`MiddleName`).

21.1.1. Функция COALESCE

В главе 15 мы прибегли к следующему запросу, передававшему для проверки два значения в функцию `COALESCE`:

```
SELECT CONCAT(FirstName, ' ', COALESCE(MiddleName, ''), ' ', LastName)
    AS AuthorName
FROM author;
```

Функция COALESCE последовательно обрабатывает любое количество выражений слева направо и возвращает первое из найденных значений, не равное NULL. Поскольку столбец MiddleName таблицы author был указан первым, функция COALESCE оценила значение MiddleName для каждого автора и определила, является ли оно NULL. Для каждой строки, где значение не было пустым, задействовалось значение MiddleName в контексте запроса. Для строк, где значение оказалось NULL, применялось второе значение — пустая строка, представленная двумя одиночными кавычками (''), которая использовалась при конкатенации.

Этим возможности функции COALESCE не ограничиваются: она может принимать не два, а сколько угодно выражений для проверки на NULL. Следующий пример демонстрирует работу COALESCE, когда первые два выражения оказываются NULL:

```
SELECT COALESCE(NULL, NULL, 'Я не null!') AS CoalesceTest;
```

Функция COALESCE сравнила первые два выражения, определила их как NULL-значения и вернула третье выражение — строку 'Я не null!' — как первое, не равное NULL. Мы могли бы передать функции COALESCE более трех аргументов, однако при обнаружении первого не NULL-выражения все последующие игнорируются.

К практике!

Выполните следующий запрос с использованием функции COALESCE:

```
SELECT COALESCE(NULL, NULL, 'Я не null!', 'Меня проигнорировали!')
    AS CoalesceTest;
```

21.1.2. Функция IFNULL

Еще одна распространенная функция для работы с NULL — IFNULL. Функция IFNULL работает почти так же, как COALESCE, за тем исключением, что она ограничена сравнением только двух аргументов. Если первое выражение определяется как NULL, то возвращается второе. Вот пример:

```
SELECT IFNULL(NULL, 'Я не null!') AS IfNullTest;
```

> **НА ЗАМЕТКУ** Функция IFNULL поддерживается не всеми реляционными СУБД. В Microsoft Access и SQL Server ее аналогом является ISNULL, а в Oracle — функция NVL. Несмотря на разницу в названиях, все они работают по тому же принципу, что и IFNULL.

Несмотря на то что в примерах мы пользовались литеральными значениями, COALESCE и IFNULL часто применяются в вычислениях, которые могут содержать

NULL-значения. Представим типичную задачу: нужно вывести список названий всех произведений и выяснить, включены ли они в какие-либо заказы.

Для начала определимся с таблицами, которые понадобятся в таком запросе. Нам нужна таблица `title`, поскольку она содержит названия книг (`TitleName`). Нам также потребуется таблица `orderitem` для доступа к столбцу `Quantity`, отражающему количество заказов для каждой товарной позиции. Наконец, нам необходима таблица `orderheader`: она связана с таблицами `title` и `orderitem` через столбцы `TitleID` и `OrderID` соответственно.

Способ соединения этих таблиц крайне важен для нашей выборки. Исходной таблицей должна быть `title`, потому что нам нужно общее количество продаж для каждого издания, однако для присоединения остальных таблиц необходимо задействовать `LEFT JOIN`, так как некоторые книги могут отсутствовать в заказах. Примени мы `INNER JOIN` для соединения всех таблиц, мы получили бы результирующий набор, включающий лишь произведения, которые были в заказах, — а это не соответствует цели запроса.

Также требуется выполнить группировку по столбцу `TitleName` из таблицы `title` и применить функцию `SUM` для получения суммы столбца `Quantity` из `orderitem` для каждого названия книги. Наш запрос будет выглядеть примерно так (результаты отображены на рис. 21.1):

```
SELECT
    t.TitleName,
    SUM(oi.Quantity) AS TotalQuantity
FROM title t
LEFT JOIN orderitem oi
    ON t.TitleID = oi.TitleID
LEFT JOIN orderheader oh
    ON oh.OrderID = oi.OrderID
GROUP BY t.TitleName
ORDER BY t.TitleName;
```

TitleName	TotalQuantity
A Table of Two Cities	NULL
Anne of Fact Tables	2
Catcher in the Try	11
David Emptyfield	NULL
Of Mice and Metadata	NULL
Pride and Predicates	25
Red Badge of Cursors	NULL
The Call of the While	2
The DateTime Machine	13
The Great GroupBy	1
The Join Luck Club	13
The Sum Also Rises	3

Рис. 21.1. Выборка всех названий книг с указанием общего числа заказанных экземпляров. Произведения, отсутствующие в заказах, представлены значением NULL

Если вы выполняли запросы из главы 16, добавлявшие новые издания, то некоторые строки выборки будут содержать NULL в столбце `TotalQuantity`. В отчетах о продажах величина NULL обычно не фигурирует, поэтому внесем небольшое улучшение в наш запрос, добавив `IFNULL`, возвращающий значение 0 для любого произведения, отсутствующего в заказах (результаты приведены на рис. 21.2):

```
SELECT
    t.TitleName,
    IFNULL(SUM(oi.Quantity),0) AS TotalQuantity
FROM title t
LEFT JOIN orderitem oi
    ON t.TitleID = oi.TitleID
LEFT JOIN orderheader oh
    ON oh.OrderID = oi.OrderID
GROUP BY t.TitleName
ORDER BY t.TitleName;
```

TitleName	TotalQuantity
A Table of Two Cities	0
Anne of Fact Tables	2
Catcher in the Try	11
David Emptyfield	0
Of Mice and Metadata	0
Pride and Predicates	25
Red Badge of Cursors	0
The Call of the While	2
The DateTime Machine	13
The Great GroupBy	1
The Join Luck Club	13
The Sum Also Rises	3

Рис. 21.2. Выборка всех названий книг с указанием общего числа заказанных экземпляров. Произведения, отсутствующие в заказах, представлены значением 0 вместо NULL благодаря применению функции IFNULL

Теперь для произведений, отсутствующих в заказах, отображается значение 0 вместо NULL, что, согласитесь, более привычно и информативно.

> **СОВЕТ** Поскольку функция `COALESCE` обладает большей функциональностью и поддерживается всеми СУБД, для обработки значений NULL она предпочтительнее функций `IFNULL` или `ISNULL`. Далее в книге вместо `IFNULL` мы будем пользоваться `COALESCE`.

21.1.3. Выражение CASE

Функции `COALESCE` и `IFNULL` позволяют нам выполнять проверку выражений на наличие NULL-значений. Но что, если необходимо проверить не на NULL, а на другие условия или по разным критериям? В таких случаях на помощь приходит выражение `CASE`.

Выражение CASE, которое в запросах частенько называют CASE-командой, является более мощным инструментом, чем COALESCE и IFNULL: оно позволяет оценивать различные условия и возвращать разные значения в зависимости от результата этих условий. С помощью CASE можно управлять возвращаемыми значениями, задействовав логику выбора, напоминающую конструкции естественного языка. Так, если требуется установить названия и цены книг со значением цены 7 долларов 95 центов и сформировать вывод, подтверждающий, что цена действительно равна означенной сумме, мы могли бы сформулировать это следующим образом: "I would like title name and price from the title table. When the price is $7.95, I want to say, 'This title is $7.95.' Otherwise, I want to say, 'This title is not $7.95.' " («Мне нужны названия и цены книг из таблицы title. Если цена равна 7.95 доллара, выведи: "Эта книга стоит 7.95 доллара", а если нет — "Эта книга не стоит 7.95 доллара"»).

Решить задачу, описанную в последних предложениях, нам поможет выражение CASE. Вот основные правила его составления:

- Оно должно начинаться с ключевого слова CASE.
- Оно должно содержать одно или несколько условий проверки равенства вида WHEN (некоторое значение или выражение) THEN (требуемое значение). Как и все проверки на равенство в SQL, она не обрабатывает NULL-значения.
- Для обработки любых значений, не удовлетворяющих условиям WHEN, может применяться блок ELSE, однако размещать его допускается лишь после всех условий WHEN. Большинство выражений CASE включают блок ELSE для учета неизвестных или NULL-значений.
- Выражение CASE должно завершаться ключевым словом END. Если выражение CASE помещено в раздел SELECT SQL-запроса, рекомендуется в целях удобочитаемости присвоить столбцу псевдоним.

На первый взгляд выражение CASE может показаться несколько замысловатым, однако на практике оно легко и удобно в применении. Вот как выглядит наш SQL из предыдущего примера с использованием CASE-инструкции (результаты представлены на рис. 21.3):

```
SELECT
    TitleName,
    Price,
    CASE Price
        WHEN 7.95 THEN 'Эта книга стоит 7 долларов 95 центов.'
        ELSE 'Эта книга не стоит 7 долларов 95 центов.'
        END AS IsPrice795
FROM title;
```

Как отмечалось ранее, конструкция CASE может обрабатывать не только значения столбцов, но и любое выражение, в том числе результаты различных операций:

от конкатенации двух и более столбцов до математических расчетов. Иными словами, CASE справится с любым выражением, которое поддается вычислению.

TitleName	Price	IsPrice795
Pride and Predicates	9.95	Эта книга не стоит 7 долларов 95 центов.
The Join Luck Club	9.95	Эта книга не стоит 7 долларов 95 центов.
Catcher in the Try	8.95	Эта книга не стоит 7 долларов 95 центов.
Anne of Fact Tables	12.95	Эта книга не стоит 7 долларов 95 центов.
The DateTime Machine	7.95	Эта книга стоит 7 долларов 95 центов.
The Great GroupBy	10.95	Эта книга не стоит 7 долларов 95 центов.
The Call of the While	8.95	Эта книга не стоит 7 долларов 95 центов.
The Sum Also Rises	7.95	Эта книга стоит 7 долларов 95 центов.
David Emptyfield	9.95	Эта книга не стоит 7 долларов 95 центов.
Red Badge of Cursors	7.95	Эта книга стоит 7 долларов 95 центов.
Of Mice and Metadata	8.95	Эта книга не стоит 7 долларов 95 центов.
A Table of Two Cities	9.95	Эта книга не стоит 7 долларов 95 центов.

Рис. 21.3. Название и цена всех изданий, а также столбец с псевдонимом IsPrice795. Значения в столбце IsPrice795 являются результатом проверки цены посредством выражения CASE

Рассмотрим пример с функцией округления ROUND (мы впервые познакомились с ней в главе 15). Если требуется получить целочисленное представление значения, такого как цена произведения, следует прибегнуть к выражению ROUND(Price, 0) для нахождения ближайшего целого числа.

Модифицируем наш последний запрос: найдем книги стоимостью приблизительно 8 долларов при помощи функции округления ROUND(Price, 0), а затем применим выражение CASE для возврата фактического утверждения о цене (результаты отображены на рис. 21.4):

```
SELECT
     TitleName,
    Price,
    CASE ROUND(Price, 0)
        WHEN 8 THEN 'Эта книга стоит примерно 8 долларов.'
        ELSE 'Эта книга не стоит примерно 8 долларов.'
        END AS IsPriceAround8Dollars
FROM title;
```

Два предыдущих запроса демонстрируют применение простых CASE-команд, предназначенных для сопоставления различных возможных значений с одним выражением. Помимо этого, можно задействовать так называемое поисковое выражение CASE для более сложных проверок, в частности при работе с диапазонами значений данных. При помощи поискового выражения CASE можно проверить одно или несколько выражений, определяя их истинность или ложность.

TitleName	Price	IsPriceAround8Dollars
Pride and Predicates	9.95	Эта книга не стоит примерно 8 долларов.
The Join Luck Club	9.95	Эта книга не стоит примерно 8 долларов.
Catcher in the Try	8.95	Эта книга не стоит примерно 8 долларов.
Anne of Fact Tables	12.95	Эта книга не стоит примерно 8 долларов.
The DateTime Machine	7.95	Эта книга стоит примерно 8 долларов.
The Great GroupBy	10.95	Эта книга не стоит примерно 8 долларов.
The Call of the While	8.95	Эта книга не стоит примерно 8 долларов.
The Sum Also Rises	7.95	Эта книга стоит примерно 8 долларов.
David Emptyfield	9.95	Эта книга не стоит примерно 8 долларов.
Red Badge of Cursors	7.95	Эта книга стоит примерно 8 долларов.
Of Mice and Metadata	8.95	Эта книга не стоит примерно 8 долларов.
A Table of Two Cities	9.95	Эта книга не стоит примерно 8 долларов.

Рис. 21.4. Название и цена всех изданий, а также столбец с псевдонимом IsPriceAround8Dollars. Значения в столбце IsPriceAround8Dollars являются результатом применения функции ROUND(Price, 0) и выражения CASE

К примеру, можно перестроить предыдущий запрос таким образом, чтобы производить поиск по ценовым диапазонам и определять, является ли каждая цена меньше, равной или больше 8 долларов. Для этого используется поисковое выражение `CASE`. Результат выполнения такого запроса показан на рис. 21.5:

```
SELECT
    TitleName,
    Price,
    CASE
        WHEN Price < 8.00 THEN 'Эта книга стоит дешевле 8 долларов.'
        WHEN Price = 8.00 THEN 'Эта книга стоит 8 долларов.'
        WHEN Price > 8.00 THEN 'Эта книга стоит дороже 8 долларов.'
        END AS IsPriceAround8Dollars
FROM title;
```

TitleName	Price	IsPriceAround8Dollars
Pride and Predicates	9.95	Эта книга стоит дороже 8 долларов.
The Join Luck Club	9.95	Эта книга стоит дороже 8 долларов.
Catcher in the Try	8.95	Эта книга стоит дороже 8 долларов.
Anne of Fact Tables	12.95	Эта книга стоит дороже 8 долларов.
The DateTime Machine	7.95	Эта книга стоит дешевле 8 долларов.
The Great GroupBy	10.95	Эта книга стоит дороже 8 долларов.
The Call of the While	8.95	Эта книга стоит дороже 8 долларов.
The Sum Also Rises	7.95	Эта книга стоит дешевле 8 долларов.
David Emptyfield	9.95	Эта книга стоит дороже 8 долларов.
Red Badge of Cursors	7.95	Эта книга стоит дешевле 8 долларов.
Of Mice and Metadata	8.95	Эта книга стоит дороже 8 долларов.
A Table of Two Cities	9.95	Эта книга стоит дороже 8 долларов.

Рис. 21.5. Название и цена всех изданий, а также столбец с псевдонимом IsPriceAround8Dollars. Значения в столбце IsPriceAround8Dollars являются результатом поискового выражения CASE, оценивающего, является ли значение Price меньше, равным или больше 8.00 долларов

Поисковые выражения CASE, которые проверяются на истинность или ложность, известны как *булевы* (или *логические*) *выражения* (Boolean expressions). Я понимаю, что глава перегружена различными видами выражений, да еще к тому же выражения CASE проверяют другие выражения. Однако попытайтесь запомнить, что выражения, обрабатываемые в частях WHEN предыдущего запроса, являются булевыми; мы еще вернемся к ним в последующих разделах главы.

НА ЗАМЕТКУ Пусть вас не смущает тот факт, что выражения CASE демонстрировались исключительно в предложении SELECT. Если подобная логика выбора требуется в других частях запросов, выражения CASE допустимо применять и в других предложениях, включая WHERE, HAVING и ORDER BY.

21.2. Управляющие конструкции

Функции и выражения — не единственные инструменты, которыми располагает SQL для оценки данных и принятия решений. Можно также применять ряд ключевых слов, позволяющих управлять тем, будут ли при определенных условиях выполняться те или иные SQL-команды.

Практически в каждой реляционной СУБД имеются ключевые слова, с помощью которых можно управлять логикой выполнения запросов. Если вы уже работали с языками программирования, эти ключевые слова покажутся вам вполне привычными. А если вы только начинаете изучать программирование, не волнуйтесь — слова эти интуитивно понятны.

21.2.1. IF и THEN

В первую очередь рассмотрим ключевое слово, с которого начинается любая программная структура с выбором, — IF («если»). Это ключевое слово является отправной точкой для *управляющей конструкции* (decision structure) с ветвлением, то есть для любого SQL-кода, где нужно определить, следует ли выполнять определенную инструкцию. Наши решения будут основываться на тех же булевых (логических) выражениях, что и в разделе 21.1.3. Это означает, что если условие истинно (true), соответствующий SQL-код выполняется; если же условие ложно (false), то он просто пропускается.

Управляющие конструкции обычно задействуются в хранимых процедурах, поэтому рассмотрим простой пример применения ключевого слова IF для реализации ветвления внутри хранимой процедуры, добавляющей данные в таблицу promotion. Можно создать хранимую процедуру для вставки строки с новым значением PromotionCode и при этом реализовать логическую проверку, чтобы не добавлять строку при отсутствии значения PromotionCode.

Прежде чем мы рассмотрим всю хранимую процедуру, давайте изучим отдельные части SQL-кода, который мы поместим внутри хранимой процедуры,

чтобы определить, существует ли значение PromotionCode. Вот первый фрагмент хранимой процедуры:

```
CREATE PROCEDURE AddPromotion (
    IN _PromotionID int,
    IN _PromotionCode varchar(10),
    IN _PromotionStartDate datetime,
    IN _PromotionEndDate datetime
)
BEGIN
```

Анализируя этот раздел хранимой процедуры, мы выделяем имя (AddPromotion) и четыре входных параметра: _PromotionID, _PromotionCode, _PromotionStartDate и _PromotionEndDate. Типы данных для этих параметров совпадают с типами соответствующих столбцов в таблице promotion. Также присутствует ключевое слово BEGIN после объявления параметров, обозначающее начало исполняемого блока процедуры.

> **СОВЕТ** Всегда создавайте параметры с такими же типами данных, как и у столбцов, с которыми они будут совершать операции чтения и записи. Применение другого типа данных заставит СУБД выполнять дополнительную работу и негативно скажется на производительности запросов. Если вам не известны типы данных столбцов, для их определения всегда можно прибегнуть к оператору SHOW COLUMNS (мы его рассматривали в главе 20). Хотя SHOW COLUMNS существует только в MySQL, в каждой СУБД имеются похожие ключевые слова, которые помогут вам просмотреть типы данных столбцов любой таблицы.

Далее рассмотрим SQL-код для оставшейся части процедуры:

```
IF _PromotionCode IS NOT NULL THEN
    INSERT INTO promotion (
    PromotionID,
    PromotionCode,
    PromotionStartDate,
    PromotionEndDate
    )
    SELECT
        _PromotionID,
        _PromotionCode,
        _PromotionStartDate,
        _PromotionEndDate
        ;
END IF;
END
```

В этом фрагменте ключевое слово IF проверяет, является ли условие _PromotionCode IS NOT NULL истинным. Если для _PromotionCode было предоставлено значение, отличное от NULL, выполняется инструкция INSERT. Если значение _PromotionCode равно NULL, добавления строки не произойдет.

Блок условия мы завершаем через `END IF`, а хранимую процедуру — при помощи `END`. Сформируем единую инструкцию для создания хранимой процедуры и реализуем ее, чтобы протестировать условную логику нашей управляющей конструкции. Задействуем в MySQL те же команды `DELIMITER` для смены завершения команды с точки с запятой на две прямые косые черты, что позволит нам выполнить всю хранимую процедуру:

```
DELIMITER //

CREATE PROCEDURE AddPromotion (
    IN _PromotionID int,
    IN _PromotionCode varchar(10),
    IN _PromotionStartDate datetime,
    IN _PromotionEndDate datetime
 )
 BEGIN

IF _PromotionCode IS NOT NULL THEN
    INSERT INTO promotion (
    PromotionID,
    PromotionCode,
    PromotionStartDate,
    PromotionEndDate
    )
    SELECT
        _PromotionID,
        _PromotionCode,
        _PromotionStartDate,
        _PromotionEndDate
        ;
END IF;
END //

DELIMITER ;
```

После выполнения этого SQL-кода хранимая процедура `AddPromotion` готова к работе. Для начала протестируем ее со следующими значениями параметров:

```
CALL AddPromotion (14, '2OFF2023', '2023-01-04', '2023-02-11');
```

Выполнение такой команды должно привести к сообщению `1 row(s) affected` («Затронута 1 строка») на панели вывода. Чтобы убедиться, что строка действительно добавилась, выполним проверочный запрос (результаты отображены на рис. 21.6):

```
SELECT *
FROM promotion
WHERE PromotionID = 14;
```

Удостоверившись в том, что наша хранимая процедура работает должным образом, когда результат `IF`-условия истинен, давайте протестируем ее, когда

результат ложен. Для этого выполним следующий SQL-код, передающий аргумент NULL для параметра _PromotionCode:

```
CALL AddPromotion (15, NULL, '2023-07-04', '2023-07-11');
```

PromotionID	PromotionCode	PromotionStartDate	PromotionEndDate
14	2OFF2023	2023-01-04 00:00:00	2023-02-11 00:00:00

Рис. 21.6. Результаты выборки всех строк из таблицы promotion, в которых PromotionID равен 14. Мы добавили эту строку с помощью хранимой процедуры AddPromotion

Запуск этой команды должен привести к сообщению об ошибке 0 row(s) affected («Затронуто 0 строк») на панели вывода. Можно воспользоваться запросом, аналогичным примененному выше, и проверить, что строка не добавлена. Выполнение следующего запроса не возвращает результатов:

```
SELECT *
FROM promotion
WHERE PromotionID = 15;
```

Вы разработали хранимую процедуру AddPromotion и изучили управляющую конструкцию, определяющую, необходимо ли добавлять новую строку в таблицу promotion. Однако представьте, что вы не знакомы с ее внутренней логикой. Допустим, вы выполнили вызов (CALL) этой процедуры, и она не вернула никаких результатов. Разве вам не захотелось бы понять, что именно пошло не так?

При разработке хранимых процедур, особенно если в них применяются логические конструкции ветвления, обычно хочется, чтобы программа предоставляла какой-то механизм обратной связи — информацию о том, как сработали заложенные в ней условия. Чтобы добавить такие функциональные возможности — в частности, чтобы процедура выводила сообщение пользователю, выполняла особое действие, когда условие ложно, или проверяла несколько условий подряд, — можно задействовать другие ключевые слова.

21.2.2. ELSE

Если IF проверяет условие на истинность, то ключевое слово ELSE задает альтернативное действие, которое выполняется, когда проверенное условие оказывается ложным или равно NULL. Используется ELSE аналогично IF с тем лишь отличием, что следует после IF-блока.

> **ВНИМАНИЕ** Хотя применение ELSE не является обязательным, но если ELSE присутствует, то только после IF. Попытка использовать ELSE без предшествующего IF приведет к ошибке синтаксиса.

Давайте еще раз посмотрим, как должна работать управляющая конструкция в процедуре AddPromotion:

1. Если параметр _PromotionCode не равен NULL, то выполняется вставка переданных значений в таблицу promotion.
2. В противном случае вставка значений не производится — вместо этого процедура должна вернуть сообщение, объясняющее, почему вставка была пропущена.

Представьте себе ELSE («иначе») как краткий аналог словосочетания «в противном случае», означающего ту операцию, которая будет запущена, если IF-условие не выполняется. Для реализации задачи, сформулированной в пункте 2, нужно добавить следующий фрагмент SQL-кода перед END IF:

```
ELSE
    SELECT 'Отсутствует PromotionCode, вставка (INSERT) пропущена.' AS Message;
```

Этот фрагмент программного кода означает, что если условие IF ложно или равно NULL, то в качестве Message будет возвращено литеральное значение 'Отсутствует PromotionCode, вставка (INSERT) пропущена'. Здесь все предельно просто. Выполним SQL-код DROP PROCEDURE AddPromotion, чтобы удалить хранимую процедуру AddPromotion, а затем воссоздадим ее с обновленной логикой:

```
DROP PROCEDURE AddPromotion;

DELIMITER //

CREATE PROCEDURE AddPromotion (
    IN _PromotionID int,
    IN _PromotionCode varchar(10),
    IN _PromotionStartDate datetime,
    IN _PromotionEndDate datetime
)
BEGIN

IF _PromotionCode IS NOT NULL THEN
    INSERT INTO promotion (
    PromotionID,
    PromotionCode,
    PromotionStartDate,
    PromotionEndDate
    )
    SELECT
        _PromotionID,
        _PromotionCode,
        _PromotionStartDate,
        _PromotionEndDate
        ;
ELSE
```

```
    SELECT 'Отсутствует PromotionCode, вставка (INSERT) пропущена.' AS Message;
END IF;
END //

DELIMITER ;
```

После выполнения SQL-кода, приведенного выше, можно повторить вызов `AddPromotion` с аргументом NULL для параметра `_PromotionCode`. Результаты на рис. 21.7 демонстрируют получение сообщения, определенного в блоке `ELSE`:

```
CALL AddPromotion (15, NULL, '2023-07-04', '2023-07-11');
```

Message
Отсутствует PromotionCode, вставка (INSERT) пропущена.

Рис. 21.7. Результат выполнения инструкции ELSE, добавленной в AddPromotion, которая выводит информативное сообщение о том, почему пропущена вставка

До сих пор мы имели дело с простой управляющей конструкцией ветвления, проверяющей одно конкретное условие, однако при помощи `IF` и `ELSE` можно также организовать обработку нескольких условий.

21.2.3. Набор условий

Наша предыдущая управляющая конструкция оценивала значение `_PromotionCode` по условию `NOT NULL`, однако для повышения практичности хранимой процедуры `AddPromotion` целесообразно учитывать потенциальные NULL-значения и в остальных параметрах. Для этого нужно усовершенствовать нашу конструкцию выбора, реализовав в ней несколько вариантов проверки. Рассмотрим новую структуру ветвления для `AddPromotion`:

1. Если `_PromotionID` присвоено значение NULL, следует вывести сообщение, объясняющее, почему пропущена вставка.
2. Если `_PromotionCode` присвоено значение NULL, следует вывести сообщение, объясняющее, почему пропущена вставка.
3. Если `_PromotionStartDate` присвоено значение NULL, следует вывести сообщение, объясняющее, почему пропущена вставка.
4. Если `_PromotionEndDate` присвоено значение NULL, следует вывести сообщение, объясняющее, почему пропущена вставка.
5. Во всех остальных случаях следует добавить предоставленные значения в таблицу `promotion`.

SQL-код внутри процедуры `AddPromotion`, реализующий такого рода конструкцию ветвления, требует применения нового ключевого слова. Это ключевое слово — `ELSEIF`, которое представляет собой дополнительный оператор `IF` для обработки всех дополнительных проверок. Применяется он следующим образом:

```
IF _PromotionID IS NULL THEN
    SELECT 'Отсутствует PromotionID, вставка (INSERT) пропущена.' AS Message;
ELSEIF _PromotionCode IS NULL THEN
    SELECT 'Отсутствует PromotionCode, вставка (INSERT) пропущена.' AS Message;
ELSEIF _PromotionStartDate IS NULL THEN
    SELECT 'Отсутствует PromotionStartDate, вставка (INSERT) пропущена.' AS Message;
ELSEIF _PromotionEndDate IS NULL THEN
    SELECT 'Отсутствует PromotionEndDate, вставка (INSERT) пропущена.' AS Message;
ELSE
    INSERT INTO promotion (
    PromotionID,
    PromotionCode,
    PromotionStartDate,
    PromotionEndDate
    )
    SELECT
        _PromotionID,
        _PromotionCode,
        _PromotionStartDate,
        _PromotionEndDate
        ;
END IF;
```

Если добавить в `AddPromotion` такую логическую конструкцию с `ELSEIF`, то для каждого возможного случая, когда вставка не удалась, будет выводиться пояснение.

ВНИМАНИЕ В SQL Server ключевое слово `ELSEIF` пишется в два слова: `ELSE IF`.

Пока что мы не предусмотрели никакого уведомления при успешной вставке значений. Для реализации такой функциональности требуется добавить еще одну инструкцию в блок `ELSE`. Поскольку теперь у нас будет несколько инструкций после `ELSE`, нужно сгруппировать их в единый блок при помощи ключевых слов `BEGIN` и `END`. Вот так будет выглядеть `ELSE`-блок с сообщением `'Вставка (INSERT) выполнена'`:

```
ELSE BEGIN
    INSERT INTO promotion (
    PromotionID,
    PromotionCode,
    PromotionStartDate,
    PromotionEndDate
    )
    SELECT
        _PromotionID,
```

```
        _PromotionCode,
        _PromotionStartDate,
        _PromotionEndDate
        ;

    SELECT 'Вставка (INSERT) выполнена' AS Message;
    END;
```

Ключевые слова `BEGIN` и `END` позволяют сгруппировать несколько команд в единый блок — подобно тому, как вся процедура заключена между `BEGIN` и `END`. По мере усложнения вашего SQL-кода вы будете нередко прибегать к этим ключевым словам, особенно в хранимых процедурах.

К практике!

Удалите (`DROP`) и заново создайте (`CREATE`) хранимую процедуру `AddPromotion`, внедрив усовершенствования в управляющей конструкции, рассмотренные в разделе 21.2.3. Затем попробуйте выполнить ее с аргументами NULL для различных параметров и убедитесь в том, что выдается соответствующее сообщение.

Сегодня вы проделали большую работу по созданию логических конструкций, и теперь в вашем распоряжении множество средств управления выбором в запросах. В следующей главе мы применим их для проверки отдельных строк данных.

21.3. Практическое занятие

1. С помощью `CASE` составьте запрос, выводящий из таблицы `promotion` три столбца:
   - столбец `PromotionCode`;
   - столбец с псевдонимом `PromotionCodeLeft1`, значения которого содержат первый символ из столбца `PromotionCode`;
   - столбец с псевдонимом `PromotionDiscount`, содержащий фразу `'Промокод со скидкой в X доллар(а)'`, где литеральное значение `X` заменено значением второго столбца.
2. Следующий запрос задействует выражение `CASE` в попытке заменить NULL-значения для `MiddleName` на пустую строку. Почему он не работает как положено?

```
SELECT
    FirstName,
    CASE MiddleName
        WHEN NULL THEN ''
```

```
        ELSE MiddleName
        END AS MiddleName,
    LastName
FROM author;
```

3. Требуется добавить логические операторы в `AddPromotion`, чтобы пропускать вставку строки в случае, если аргумент `_PromotionCode` уже существует в таблице `promotion`. Как реализовать такой код в SQL? (Вопрос повышенной сложности, но попробуйте ответить на него, применив полученные знания.)

21.4. Ответы

1. В решении поможет функция `LEFT` (упоминавшаяся в главе 14), а итоговый запрос будет примерно таким:

```
SELECT
    PromotionCode,
    LEFT(PromotionCode, 1) AS PromotionCodeLeft1,
    CASE LEFT(PromotionCode, 1)
        WHEN 1 THEN 'Промокод со скидкой в 1 доллар'
        WHEN 2 THEN 'Промокод со скидкой в 2 доллара'
        WHEN 3 THEN 'Промокод со скидкой в 3 доллара'
        END AS PromotionDiscount
FROM promotion;
```

2. Недопустимо проверять равенство с NULL, поскольку NULL никогда не равен NULL. Для корректной работы выражение `CASE` должно проверять условие следующим образом:

```
SELECT
    FirstName,
    CASE WHEN MiddleName IS NULL THEN ''
        ELSE MiddleName
        END AS MiddleName,
    LastName
FROM author;
```

3. Такую логику можно реализовать при помощи дополнительного блока `ELSEIF` следующим образом:

```
ELSEIF NOT EXISTS (SELECT PromotionCode FROM promotion WHERE
PromotionCode = _PromotionEndDate) THEN
    SELECT 'Дубликат значения PromotionCode, вставка (INSERT)
пропущена.' AS Message;
```

Этот пример посложнее тех, что мы разбирали в главе, вместе с тем он демонстрирует, что в условиях можно не только проверять NULL в булевом выражении — доступны даже операции `EXISTS` и `NOT EXISTS` для оценки целых запросов.

22 Работа с курсорами

В предыдущей главе мы рассматривали средства принятия решений в запросах и учились выполнять логические проверки. Применение ключевых слов `IF` и `THEN` позволило нам оценивать одно или несколько значений на соответствие определенному условию и в зависимости от результата выбирать последующие действия — в частности, добавлять новую строку в таблицу или нет.

Здесь же мы двинемся дальше и познакомимся с новыми способами анализа данных и принятия решений в SQL, сосредоточившись главным образом на курсорах. Курсоры позволяют обрабатывать набор данных поэлементно — построчно или по отдельным значениям. Как мы увидим, работа с курсорами имеет свои тонкости и требует учета ряда существенных факторов.

Применение курсоров в MySQL ограничено объектами базы данных, содержащими готовые SQL-команды, такими как хранимые процедуры. Поэтому, прежде чем перейти к созданию и использованию курсоров, мы разберем некоторые ранее не обсуждавшиеся возможности переменных и параметров.

22.1. Подробнее о переменных и параметрах

С переменными мы познакомились в главе 13, а с параметрами — в главе 20. Хотя и те и другие служат контейнерами для значений, переменные и параметры обладают разными свойствами в зависимости от способа их применения. Ниже показано, как можно задействовать некоторые из этих свойств.

22.1.1. Переменные в теле хранимых процедур

В главе 13 мы затрагивали вопрос, как объявляются переменные в MySQL и других реляционных СУБД. На случай если вы уже подзабыли, о чем идет речь, процитирую предупреждение оттуда:

ВНИМАНИЕ Данный метод объявления переменных в MySQL не является универсальным. При работе с другими реляционными СУБД, такими как SQL Server или PostgreSQL, требуется объявлять пользовательские переменные при помощи ключевого слова DECLARE с обязательным указанием конкретного типа данных.

Примечательно, что в MySQL для выполнения операций внутри хранимых процедур необходимо использовать более универсальный метод (с применением ключевого слова DECLARE, упомянутого в предупреждении). Рассмотрим конкретный пример. Чтобы объявить переменную и присвоить ей значение вне хранимой процедуры, мы напишем:

```
SET @TitleID = 101;
```

А вот в теле хранимой процедуры нам пришлось бы объявить переменную и ее тип данных, прибегнув к ключевому слову DECLARE:

```
DECLARE _TitleID int;
```

В целом, имеется целый ряд способов присвоения значения (такого, как 101) нашей переменной внутри хранимой процедуры. Один из них — посредством ключевого слова SET:

```
SET _TitleID = 101;
```

Кроме того, ключевое слово SET позволяет задавать значение динамически, через вложенный запрос. Вот пример:

```
SET _TitleID =
    (SELECT TitleID
    FROM title
    WHERE TitleName = 'Pride and Predicates');
```

И наконец, существует еще один способ присвоения конкретного значения переменной. Так, можно указать значение по умолчанию при помощи ключевого слова DEFAULT:

```
DECLARE _TitleID int DEFAULT 101;
```

До конца настоящей главы, изучая приемы работы с курсорами, мы будем придерживаться последнего из перечисленных способов.

22.1.2. Выходные параметры

В главе 20 мы отметили, что параметры в хранимых процедурах могут использоваться как для ввода, так и для вывода. До сих пор мы задействовали лишь входные параметры, которые позволяют передавать значение в хранимую процедуру, объявляя их с ключевым словом IN:

```
CREATE PROCEDURE GetSomeData(
    IN _TitleName varchar(50)
    )
```

Объявление параметра для вывода позволяет передавать значение, сформированное внутри хранимой процедуры и ее SQL-кода, в сценарий или даже в другую хранимую процедуру. Сделать это легко благодаря ключевому слову `OUT`, как показано в примере ниже:

```
CREATE PROCEDURE GetSomeData(
    OUT _TitleName varchar(50)
    )
```

Выходные параметры активно применяются в примерах с курсорами из этой главы, поэтому вам представится отличная возможность освоиться с ними и их применением.

22.2. Курсоры

В основе своей *курсор* (cursor) — это объект базы данных, который последовательно перебирает результаты запроса `SELECT`, позволяя извлекать и при необходимости изменять данные по одной строке за раз. Подобно мигающему курсору в текстовом редакторе, показывающему, где вы сейчас работаете, курсор в SQL указывает на текущую строку в наборе данных и дает возможность последовательно обрабатывать каждую строку по отдельности, в соответствии с поставленной задачей.

Принцип работы курсоров несложно объяснить, однако особенности их конструкции, более сложной в сравнении с другими объектами, такими как представления и хранимые процедуры, могут отпугивать начинающих разработчиков. Так, если представление или процедуру можно создать несколькими строчками кода, то даже самый простой курсор на первый взгляд выглядит куда запутаннее. Чтобы разобраться в курсорах, внимательно рассмотрим их основные составляющие.

22.2.1. Устройство курсора

Все курсоры, от простых до самых сложных, содержат четыре основных компонента, каждый из которых определяется соответствующим ключевым словом.

- `DECLARE` — подобно тому, как мы задействовали `DECLARE` ранее в этой главе для создания переменной внутри хранимой процедуры, мы применяем его для создания курсора. Раздел `DECLARE` содержит `SELECT`-запрос, определяющий набор данных, с которым будет работать курсор.

- `OPEN` — после создания курсора его необходимо открыть. Хотя в разделе `DECLARE` и определен набор данных для курсора, его `SELECT`-запрос не выполнится, пока мы не откроем курсор в этом разделе. Здесь курсор получает результаты нашего `SELECT`-запроса и сохраняет их в памяти сервера на время работы.
- `FETCH` — извлекает значения из набора данных по одной строке за раз. Этот раздел является основным компонентом курсора, где мы заполняем переменные, обрабатываем данные и выполняем другие целевые операции. Мы работаем с одной строкой до следующего вызова `FETCH`, извлекающего очередную строку при циклическом обходе результирующего множества, пока не пройдем весь набор данных.
- `CLOSE` — когда мы определяем, что завершили извлечение и обработку строк в наборе данных, мы закрываем курсор с помощью `CLOSE`, высвобождая ресурсы памяти сервера, занятые содержимым курсора.

НА ЗАМЕТКУ Четыре компонента, приведенные выше, являются обязательными для работы курсора и должны располагаться в указанном порядке.

ВНИМАНИЕ Хотя в MySQL этого не требуется, другие реляционные СУБД могут предписывать открепления (деаллокации) курсора после работы с ним. Сверьтесь с документацией вашей конкретной СУБД, чтобы узнать, необходим ли такой шаг при разработке курсоров вне среды MySQL.

22.2.2. Создание курсора

Допустим, требуется подсчитать количество экземпляров произведений, проданных по цене, указанной в таблице `title`, без применения каких-либо скидок по промоакциям. Мы могли бы написать хранимую процедуру, задействующую курсор для последовательного просмотра каждого заказа и проверки наличия книг, проданных по цене из таблицы `title`. При прохождении каждого заказа курсор может подсчитывать текущую сумму количества экземпляров, проданных по стандартной цене, в выходном параметре, который возвращает нам итоговое число таких книг.

Позже мы подробно разберем составляющие этого курсора. Пока же, при первом ознакомлении с приведенной хранимой процедурой, попытайтесь самостоятельно выделить четыре основных компонента курсора:

```
DELIMITER //

CREATE PROCEDURE GetTitleTotalQuantitySoldListPrice(
    OUT _TotalQuantitySold int
    )
BEGIN
```

```
DECLARE _Done boolean DEFAULT FALSE;

DECLARE _OrderID int;

DECLARE AllOrders CURSOR FOR

SELECT OrderID
FROM orderheader;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET _Done = TRUE;

SET _TotalQuantitySold = 0;

OPEN AllOrders;

GetOrders: LOOP

    FETCH AllOrders INTO _OrderID;

    SET _TotalQuantitySold = _TotalQuantitySold +

        (SELECT COALESCE(SUM(Quantity),0)
        FROM title t
        INNER JOIN orderitem oi
             ON t.TitleID = oi.TitleID
             AND t.Price = oi.ItemPrice
        WHERE oi.OrderID = _OrderID
        );

        IF _Done = TRUE THEN

            LEAVE GetOrders;
        END IF;

    END LOOP GetOrders;

    CLOSE AllOrders;

END //

DELIMITER ;
```

Нам предстоит проанализировать довольно большой объем SQL-кода, поэтому будем изучать его по частям. Сначала мы заменяем завершение команды на две косые черты, чтобы свободно пользоваться точками с запятой внутри нашей хранимой процедуры:

```
DELIMITER //
```

Затем при помощи CREATE PROCEDURE создаем процедуру с именем GetTitleTotal QuantitySoldListPrice. Обратите внимание — в процедуре определен параметр _TotalQuantitySold, который не только имеет тип данных int, но и используется как выходной параметр, на что указывает слово OUT перед его именем:

```
CREATE PROCEDURE GetTitleTotalQuantitySoldListPrice(
    OUT _TotalQuantitySold int
    )
BEGIN
```

Затем мы объявляем две переменные: _Done и _OrderID. Переменная _OrderID предназначена для значений OrderID, которые будут обрабатываться в курсоре по одному. А переменная _Done призвана отслеживать, завершили ли мы обход всех строк. Эта переменная объявлена с новым типом данных boolean, допускающим значения TRUE или FALSE. В начале процедуры она инициализируется со значением FALSE, поскольку мы еще не завершили (и даже не начали) обработку набора данных в нашем курсоре:

```
DECLARE _Done boolean DEFAULT FALSE;
DECLARE _OrderID int;
```

Теперь у нас есть первая составляющая нашего курсора: раздел DECLARE. Мы объявили наш курсор с именем AllOrders, и наш набор данных будет включать каждый OrderID из таблицы orderheader:

```
DECLARE AllOrders CURSOR FOR

SELECT OrderID
FROM orderheader;
```

Далее мы воспользуемся еще одной командой DECLARE, чтобы предписать системе проверять условие отсутствия строк для обработки (NOT FOUND SET), устанавливая нашу переменную _Done (которая сигнализирует, что мы завершили работу с курсором) в значение TRUE. Это даст нам возможность выйти из цикла и прекратить извлечение строк:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET _Done = TRUE;
```

Поскольку MySQL не позволяет устанавливать значение по умолчанию для параметров, присвоим _TotalQuantitySold начальное значение 0. Позже мы будем приращивать его по мере нахождения книг, проданных по стандартной цене:

```
SET _TotalQuantitySold = 0;
```

Теперь мы переходим ко второму разделу курсора, где открываем (OPEN) его. Здесь выполняется запрос из раздела DECLARE, и результирующее множество данных сохраняется в памяти для последующей обработки курсором:

```
OPEN AllOrders;
```

Далее мы задействуем инструкцию LOOP с именем GetOrders для итерации набора данных. Цикл LOOP является обязательным при работе с курсорами в MySQL:

```
GetOrders: LOOP
```

НА ЗАМЕТКУ Не все реляционные СУБД требуют применения LOOP с курсором, поэтому эта инструкция может не понадобиться для работы в другой системе. Обратитесь к документации вашей СУБД для выяснения специфических требований к реализации курсоров.

После открытия курсора мы извлекаем первую строку значений с помощью раздела FETCH нашего запроса. В контексте этого курсора выбирается только столбец OrderID, поэтому мы извлекаем первое значение OrderID и присваиваем его переменной _OrderID:

```
FETCH AllOrders INTO _OrderID;
```

Теперь, когда у нас есть значение _OrderID, можно проверить, есть ли в заказе книги, проданные по цене, указанной в таблице title. Если да, инкрементируем счетчик _TotalQuantitySold, прибавив к нему число произведений, проданных по стандартной цене в заказе. В противном случае применение COALESCE позволит нам увеличить значение _TotalQuantitySold на ноль. Не забывайте, что запрос, используемый курсором, обрабатывается в цикле, поэтому он выполняется для каждого OrderID из множества, определенного в разделе DECLARE:

```
SET _TotalQuantitySold = _TotalQuantitySold +

    (SELECT COALESCE(SUM(Quantity),0)
    FROM title t
    INNER JOIN orderitem oi
         ON t.TitleID = oi.TitleID
         AND t.Price = oi.ItemPrice
    WHERE oi.OrderID = _OrderID
    );
```

Напомню, что мы объявили обработчик, устанавливающий значение _Done в TRUE по достижении конца выборки в курсоре. Если это произошло, задействуем IF, чтобы выйти из цикла с помощью ключевого слова LEAVE:

```
IF _Done = TRUE THEN

    LEAVE GetOrders;
END IF;
```

НА ЗАМЕТКУ LEAVE — еще одно ключевое слово в MySQL, однако оно не применяется для выхода из циклов курсора в других СУБД. По этому вопросу также рекомендуется свериться с документацией управляющей системы и уточнить, как выйти из цикла курсора.

Цикл не может работать бесконечно, поэтому здесь мы завершаем его блок. Если мы не вышли из цикла посредством предыдущей инструкции, то извлекаем следующее значение `OrderID`, вернувшись к началу цикла:

```
    END LOOP GetOrders;
```

Достижение данного участка программного кода означает, что мы вышли из цикла и закончили работу с нашим курсором. Следовательно, теперь нам нужно закрыть курсор и высвободить занимаемые им ресурсы памяти. Для этого служит ключевое слово `CLOSE`, которое является четвертым, последним компонентом нашего курсора:

```
    CLOSE AllOrders;
```

Работа с курсором завершена — остается лишь отметить конец хранимой процедуры ключевым словом `END` и нашим пользовательским завершением команды, указанным в начале сценария:

```
END //
```

В самом конце мы возвращаем точку с запятой:

```
DELIMITER ;
```

После выполнения представленного SQL-кода мы сможем вызвать хранимую процедуру с выходным параметром, сохраняя результат в переменную `@TotalQuantitySold`, а затем выбрать значение этой переменной, чтобы отобразить общее количество произведений, проданных по стандартной цене, как показано на рис. 22.1:

```
CALL GetTitleTotalQuantitySoldListPrice(@TotalQuantitySold);
SELECT @TotalQuantitySold AS TotalQuantitySold;
```

TotalQuantitySold
27

Рис. 22.1. Суммарное количество (TotalQuantity) книг, проданных по стандартной цене, вычисленное разработанной нами хранимой процедурой с применением курсора для определения данного значения

К практике!

Создайте хранимую процедуру `GetTitleTotalQuantitySoldListPrice` и выполните предыдущий запрос, чтобы проверить общее количество произведений, проданных по стандартной цене.

Даже самые простые курсоры могут казаться чем-то мудреным, и все же я надеюсь, что представленное пошаговое описание работы курсора внутри хранимой

процедуры расставило все по местам. Если же мне не удалось полностью развеять ваши сомнения, то следует отметить, что существуют менее сложные альтернативы курсорам, способные обеспечить почти такие же функциональные возможности.

22.3. Альтернативы курсорам

Распространенной заменой курсора в SQL является цикл `WHILE`, который также позволяет осуществлять построчную обработку данных в пределах выборки, но требует значительно меньшего объема программного кода для своей реализации.

22.3.1. Цикл WHILE

Цикл `WHILE` проще уже тем, что избавляет от необходимости открывать или закрывать набор данных. Даже сам набор данных здесь не нужен — требуется лишь условие продолжения цикла для команды `WHILE`. Вот как мы перепишем хранимую процедуру `GetTitleTotalQuantitySoldListPrice`, чтобы применить цикл `WHILE` вместо курсора:

```
DROP PROCEDURE GetTitleTotalQuantitySoldListPrice;

DELIMITER //

CREATE PROCEDURE GetTitleTotalQuantitySoldListPrice(
    OUT _TotalQuantitySold int
    )
BEGIN

DECLARE _OrderID int;

SET _TotalQuantitySold = 0;
SET _OrderID = (SELECT MIN(OrderID) FROM orderheader);

WHILE _OrderID IS NOT NULL DO
    SET _TotalQuantitySold = _TotalQuantitySold +

        (SELECT COALESCE(SUM(Quantity),0)
        FROM title t
        INNER JOIN orderitem oi
             ON t.TitleID = oi.TitleID
             AND t.Price = oi.ItemPrice
        WHERE oi.OrderID = _OrderID
        );

    SET _OrderID =
        (SELECT MIN(OrderID)
        FROM orderheader
        WHERE OrderID > _OrderID);
```

```
    END WHILE;

END //

DELIMITER ;
```

Проанализируем новые компоненты, чтобы понять логику их работы. Прежде всего, вместо извлечения первого значения в переменную `_OrderID` посредством `FETCH` мы задействовали оператор `SET`, который применяет функцию `MIN` для выбора наименьшего значения из таблицы `orderheader`. Это значение фактически совпадает с первым значением, выбранным предыдущим курсором:

```
SET _OrderID = (SELECT MIN(OrderID) FROM orderheader);
```

Затем мы объявляем оператор `WHILE`, предписывающий выполнение SQL-кода внутри цикла до тех пор, пока переменная `_OrderID` не станет равной NULL. Цикл инициируется новым ключевым словом `DO`:

```
WHILE _OrderID IS NOT NULL DO
```

> **НА ЗАМЕТКУ** Хотя в MySQL используется ключевое слово `DO`, в других реляционных СУБД начало цикла часто обозначается словом `BEGIN`. Понимаю, что постоянные предупреждения о специфике реализации SQL для разных систем могут утомить читателя, тем не менее не забывайте сверяться с соответствующей документацией, чтобы избежать синтаксических ошибок.

Следующая часть покажется вам знакомой. В ней реализован тот же алгоритм, что мы применяли в нашем курсоре для пошагового приращения параметра `_TotalQuantitySold`:

```
    SET _TotalQuantitySold = _TotalQuantitySold +

        (SELECT COALESCE(SUM(Quantity),0)
        FROM title t
        INNER JOIN orderitem oi
             ON t.TitleID = oi.TitleID
             AND t.Price = oi.ItemPrice
        WHERE oi.OrderID = _OrderID
        );
```

Далее мы увеличим значение `_OrderID` до следующего большего значения, задействовав алгоритм, похожий на тот, что послужил нам для получения первого минимального значения. Разница лишь в том, что теперь мы берем минимальное значение, которое больше текущего, то есть следующее значение `OrderID` в таблице `orderheader`:

```
    SET _OrderID =
         (SELECT MIN(OrderID)
         FROM orderheader
         WHERE OrderID > _OrderID);
```

И наконец, мы завершаем SQL-код, содержащийся в цикле `WHILE`, с помощью `END WHILE`:

```
    END WHILE;
```

Представленный код содержит меньше SQL-команд, чем его аналог с курсором, и прекрасно справляется с той же задачей. Тем не менее, поскольку и курсор, и цикл `WHILE` реализуют построчную обработку данных, оба могут вызывать схожие проблемы — в частности, блокировки. *Блокировка* (blocking) возникает в том случае, когда выполняемый запрос удерживает ресурсы (к примеру, строки таблицы), препятствуя доступу к ним других запросов, которые вынуждены ожидать завершения текущей операции.

Хотя весь SQL-код, который мы писали и выполняли до сего момента, предназначен для нашей тестовой базы данных MySQL, где мы являемся единственными пользователями, в реальной практике вы будете работать с базами данных, к которым обращаются десятки, сотни или даже тысячи пользователей. В зависимости от настроек базы данных, не всегда находящихся под вашим контролем, курсор или цикл `WHILE` в многопользовательской среде может вызывать блокировки, замедляя выполнение чужих запросов или даже приводя к их сбоям при слишком долгом ожидании. Один из способов избежать подобных проблем — использование временных таблиц.

> **СОВЕТ** Во многих реляционных СУБД предусмотрены дополнительные настройки курсоров, выходящие за рамки рассмотренных нами возможностей, которые позволяют снизить вероятность блокировок. Однако параметры курсоров по умолчанию часто приводят к блокировкам.

22.3.2. Временные таблицы

Временные таблицы (temporary tables) полезны тем, что позволяют скопировать набор данных, с которым могут активно работать другие пользователи и процессы, в отдельную таблицу, существующую только в течение соединения с БД. Когда соединение закрывается, временные таблицы автоматически удаляются из базы. Что особенно важно при работе с курсорами и циклами `WHILE`, эти таблицы можно использовать без риска блокировок, так как они доступны только для запросов внутри нашего соединения.

Синтаксис создания временной таблицы в MySQL почти полностью совпадает с синтаксисом, который послужил нам для создания обычных таблиц в главе 18. Единственное различие — между ключевыми словами `CREATE` и `TABLE` ставится слово `TEMPORARY`.

Для предотвращения блокировок можно создать временную таблицу внутри нашей хранимой процедуры, заполнить ее диапазоном значений, с которыми

мы планируем работать, а затем направить наш цикл `WHILE` (или курсор) для перебора этой временной таблицы.

> **НА ЗАМЕТКУ** Почти каждая реляционная СУБД поддерживает временные таблицы, вот только правила их создания везде разные. Надеюсь, вы еще не устали видеть этот совет, но все же не ленитесь сверяться с соответствующей документацией.

Применительно к предыдущей версии `GetTitleTotalQuantitySoldListPrice` вот как можно удалить существующую хранимую процедуру и затем воссоздать ее, задействовав временную таблицу `orderheadertemp`, заменяющую обращение к таблице `orderheader`:

```
DROP PROCEDURE GetTitleTotalQuantitySoldListPrice;

DELIMITER //

CREATE PROCEDURE GetTitleTotalQuantitySoldListPrice(
    OUT _TotalQuantitySold int
    )
BEGIN
DECLARE _OrderID int;

SET _TotalQuantitySold = 0;

CREATE TEMPORARY TABLE orderheadertemp (OrderID int);

INSERT orderheadertemp (OrderID)
SELECT OrderID
FROM orderheader;

SET _OrderID = (SELECT MIN(OrderID) FROM orderheadertemp);

WHILE _OrderID IS NOT NULL DO
    SET _TotalQuantitySold = _TotalQuantitySold +

        (SELECT COALESCE(SUM(Quantity),0)
        FROM title t
        INNER JOIN orderitem oi
             ON t.TitleID = oi.TitleID
             AND t.Price = oi.ItemPrice
        WHERE oi.OrderID = _OrderID
        );

    SET _OrderID =
        (SELECT MIN(OrderID)
        FROM orderheadertemp
        WHERE OrderID > _OrderID);
    END WHILE;

END //

DELIMITER ;
```

Временные таблицы — настоящая палочка-выручалочка, применение которой не ограничивается предотвращением блокировок. Они позволяют сохранять наборы данных, многократно используемые в SQL-сценариях, а также упрощают громоздкие запросы, преобразуя их в набор простых и оптимизированных операций. Впрочем, после всех этих разговоров о курсорах, циклах `WHILE` и временных таблицах невольно возникает вопрос: а так уж ли все это необходимо?

22.4. Что важно учитывать при работе с курсорами

Если вы прочли все предыдущие главы и выполнили упражнения, то, вероятнее всего, вы уже придумали более простой способ подсчета общего числа книг, проданных по цене прайс-листа. И вы совершенно правы — для обработки этого запроса можно вполне обойтись без курсора и без цикла `WHILE`:

```
SELECT COALESCE(SUM(Quantity),0) AS TotalQuantitySold
FROM title t
INNER JOIN orderitem oi
    ON t.TitleID = oi.TitleID
    AND t.Price = oi.ItemPrice;
```

Увы, у курсоров и циклов `WHILE` есть один общий недостаток: в SQL почти всегда есть решение получше. Курсоры и циклы `WHILE` — не идеальный вариант для большинства запросов, поскольку построчная обработка данных вступает в противоречие с фундаментальным принципом работы реляционных СУБД, которые по своей сути предназначены для работы с наборами данных (множествами), а не с отдельными строками.

22.4.1. Мышление в терминах множеств

Начиная с самого первого запроса и вплоть до конца предыдущей главы, все, что мы рассматривали в настоящем руководстве, зиждется на принципах *программирования на основе множеств*, или, как нередко говорят сами разработчики, set-based (множественном) подходе (set-based programming). При таком подходе мы просто указываем СУБД, какие наборы данных нужно обработать, а дальнейшие шаги по выполнению запроса система определяет сама.

Мы занимались программированием на основе множеств во всех наших SQL-запросах до тех пор, пока не начали работать с ключевыми словами `IF`, `THEN` и `ELSE`, а также с курсорами и циклами `WHILE`. Подход, при котором системе задаются конкретные команды, что и как нужно выполнить, именуется *процедурным программированием* (procedural programming). Процедурное программирование характерно для многих традиционных языков общего назначения, но не является естественным для SQL.

Одна из причин, по которой курсоры получаются такими громоздкими и длинными, состоит в том, что нам приходится указывать реляционной СУБД каждый шаг, необходимый для обработки данных при помощи курсора. К сожалению, поскольку мы сами диктуем системе порядок действий, процедурный подход в SQL часто приводит к низкой производительности, высокому риску блокировок и большему расходу ресурсов сервера по сравнению с применением программирования на основе множеств.

22.4.2. Стоит ли вообще использовать курсоры?

Я вовсе *не* утверждаю, что курсоры — это зло и их нужно избегать любой ценой. Хотя, если честно, практически любую задачу можно решить и без их помощи. Подходя к заключительным главам этой книги, я лишь хочу призвать вас: смотрите на курсоры трезво и с известной долей скептицизма, опираясь на все уже полученные знания о SQL.

Умение создавать курсор может пригодиться, когда запрос невозможно выполнить иначе, кроме как обработав каждую строку набора данных по отдельности. Однако такие ситуации случаются редко, поэтому даже если вам кажется, что без курсора не обойтись, стоит задуматься: а нельзя ли решить эту задачу средствами программирования на основе множеств?

При анализе рабочего SQL-кода рассматривайте любой встреченный курсор как возможность повысить быстродействие программы, заменив его реализацией в духе set-based-программирования. Скорее всего, вам будут часто встречаться курсоры в сторонних хранимых процедурах, ведь многие разработчики, имеющие опыт процедурного программирования на других языках, склонны задействовать в SQL курсоры вместо набора инструментов программирования на основе множеств, разбору которых посвящена эта книга. Применяйте ваши знания не только для того, чтобы упростить код, связанный с курсорами, но и для повышения производительности хранимых процедур и снижения нагрузки на ресурсы сервера.

Надеюсь, вам будет интересно попробовать себя в разборе и усовершенствовании стороннего SQL-кода, ведь именно этим нам предстоит заняться в следующей главе.

22.5. Практическое занятие

Задания к этой главе несколько отличаются от предыдущих. Вам предстоит рассмотреть ряд сценариев и попытаться определить, нужно ли в них применять курсор для извлечения данных:

1. Проанализируйте все значения `TitleName` в таблице `title` и подсчитайте, сколько экземпляров каждой книги заказано покупателями из Калифорнии.

Запрос должен включать в себя покупателей со значением «CA» для столбца `State` в таблице `customer`.

2. Проанализируйте все значения `CustomerID` в таблице `customer` и выясните, приобретал ли покупатель книгу «Pride and Predicates». Добавьте столбец `OrderedPrideAndPredicates` и выведите в него значения `Да` (для тех, кто купил книгу) и `Нет` (для тех, кто не купил).
3. Проверьте каждый заказ и выясните, был ли он первым для покупателя. Выведите `OrderID`, `CustomerID` и `OrderDate` для всех первых заказов.
4. Проанализируйте все значения `CustomerID` в таблице `customer` и выясните, размещал ли покупатель заказ за последний год. Если да, то выполните хранимую процедуру `CreateThankYouMessage`. Эта процедура, содержащая единственный параметр `CustomerID`, должна генерировать благодарственное сообщение для покупателя.

22.6. Ответы

1. Курсор здесь не понадобится. Можно выбрать требуемый набор данных при помощи следующего запроса, который использует подзапрос для подсчета количества произведений, заказанных покупателями из Калифорнии:

```
SELECT
    t.TitleName,
    COALESCE(SUM(x.Quantity),0) AS QuantityFromCA
FROM title t
LEFT JOIN (
    SELECT
        oi.TitleID,
        oi.Quantity
    FROM orderitem oi
    INNER JOIN orderheader oh
        ON oi.OrderID = oh.OrderID
    INNER JOIN customer c
        ON oh.CustomerID = c.CustomerID
    WHERE c.State = 'CA'
    ) x
    ON t.TitleID = x.TitleID
GROUP BY t.TitleName;
```

2. Для решения этой задачи курсор тоже не требуется. Можно применить подзапрос и ключевое слово `CASE` для проверки наличия данных в подзапросе. Будьте внимательны: используйте `COALESCE` или иной способ обработки значений, отличных от NULL, поскольку NULL нельзя сравнивать внутри `CASE`. В примере ниже значения NULL по умолчанию заменяются на 0, так как `CustomerID` со значением 0 не существует:

```
SELECT
    c.CustomerID,
    CASE COALESCE(x.CustomerID,0)
        WHEN 0 THEN 'Нет'
        ELSE 'Да'
        END AS OrderedPrideAndPredicates
FROM customer c
LEFT JOIN (
    SELECT oh.CustomerID
    FROM orderheader oh
    INNER JOIN orderitem oi
        ON oh.OrderID = oi.OrderID
    INNER JOIN title t
        ON oi.TitleID = t.TitleID
    WHERE t.TitleName = 'Pride and Predicates'
    GROUP BY oh.CustomerID
    )x
    ON c.CustomerID = x.CustomerID
ORDER BY c.CustomerID;
```

3. И снова курсор здесь не нужен. Можно задействовать другой подзапрос для определения первого `OrderID` для каждого `CustomerID`, а затем выбрать нужные строки из подзапроса посредством соединения `INNER JOIN`:

```
SELECT
    oh.OrderID,
    oh.CustomerID,
    oh.OrderDate
FROM orderheader oh
INNER JOIN (
    SELECT
        ohf.CustomerID,
        MIN(ohf.OrderID) AS FirstOrderID
    FROM orderheader ohf
    GROUP BY ohf.CustomerID
    ) x
    ON oh.OrderID = x.FirstOrderID;
```

   Кроме того, в каждом из этих трех упражнений вы могли бы создать временную таблицу и задействовать SQL-код из подзапросов для ее заполнения, а затем соединяться с этой временной таблицей вместо применения подзапроса. Важно понимать, что курсор — не единственный способ достижения желаемых результатов, есть и другие варианты.

4. Это один из тех редких случаев, когда необходимо прибегнуть к курсору. Несмотря на наличие набора данных со значениями `CustomerID`, для выполнения запроса нельзя применить программирование на основе множеств, поскольку в хранимую процедуру `CreateThankYouMessage` допустимо передавать лишь одно значение за раз.

23 Работа со сторонними сценариями

Полагаю, что к настоящему моменту вы уверенно владеете основными средствами SQL, о которых узнали из книги. Мы рассмотрели самые важные и часто применяемые ключевые слова и команды, поэтому теперь вы готовы самостоятельно разрабатывать запросы на выборку данных и выполнять различные операции с ними в реляционной СУБД.

Однако, как и при изучении иностранного языка, в программировании важно освоить навыки не только письма, но и чтения. Вы должны уметь разбирать и анализировать SQL-код, содержащийся в хранимых процедурах и прочих компонентах баз данных вашей организации. Учитывая, что эта книга является введением в язык SQL, вы, вероятно, еще не раз обратитесь к интернету в поисках примеров SQL-сценариев, использующих ключевые слова и понятия, с которыми вы пока не сталкивались.

Для закрепления основных навыков и практического применения изученного материала в этой главе вам предстоит проинспектировать примеры SQL-кода, написанного другим разработчиком. Имейте в виду, что примеры эти работают корректно, однако вам необходимо внимательно изучить их, чтобы понять, чего именно хотел добиться их автор. Плюс ко всему, в представленных сценариях присутствуют отклонения от передовых методов и практик, обсуждавшихся в книге, и потому ваша задача — предложить способы оптимизации приведенного SQL-кода.

В конце главы не предусмотрено обычного тренинга — ведь вся глава целиком является своего рода обобщающим практикумом. В ней даже нет привычной рубрики «К практике!», но при желании вы можете самостоятельно выполнять и тестировать рассматриваемые примеры программного кода. Главное — понять,

как работают эти сценарии, и отыскать способы их улучшения, руководствуясь знаниями, которые вы почерпнули из этой книги.

23.1. Сценарий стороннего разработчика: создание таблицы

Все приведенные ниже примеры работают с новой таблицей `authorpayment`, в которой отслеживаются выплаты авторских гонораров. Строки в таблице отражают суммы, выплаченные каждому автору по произведениям и годам. Предполагается, что авторы получают выплаты ежегодно на основе продаж их произведений.

23.1.1. Запрос CREATE TABLE

Начнем с запроса, создающего таблицу:

```
CREATE TABLE authorpayment (
    ID int,
    Author int,
    Title int,
    PaymentYear char(4),
    PaymentAmount decimal(7,2)
    );
```

Несмотря на то что запрос этот довольно компактный, я уверен, что, если вы вспомните идеи и примеры, обсуждавшиеся в главах 18 и 19, вы сразу обнаружите ряд моментов, которые можно усовершенствовать. Потратьте немного времени на анализ сценария и отметьте, что бы вы изменили. Когда будете готовы, продолжайте чтение, и я поделюсь своими соображениями.

23.1.2. Обзор сценария CREATE TABLE

Первое, что бросается в глаза, — это столбец с безликим именем `ID`. Увы, присвоение первому столбцу таблицы имени `ID` — это распространенная практика, особенно при поспешном проектировании таблицы без четкой концепции. Рекомендуется точно указывать назначение столбца на случай, если он будет задействоваться в других запросах, поэтому следует дать этому столбцу осмысленное имя — `AuthorPaymentID`.

Кроме того, если столбец `AuthorPaymentID` предназначен для хранения уникальных значений, формирующих первичный ключ таблицы, следует добавить к столбцу ограничение `PRIMARY KEY` и даже рассмотреть возможность присвоения свойства `AUTO_INCREMENT` для автоматического приращения значений.

Двинемся дальше. Столбец `Author`, похоже, не был достаточно продуман. Он имеет тот же тип данных, что и столбец `AuthorID`, присутствующий в нескольких таблицах базы данных `sqlnovel`, поэтому для обеспечения согласованности его название следует изменить на `AuthorID`. Чтобы гарантировать целостность данных, столбец также должен содержать ссылку внешнего ключа на таблицу `author`, чтобы `AuthorID` в таблице `authorpayment` заполнялся только значениями из таблицы `author`. И еще — поскольку каждый платеж должен быть привязан к автору, для этого столбца следует добавить ограничение `NOT NULL`.

Те же замечания применимы и к столбцу `Title`. Следует переименовать его в `TitleID` для согласованности, создать ограничение `FOREIGN KEY`, которое ссылается на значения `TitleID` в таблице `title`, и добавить ограничение `NOT NULL`, чтобы гарантировать наличие `TitleID` в каждой строке.

Столбец `PaymentYear` выглядит несколько необычно: ему присвоен тип данных `char(4)`. Из этого следует, что значения будут храниться в виде строки символов, даже если годы являются числовыми значениями. Но при осуществлении выплат авторам в значении года точно не появятся нечисловые символы, следовательно, имеет смысл указать целочисленный тип данных (`int`).

> **НА ЗАМЕТКУ** Применение символьного типа данных для хранения числовых значений может быть целесообразным лишь том случае, когда требуется сохранение ведущих нулей. Типичным тому примером служат почтовые индексы США, многие из которых начинаются по крайней мере с одного нуля. Если ввести почтовый индекс 03872 в столбец с целочисленным типом данных, он сохранится как 3872. В силу данного обстоятельства почтовые индексы США обычно хранятся как символьные типы данных (`char`).

На столбец `PaymentYear` также следует наложить ограничение `NOT NULL`, поскольку каждая строка должна отражать определенный год. Кроме того, хотя это и не обязательно, можно добавить ограничение `CHECK` для столбца `PaymentYear`, чтобы разрешить лишь те годы, которые попадают в заданный диапазон. Такое ограничение позволит сузить допустимый набор значений, предотвращая множество потенциальных ошибок ввода, способных повлиять на целостность данных. Оптимальным диапазоном для данного ограничения представляется период с 2000 по 2100 год, ведь, честно говоря, даже если эта БД и будет работать в 2100 году, к тому времени она потребует капитального обновления.

Столбец `PaymentAmount` выглядит корректно: тип данных `decimal(7,2)` позволяет работать с суммами выплат до 99 999,99. Для этого столбца также следует добавить ограничение `NOT NULL`, так как каждая выплата требует указания такой суммы. Единственное дополнительное усовершенствование, которое можно рассмотреть, — это ограничение `CHECK` для значений, гарантирующее наличие

только положительных значений в этом столбце; надо полагать, авторам не будут начисляться отрицательные суммы.

23.1.3. Доработка запроса CREATE TABLE

Если собрать все воедино, исправленный SQL-код для создания таблицы `authorpayment` будет выглядеть примерно так:

```
CREATE TABLE authorpayment (
    AuthorPaymentID int NOT NULL AUTO_INCREMENT,
    AuthorID int NOT NULL,
    TitleID int NOT NULL,
    PaymentYear int NOT NULL CHECK (PaymentYear BETWEEN 2000 AND 2100),
    PaymentAmount decimal(7,2) NOT NULL CHECK (PaymentAmount BETWEEN 0.00 AND
     99999.99),
    CONSTRAINT PK_AuthorPayment PRIMARY KEY (AuthorPaymentID),
    CONSTRAINT FK_authorpayment_author FOREIGN KEY (AuthorID) REFERENCES
     author(AuthorID),
    CONSTRAINT FK_authorpayment_title FOREIGN KEY (TitleID) REFERENCES
     title(TitleID)
);
```

Думаю, вы понимаете, как внесенные нами правки позволят обеспечить целостность данных и гармонизируют таблицу с остальной БД. В будущем имеет смысл рассмотреть возможность добавления уникального индекса, включающего `AuthorID`, `TitleID` и `PaymentYear`, поскольку эти значения должны быть уникальными для каждой строки. Помимо этого, можно преобразовать первичный ключ к составному типу на основе данной комбинации столбцов вместо `AuthorPaymentID`, что избавит от необходимости создания отдельного уникального индекса.

23.2. Сценарий стороннего разработчика: вставка данных

Теперь рассмотрим сценарий хранимой процедуры, которая вставляет строки в нашу новую таблицу. Она должна запускаться для каждого года, агрегируя данные о продажах, а затем определяя размер авторского вознаграждения.

23.2.1. Хранимая процедура INSERT

Предлагаю рассмотреть следующую хранимую процедуру:

```
DELIMITER //

CREATE PROCEDURE InsertAnnualPayment(
    IN _PaymentYear int
```

```
    )
BEGIN

DECLARE _Done boolean DEFAULT FALSE;
DECLARE _TitleID int;
DECLARE _AuthorID int;
DECLARE _Royalty decimal(5,2);
DECLARE _AuthorCount int;
DECLARE _TotalSales decimal(7,2);
DECLARE _PaymentAmount decimal(7,2);

DECLARE AllTitles CURSOR FOR

SELECT TitleID, AuthorID
FROM titleauthor
ORDER BY
    TitleID,
    AuthorOrder;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET _Done = TRUE;

OPEN AllTitles;

GetTitles: LOOP

    FETCH AllTitles INTO _TitleID, _AuthorID;

    SET _Royalty = (
        SELECT Royalty
        FROM title
        WHERE TitleID = _TitleID
        );

    SET _AuthorCount = (
        SELECT COUNT(AuthorID)
        FROM titleauthor
        WHERE TitleID = _TitleID
        );

    SET _TotalSales = (
        SELECT SUM(orderitem.Quantity * orderitem.ItemPrice)
        FROM orderheader
        INNER JOIN orderitem
            ON orderheader.OrderID = orderitem.OrderID
        WHERE orderitem.TitleID = _TitleID
            AND YEAR(orderheader.OrderDate) = _PaymentYear
        );

    SET _PaymentAmount =
        COALESCE(CONVERT(
            ((_TotalSales * (_Royalty/100))/_AuthorCount), decimal(7,2))
            , 0.00);
```

```
    IF _PaymentAmount > 0.00 THEN
        INSERT authorpayment (
            AuthorID,
            TitleID,
            PaymentYear,
            PaymentAmount
            )
        SELECT
            _AuthorID,
            _TitleID,
            _PaymentYear,
            _PaymentAmount;
        END IF;

        IF _Done = TRUE THEN

            LEAVE GetTitles;
        END IF;

    END LOOP GetTitles;

    CLOSE AllTitles;

END //

DELIMITER ;
```

Внимательно проанализируйте хранимую процедуру и запишите ключевые моменты перед дальнейшим чтением.

23.2.2. Обзор хранимой процедуры INSERT

Если вы внимательно читали главу 22, то первое, что вы заметите в такой хранимой процедуре, — это применение курсора для определения ежегодных авторских отчислений. Смею надеяться, что, лишь завидев курсор, вы сразу подумали: а нельзя ли заменить этот построчный обход на обработку данных методами программирования на основе множеств?

Чтобы решить, можно ли заменить курсор, разберем сначала, какую функцию он выполняет в хранимой процедуре. Для этого внимательно изучим каждый ее раздел.

Начало хранимой процедуры вполне стандартное. Похоже, что требуется целочисленное значение для входного параметра `_PaymentYear`, который является единственным параметром:

```
DELIMITER //

CREATE PROCEDURE InsertAnnualPayment(
```

```
    IN _PaymentYear int
    )
BEGIN
```

После этого раздела объявляются несколько переменных. Хотя на первый взгляд типы данных всех переменных выглядят обоснованно, при внимательном рассмотрении обнаруживается, что значение `_Royalty` не соответствует типу `decimal(5,2)`, заданному для столбца `Royalty` в таблице `title`. Несовпадение типов данных может привести к ошибкам или несогласованности данных. Кроме того, в ходе разбора этого курсора вы увидите, что многие, если не все, из этих переменных потенциально избыточны:

```
DECLARE _Done boolean DEFAULT FALSE;
DECLARE _TitleID int;
DECLARE _AuthorID int;
DECLARE _Royalty int;
DECLARE _AuthorCount int;
DECLARE _TotalSales decimal(7,2);
DECLARE _PaymentAmount decimal(7,2);
```

Объявляется курсор с именем `AllTitles`. Обратите внимание, что, в отличие от курсоров, с которыми мы работали в главе 22, этот курсор использует два столбца вместо одного:

```
DECLARE AllTitles CURSOR FOR

SELECT TitleID, AuthorID
FROM titleauthor
ORDER BY
    TitleID,
    AuthorOrder;
```

Для определения условия выхода из цикла курсора создается переменная-обработчик `_Done`:

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET _Done = TRUE;
```

Затем происходит открытие курсора (`OPEN`), при котором извлекаются результаты запроса, задействованного курсором:

```
OPEN AllTitles;
```

Далее следует цикл (`LOOP`), служащий для обработки данных курсором:

```
GetTitles: LOOP
```

Извлекаем из курсора первые результаты и сохраняем значения в переменные `_TitleID` и `_AuthorID`:

```
    FETCH AllTitles INTO _TitleID, _AuthorID;
```

После этого осуществляется присвоение значений другим переменным. Первым задается значение для переменной `_Royalty`, соответствующее конкретному `_TitleID`. Однако она, по всей видимости, является излишней. Вместо заполнения и применения данной переменной куда проще позаимствовать значение непосредственно из таблицы `title`:

```
SET _Royalty = (
    SELECT Royalty
    FROM title
    WHERE TitleID = _TitleID
    );
```

Значение `_AuthorCount` определяется для конкретного `_TitleID`. Использование отдельного запроса для этой цели не является ошибочным, однако применение `GROUP BY` к таблице `titleauthor` с группировкой по `TitleID` в сочетании с `INNER JOIN` позволило бы включить и значение `Royalty`, упомянутое выше. Поскольку `Royalty` является столбцом в таблице `title`, между сгруппированными результатами по `TitleID` в таблице `titleauthor` и таблицей `title` существует отношение «один к одному»:

```
SET _AuthorCount = (
    SELECT COUNT(AuthorID)
    FROM titleauthor
    WHERE TitleID = _TitleID
    );
```

Значение `_TotalSales`, представляющее объем продаж в долларах, определяется путем суммирования (`SUM`) произведения количества (`Quantity`) на стоимость (`Price`) для книги по всем заказам, оформленным в год `_PaymentYear`. Строки, соответствующие `_PaymentYear`, вычисляются функцией `YEAR`, которая вычленяет год из каждого значения столбца `OrderDate` в таблице `orderheader`. Логично выделить этот расчет в отдельный запрос — впрочем, нам следует по возможности избегать такого применения функции `YEAR`. В главе 14 я обращал ваше внимание на то, что при всем своем удобстве функции могут ударить по производительности, когда требуется обработать миллионы строк в таблицах:

```
SET _TotalSales = (
    SELECT SUM(orderitem.Quantity * orderitem.ItemPrice)
    FROM orderheader
    INNER JOIN orderitem
        ON orderheader.OrderID = orderitem.OrderID
    WHERE orderitem.TitleID = _TitleID
        AND YEAR(orderheader.OrderDate) = _PaymentYear
    );
```

В последнем присваивании переменной выполняется расчет значения `_PaymentAmount` (суммы, которая должна быть выплачена автору на основе его

гонорара) с использованием значений _TotalSales, _Royalty и _AuthorCount. Приближаясь к пониманию логики курсора, задействованного хранимой процедурой, мы видим, что, за исключением значений _TotalSales и AuthorCount, для определения значений применяется масса ненужных запросов, поскольку разработчик явно не мыслил в категориях «множественного подхода»:

```
    SET _PaymentAmount =
        COALESCE(CONVERT(
            ((_TotalSales * (_Royalty/100))/_AuthorCount), decimal(7,2))
            , 0.00);
```

Если значение _PaymentAmount больше 0, строка, представляющая платеж, вставляется в таблицу authorpayment. Такой подход необходим при использовании курсора, тогда как в программировании на основе множеств структура IF…THEN не понадобилась бы. Результаты корректно реализованных соединений INNER JOIN исключили бы любые произведения и авторов, по которым не было продаж, что автоматически устранило бы нулевые выплаты:

```
    IF _PaymentAmount > 0.00 THEN
        INSERT authorpayment (
            AuthorID,
            TitleID,
            PaymentYear,
            PaymentAmount
            )
        SELECT
            _AuthorID,
            _TitleID,
            _PaymentYear,
            _PaymentAmount;
        END IF;
```

Когда все значения строк курсора извлечены, осуществляется выход из цикла:

```
        IF _Done = TRUE THEN

            LEAVE GetTitles;
        END IF;
```

В конце хранимой процедуры разработчик завершает цикл (LOOP), закрывает курсор, применяет END для обозначения окончания всех действий в хранимой процедуре, а затем возвращает точку с запятой:

```
    END LOOP GetTitles;

    CLOSE AllTitles;

END //

DELIMITER ;
```

Проанализировав всю хранимую процедуру целиком, мы можем внести улучшения, которые позволят обойтись без курсора, что не только уменьшит нагрузку на реляционную СУБД, но и исключит необходимость в каких-либо переменных.

23.2.3. Доработка хранимой процедуры INSERT

Первое, что нужно сделать, — это переписать разделы, содержащие запросы, которые мы хотим сохранить. Наш анализ показал, что можно использовать запрос для определения количества авторов каждого произведения (что требуется для расчета авторского гонорара) и включить значения `Royalty` из таблицы `title`. Запрос, который будет применен в подзапросе, может выглядеть так:

```
SELECT
    t.TitleID,
    t.Royalty,
    COUNT(ta.AuthorID) AS AuthorCount
FROM title t
INNER JOIN titleauthor ta
    ON t.TitleID = ta.TitleID
GROUP BY
    t.TitleID,
    t.Royalty
```

Анализ также показал, что в подзапросе можно задействовать логику определения продаж книги за год в долларах. Следует избегать применения функции для столбца `OrderDate` таблицы `orderheader`, что создало бы дополнительную нагрузку на систему. Можно воспользоваться различными функциями даты, чтобы получить значение года, а затем вычислить начальную и конечную даты для значения параметра `_PaymentYear`.

Первая функция даты — `MAKEDATE`. Она позволяет создать дату первого дня года, указав только значение года. Установим дату первого дня выбранного года следующим образом:

```
MAKEDATE(_PaymentYear, 1)
```

Теоретически возможно задействовать `MAKEDATE` для выбора последнего дня года, заменив значение 1 на 365, но это работает не для каждого года. В високосных годах 366 дней, и поскольку неизвестно, соответствует ли значение, переданное в параметре `_PaymentYear`, високосному году, лучше определять конечную границу диапазона как дату, предшествующую началу следующего года. Для такого вычисления следует применить функцию `DATE_ADD` следующим образом:

```
DATE_ADD(MAKEDATE(@Year, 1), INTERVAL 1 YEAR)
```

НА ЗАМЕТКУ Хотя не все СУБД располагают подобными конкретными функциями, во всех имеются аналогичные функции с различными названиями, позволяющие формировать даты из составных частей, таких как год, и производить вычисления с датами.

После определения диапазона дат, заменяющего использование функции YEAR, сформируем запрос для расчета общего объема продаж по каждому произведению, который будет применяться в подзапросе:

```
    SELECT
        oi.TitleID,
        SUM(oi.Quantity * oi.ItemPrice) AS TotalSales
    FROM orderheader oh
    INNER JOIN orderitem oi
        ON oh.OrderID = oi.OrderID
    WHERE oh.OrderDate >= MAKEDATE(@Year, 1)
        AND oh.OrderDate < DATE_ADD(MAKEDATE(@Year, 1), INTERVAL 1 YEAR)
    GROUP BY
        oi.TitleID
```

Нам необходим еще один запрос, который вычисляет сумму выплаты. Наличие этого значения для каждого AuthorID и TitleID за выбранный год позволяет заполнить таблицу authorpayment. Учитывая, что оба предыдущих запроса, предназначенных для подзапросов, содержат столбец TitleID, их соединение не представляет сложности.

Применив логику расчета суммы выплаты из исходной хранимой процедуры, можно интегрировать все компоненты в одну процедуру с единственным запросом. Учитывая определенную сложность реализации, добавим несколько комментариев в нашу хранимую процедуру, разъясняющих наши намерения:

```
DELIMITER //

CREATE PROCEDURE InsertAnnualPayment(
    IN _PaymentYear int
    )
BEGIN

INSERT authorpayment (
    AuthorID,
    TitleID,
    PaymentYear,
    PaymentAmount
    )
/* Рассчитать общую сумму гонорара для каждого автора */
SELECT
    ta.AuthorID,
    ta.TitleID,
    _PaymentYear,
    CONVERT((
        (sales.TotalSales * (royalty.Royalty/100))/royalty.AuthorCount),
        decimal(7,2)) AS RoyaltyPerAuthor
FROM titleauthor ta
INNER JOIN (
    /* Определить ежегодную выручку по каждой книге */
```

```
    SELECT
        oi.TitleID,
        SUM(oi.Quantity * oi.ItemPrice) AS TotalSales
    FROM orderheader oh
    INNER JOIN orderitem oi
        ON oh.OrderID = oi.OrderID
    WHERE oh.OrderDate >= MAKEDATE(@Year, 1)
        AND oh.OrderDate < DATE_ADD(MAKEDATE(@Year, 1), INTERVAL 1 YEAR)
    GROUP BY
        oi.TitleID
    ) sales
    ON ta.TitleID = sales.TitleID
INNER JOIN (
    /* Определить размер гонорара и количество авторов */
    SELECT
        t.TitleID,
        t.Royalty,
        COUNT(ta2.AuthorID) AS AuthorCount
    FROM title t
    INNER JOIN titleauthor ta2
        ON t.TitleID = ta2.TitleID
    GROUP BY
        t.TitleID,
        t.Royalty
        ) royalty
    ON ta.TitleID = royalty.TitleID;

END //

DELIMITER ;
```

Теперь у вас есть хранимая процедура без курсора, без лишних переменных и запросов. Можно заполнить таблицу `authorpayment`, ограничившись методами программирования на основе множеств, что должно быть вашей целью при написании SQL-кода. Нет нужды беспокоиться о несоответствии типов данных, в то время как функции применяются без риска снижения производительности.

Процедура стала намного лучше. Но есть ли резерв для дальнейшего усовершенствования?

23.2.4. Дальнейшая доработка хранимой процедуры INSERT

Освоив предыдущие главы, вы обрели солидный багаж знаний — смело применяйте новые ключевые слова и приемы при дальнейшей оптимизации сторонних сценариев. В этой процедуре еще есть что улучшить, особенно в подзапросах.

Два подзапроса в обновленной процедуре работают быстрее курсора, однако такое решение может оказаться неоптимальным в случае, когда таблицы `orderheader` и `orderitem` содержат миллионы записей. Возможно, следует

заменить на временную таблицу подзапрос, вычисляющий `TotalSales` для каждого произведения в указанном году. Пускай этот подход предполагает запись во временную таблицу большого объема данных, зато последующее соединение `INNER JOIN` будет считывать меньше данных, чем при помощи подзапроса. Разумеется, для текущего набора данных, где в каждой таблице всего несколько строк, использование временных таблиц не критично, но во многих сценариях тестирование демонстрирует повышение производительности при их применении.

Еще одно возможное улучшение связано с параметром `_PaymentYear`. Единственный параметр `_PaymentYear`, задействованный хранимой процедурой, не предоставляет достаточной гибкости, поэтому можно рассмотреть возможность его замены параметрами для начальной и конечной дат диапазона. Такой подход позволяет работать с годовыми, квартальными, месячными и даже произвольными диапазонами значений. Старайтесь делать сценарии гибче, насколько это возможно, с учетом потенциального разнообразия запросов в будущем.

Наконец, в зависимости от применения этих данных, целесообразно заменить хранимую процедуру `InsertAnnualPayment` и таблицу `authorpayment` представлением, которое вычисляет авторское вознаграждение и совокупный гонорар за книгу для каждой продажи. Помните, что представление — это просто сохраненный запрос, который можно вызывать в любой момент. Если убрать ограничение по конкретному диапазону дат, можно построить запрос для вычисления гонорара для каждого значения `orderitem` примерно так:

```
SELECT
    ta.AuthorID,
    ta.TitleID,
    oh.OrderID,
    oh.OrderDate,
    CONVERT((
        (SUM(oi.Quantity * oi.ItemPrice) * (t.Royalty/100))/ac.AuthorCount),
        decimal(7,2)) AS RoyaltyPerAuthor
FROM title t
INNER JOIN titleauthor ta
    ON t.TitleID = ta.TitleID
INNER JOIN orderitem oi
    ON ta.TitleID = oi.TitleID
INNER JOIN orderheader oh
    ON oi.OrderID = oh.OrderID
INNER JOIN (
    /* Определить размер гонорара и количество авторов */
    SELECT
        TitleID,
        COUNT(AuthorID) AS AuthorCount
    FROM titleauthor
    GROUP BY
        TitleID
        ) ac
```

```
    ON ta.TitleID = ac.TitleID
 GROUP BY
    ta.AuthorID,
    ta.TitleID,
    oh.OrderID
    oh.OrderDate;
```

Это представление устранит избыточность данных в таблице `authorpayment`. Оно также позволит запрашивать представление для любых значений `AuthorID`, `TitleID` и `OrderID`, а также для любого диапазона значений `OrderDate`. Представление не позволяет сохранять данные так, как это делает таблица `authorpayment`, но если вам не нужно сохранять данные, представление являет собой оптимальное решение.

Главная мысль, которую следует вынести из этой главы, такова: у вас есть выбор. За 24 часа вы познакомились с десятками ключевых слов и понятий и узнали, как эффективно применять их на практике. Надеюсь, что разбор приведенных сценариев оказался полезным и помог вам почувствовать себя увереннее в обращении с языком SQL.

24 Все только начинается

Вот мы и добрались до последней главы *«Изучаем SQL за месяц, занимаясь один час в день»*. Хочется верить, что книга эта оказалась для вас по-настоящему полезной и убедила в том, что даже без опыта программирования можно научиться писать эффективные SQL-запросы.

С первой же главы наша цель заключалась в том, чтобы вы могли сразу же эффективно применять язык SQL на практике. После всех рассмотренных и примененных подходов и ключевых слов вы с достаточной уверенностью можете составлять запросы, отвечающие самым различным требованиям. Теперь вы знаете, как фильтровать, объединять и группировать данные, а также как изменять их и даже создавать объекты БД — такие, как таблицы и хранимые процедуры. Уверен, теперь вам под силу разобраться в большинстве примеров SQL-кода, написанных другими.

И все же финал книги вовсе не означает завершение вашего знакомства с SQL. Скорее наоборот — это лишь начало! Ведь чем больше вы работаете с этим языком, тем чаще открываете новые интересные команды и объекты. Куда двигаться дальше? Вот несколько соображений.

24.1. Больше SQL

Как вы, должно быть, заметили, в панели навигации MySQL Workbench есть раздел Functions (Функции), который мы обошли вниманием. Хотя на протяжении всей книги вы работали с десятками функций, такими как `CONCAT` и `COALESCE`, эти функции там не представлены, поскольку они являются системными, а раздел Functions предназначен для пользовательских функций. Именно так — вы можете

создавать собственные функции! По мере накопления опыта работы с SQL вы столкнетесь с задачами, требующими оценки значений или выражений при помощи созданных вами функций.

Еще одно направление для дальнейшего изучения — это оконные функции. Хотя они доступны не во всех реляционных СУБД, при работе с системами, которые их поддерживают, можно выполнять мощные вычисления, в частности, агрегирование с накоплением, ранжирование и определение процентилей в пределах набора строк. В некотором смысле такие функции ведут себя подобно курсору, но без его недостатков, таких как блокировки, задержки и избыточное потребление системных ресурсов.

Когда нужно сформировать SQL-запрос на основе заранее не известных условий, многие СУБД предоставляют возможность применить *динамический SQL* (dynamic SQL). Этот механизм предполагает создание текстовой строки, содержащей SQL-инструкцию, которая впоследствии может быть интерпретирована и выполнена системой. Хотя такой подход встречается нечасто, он дает дополнительную гибкость при работе с SQL — к примеру, когда по ходу работы программы нужно менять условия фильтрации или даже имена таблиц, к которым выполняется запрос.

И это лишь малая толика средств и приемов, которые вам еще предстоит открыть на пути изучения SQL. Куда направиться дальше?

24.2. Прочие SQL-ресурсы

Лучший способ совершенствования навыков в любом языке, будь то язык общения с компьютером или с человеком, — это практика. Поэтому почти в каждую главу включены упражнения, позволяющие вам учиться мыслить в категориях SQL и применять его для решения задач. Можно продолжать практиковаться, используя базу данных `sqlnovel`, составляя запросы, которые, к примеру, добавляют новые строки в таблицы `orderheader` и `orderitem` или извлекают данные о продажах по категориям. Возможности для практики ограничены лишь вашим воображением.

А может, сейчас вам требуется работать не с MySQL, а с другой СУБД. В таком случае можно установить инструмент, позволяющий работать с выбранной системой, и найти пример базы данных для практики написания SQL-запросов. Бесплатные образцы баз данных существуют для каждой СУБД, поэтому просто воспользуйтесь любимой поисковой системой, чтобы их отыскать, и задействуйте эти базы для создания собственных тренировочных запросов. Чем больше вы практикуетесь в написании кода SQL, тем увереннее и быстрее сможете решать любые стоящие перед вами задачи.

Если эта книга оказалась для вас полезной, обратите внимание и на другие издания от Manning, посвященные конкретным СУБД. Они помогут вам улучшить навыки работы с SQL. В качестве примера приведу 100 SQL Server Mistakes and How to Avoid Them Питера Картера (Peter Carter) (https://www.manning.com/books/100-sql-server-mistakes-and-how-to-avoid-them), PostgreSQL Mistakes and How to Avoid Them Джимми Ангелакоса (Jimmy Angelakos) (https://www.manning.com/books/postgresql-mistakes-and-how-to-avoid-them)[1]. Как я не раз упоминал, каждая реляционная СУБД имеет свои отличия в синтаксисе SQL, и поэтому учебное пособие по конкретной платформе поможет вам углубить знания и тем самым укрепить профессиональные позиции. Понимание универсальных принципов SQL, о котором шла речь в этой книге, безусловно важно, однако практическая специализация в рамках одной конкретной СУБД позволит вам выстроить репутацию эксперта именно в этой разновидности SQL.

Возможно, в какой-то момент вы почувствуете, что писать запросы для выборки данных вам уже мало — захочется разобраться, как создаются базы данных. Тогда стоит присмотреться к книгам Understanding Databases Дэвида Клинтона (David Clinton) (https://www.manning.com/books/understanding-databases) и Grokking Relational Database Design Цяна Хао (Qiang Hao) и Михаила Цикердекиса (Michail Tsikerdekis) (https://www.manning.com/books/grokking-relational-database-design)[2]. Эти книги посвящены методам проектирования баз данных, обеспечивающим высокую производительность и масштабируемость при работе с огромными объемами современных данных. Здесь мы подробно не рассматривали проектирование баз, однако понимание их возможностей и ограничений — не менее значимая часть мастерства.

И главное — что бы вы ни выбрали, не теряйте любознательности и продолжайте учиться.

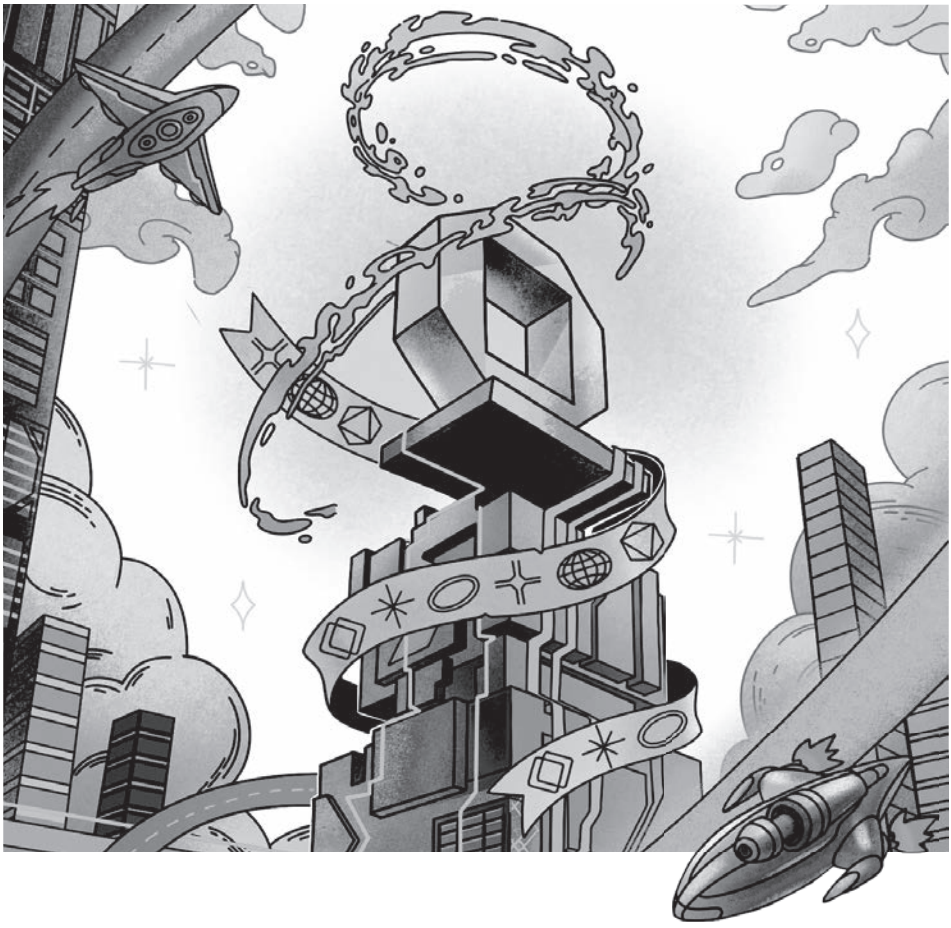
24.3. Счастливого пути!

Для меня было истинным удовольствием сопровождать вас в самом начале пути по изучению языка SQL. Искренне верю, что это путешествие будет долгим, плодотворным и наполненным открытиями. Пусть впереди вас ждет только удача, а все, чему вы уже научились, станет прочным фундаментом для новых достижений!

---

[1] *Дж. Ангелакос*. Антипаттерны PostgreSQL и как их избежать. СПб.: Питер, 2026.

[2] *Цян Хао*, *Михаил Цикердекис*. Грокаем проектирование реляционных баз данных. СПб.: Питер, 2026.



Эту книгу мы выпустили вместе с Orion soft

Orion soft создает ПО для ИТ-инфраструктуры Enterprise-бизнеса. Главные принципы разработки, которым отвечает каждый продукт компании, — стабильность, простота использования и безопасность.

В продуктовую экосистему Orion soft входит zVirt — платформа виртуализации серверов, дисков и сетей с одной из крупнейших баз инсталляций в России.

Orion soft стремится постоянно развивать как свои компетенции, так и российское ИТ-сообщество, собирая и распространяя лучшие современные практики разработки.



ВОСПОЛЬЗУЙТЕСЬ
ВОЗМОЖНОСТЬЮ

ПРИОБРЕСТИ КНИГИ
НА САЙТЕ ИЗДАТЕЛЬСТВА

piter.com



со скидкой 20%

по промокоду PITER

