В. В. Дунаев

БАЗЫ ДАННЫХ
ЯЗЫК SQL

2-издание

ДЛЯ СТУДЕНТА

SQL

BHV®

Вадим Дунаев

БАЗЫ ДАННЫХ
ЯЗЫК SQL

2-издание

Санкт-Петербург
«БХВ-Петербург»
2007

УДК 681.3.06
ББК 32.973.26-018.2
Д83

Дунаев В. В.

Д83 Базы данных. Язык SQL для студента: 2-е изд. доп. и перераб. — СПб.: БХВ-Петербург, 2007. — 320 с.: ил.

ISBN 978-5-9775-0113-2

Рассмотрен язык структурированных запросов для взаимодействия с базами данных — SQL, начиная с доступного изложения теории отношений (реляционной теории) и заканчивая вопросами администрирования СУБД с помощью запросов. На практических примерах подробно описаны основные конструкции языка, а также различные типы запросов: простые, сложные, рекурсивные. Показано, как осуществлять вычисления в запросах с помощью агрегатных функций и условных выражений. Рассмотрены операции над наборами записей, соединение таблиц, транзакции, хранимые процедуры и др. Уделено внимание администрированию СУБД с помощью запросов. Во втором издании добавлен материал по работе с базами данных посредством программ на языке PHP. Приведены задачи для самостоятельного решения.

Для студентов и программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Елена Кашлакова*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 16.07.07.
Формат 60×90 1/16. Печать офсетная. Усл. печ. л. 20.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0113-2

Оглавление

От автора **1**

Введение **3**

Как устроена эта книга 4
Как выполнять примеры SQL-выражений 7
Работа с Microsoft SQL Server 2000 7
Работа с Microsoft Access 2003 12

Глава 1. Основы реляционных баз данных **17**

1.1. Множества 17
1.2. Отношения 29
1.2.1. Общие сведения 29
1.2.2. Способы представления отношений 31
1.2.3. Операции над отношениями 35
1.3. Декомпозиция отношений 43
1.3.1. Корректная декомпозиция 44
1.3.2. Пример некорректной декомпозиции 44
1.3.3. Зависимости между атрибутами 45
1.3.4. Правила вывода зависимостей 55
1.3.5. Ключи 58
1.4. Ограничения целостности отношений 60
1.4.1. Семантическая целостность 61
1.4.2. Доменная целостность 61
1.4.3. Ссылочная целостность 62
1.5. Нормализация таблиц 63
1.5.1. Первая нормальная форма 66
1.5.2. Вторая нормальная форма 66
1.5.3. Третья нормальная форма 67
1.5.4. Доменно-ключевая нормальная форма 68
1.5.5. Денормализация 69

Глава 2. Основы SQL 71

2.1. Что такое SQL 71
2.2. Типы данных 76
2.2.1. Строки 78
2.2.2. Числа 80
2.2.3. Логические данные 83
2.2.4. Дата и время 83
2.2.5. Интервалы 86
2.2.6. Специальные типы данных 87
2.2.7. Пользовательские типы данных 89
2.2.8. Неопределенные значения 93
2.2.9. Преобразование типов 95

Глава 3. Простые выборки данных 97

3.1. Основное SQL-выражение для выборки данных 99
3.2. Уточнения запроса 102
3.2.1. Оператор *WHERE* 106
3.2.2. Оператор *GROUP BY* 114
3.2.3. Оператор *HAVING* 116
3.2.4. Оператор *ORDER BY* 117
3.2.5. Логические операторы 118
3.3. Задачи 119
Задача 3.1 120
Задача 3.2 120
Задача 3.3 120

Глава 4. Вычисления 121

4.1. Итоговые функции 121
4.2. Функции обработки значений 125
4.2.1. Строковые функции 126
4.2.2. Числовые функции 127
4.2.3. Функции даты-времени 130
4.3. Вычисляемые выражения 130
4.4. Условные выражения с оператором *CASE* 133
4.4.1. Оператор *CASE* со значениями 134
4.4.2. Оператор *CASE* с условиями поиска 135
4.4.3. Функции *NULLIF* и *COALESCE* 136

Глава 5. Сложные запросы ... 139

5.1. Подзапросы ... 140
5.1.1. Простые подзапросы ... 140
5.1.2. Связанные подзапросы ... 146
5.2. Теоретико-множественные операции ... 151
5.2.1. Декартово произведение наборов записей ... 151
5.2.2. Объединение наборов записей (*UNION*) ... 154
5.2.3. Пересечение наборов записей (*INTERSECT*) ... 157
5.2.4. Вычитание наборов записей (*EXCEPT*) ... 158
5.3. Операции соединения ... 158
5.3.1. Естественное соединение (*NATURAL JOIN*) ... 159
5.3.2. Условное соединение (*JOIN ... ON*) ... 161
5.3.3. Соединение по именам столбцов (*JOIN ... USING*) ... 162
5.3.4. Внешние соединения ... 164
5.3.5. Рекурсивные запросы ... 169
5.4. Задачи ... 173
Задача 5.1 ... 173
Задача 5.2 ... 175
Задача 5.3 ... 176
Задача 5.4 ... 176
Задача 5.5 ... 176

Глава 6. Добавление, удаление и изменение данных в таблицах ... 179

6.1. Добавление новых записей ... 180
6.2. Удаление записей ... 182
6.3. Изменение данных ... 185
6.4. Проверка ссылочной целостности ... 188

Глава 7. Создание и модификация таблиц ... 191

7.1. Создание таблиц ... 191
7.1.1. Ограничения для столбцов ... 193
7.1.2. Ограничения для таблиц ... 197
7.1.3. Внешние ключи ... 199
7.2. Удаление таблиц ... 202
7.3. Модификация таблиц ... 202

7.4. Представления ... 207
7.4.1. Что такое представление ... 207
7.4.2. Создание представлений ... 210
7.4.3. Изменение данных в представлениях ... 213
7.5. Задачи ... 215
Задача 7.1 ... 215
Задача 7.2 ... 215
Задача 7.3 ... 216
Задача 7.4 ... 216

Глава 8. Транзакции ... 217

8.1. Как устроена транзакция ... 218
8.2. Определение параметров транзакции ... 220
8.3. Уровни изоляции транзакций ... 221
8.3.1. Неподтвержденное чтение ... 221
8.3.2. Подтвержденное чтение ... 222
8.3.3. Повторяющееся чтение ... 223
8.3.4. Последовательное выполнение ... 223
8.4. Субтранзакции ... 224
8.5. Ограничения в транзакциях ... 225

Глава 9. Курсоры и применение SQL в приложениях ... 231

9.1. Объявление курсора ... 234
9.1.1. Чувствительность ... 235
9.1.2. Перемещаемость ... 235
9.1.3. Выражение запроса ... 236
9.1.4. Сортировка ... 236
9.1.5. Разрешение обновления ... 238
9.2. Открытие и закрытие курсора ... 238
9.3. Работа с отдельными записями ... 239
9.4. SQL в приложениях ... 243
9.4.1. Использование встроенного SQL ... 243
9.4.2. Использование программ на языке PHP ... 246

Глава 10. Постоянно хранимые модули ... 257

10.1. Составные команды ... 258
10.1.1. Атомарность ... 259

10.1.2. Переменные ... 260
10.1.3. Обработка состояния ... 261
10.2. Операторы условного перехода ... 264
10.2.1. Оператор *IF* ... 265
10.2.2. Оператор *CASE ... END CASE* ... 266
10.3. Операторы цикла ... 268
10.3.1. Оператор *LOOP...END LOOP* ... 268
10.3.2. Оператор *WHILE ... DO ... END WHILE* ... 269
10.3.3. Оператор *REPEAT ... UNTIL ... END REPEATE* ... 270
10.3.4. Оператор *FOR ... DO ... END FOR* ... 270
10.3.5. Оператор *ITERATE* ... 272
10.4. Хранимые процедуры и функции ... 273
10.5. Хранимые модули ... 276

Глава 11. Управление правами доступа ... 279

11.1. Пользователи ... 279
11.1.1. Администратор базы данных ... 280
11.1.2. Владелец объектов базы данных ... 281
11.1.3. Другие пользователи ... 281
11.1.4. Создание пользователей ... 281
11.2. Предоставление привилегий ... 282
11.2.1. Роли и группы ... 284
11.2.2. Право просмотра данных ... 285
11.2.3. Право изменять данные ... 286
11.2.4. Право удалять записи ... 286
11.2.5. Право на использование ссылок ... 286
11.2.6. Право на домены ... 288
11.2.7. Право предоставлять права ... 289
11.3. Отмена привилегий ... 289

Приложение. Зарезервированные слова SQL ... 293

Предметный указатель ... 297

От автора

Занимаясь приложениями реляционных баз данных более десяти лет, после двухлетней паузы я вдруг неожиданно для себя обнаружил, что изрядно подзабыл язык SQL. В памяти остались лишь общие идеи. Пришлось обратиться к нескольким книгам. И тогда я подумал, что хорошо бы иметь под рукой небольшую книжку, в которой было бы все самое необходимое, чтобы легко и быстро все вспомнить. А тем, кто не знал SQL никогда, эта книжка могла бы помочь быстро его освоить в такой степени, чтобы получать практически полезные результаты. Воплощение этой идеи вы держите в руках. Надеюсь, что данная книга окажется полезной как для новичков, так и для успевших кое-что забыть.

Во втором издании данной книги исправлены замеченные ошибки и добавлен материал по работе с базами данных посредством программ на языке PHP. Ваши замечания и отзывы я буду рад прочитать в своей гостевой книге по адресу **http://dunaevv1.narod.ru**.

Благодарности

Я искренне благодарен коллегам Дмитрию Лямаеву и Игорю Слюсаренко за практические советы и консультации по техническим вопросам.

Введение

SQL (Structured Query Language) — это структурированный язык запросов к реляционным базам данных. На этом языке можно формулировать выражения (запросы), которые извлекают требуемые данные, модифицируют их, создают таблицы и изменяют их структуры, определяют права доступа к данным и многое другое.

Запросы выполняются системой управления базой данных (СУБД). Если вы не являетесь специалистом по разработке и администрированию баз данных, то вполне можете быть их пользователем, который просматривает или/и изменяет данные в уже имеющихся таблицах. Во многих случаях эти и другие операции с базой данных выполняются с помощью специальных приложений, предоставляющих пользователю удобный интерфейс. Обычно приложения пишутся на специальных языках программирования (C, Pascal, Visual Basic и т. п.) и чаще всего создаются с помощью интегрированных сред разработки, например, Delphi, C++ Builder и др. Однако доступ к базе данных можно получить и без них — с помощью только SQL. Замечу также, что и специализированные приложения обычно используют SQL-фрагменты кода при обращениях к базе данных.

Таким образом, SQL — широко распространенный стандартный язык работы с реляционными базами данных. Синтаксис этого языка достаточно прост, чтобы его могли использовать рядовые пользователи, а не только программисты. В настоящее время обычный пользователь компьютера должен владеть, по крайней мере, текстовым редактором (например, Microsoft Word) и электронными таблицами (например, Microsoft Excel). Неплохо, если

он также умеет пользоваться базами данных. Различных СУБД существует много, а универсальное средство работы с базами данных одно — SQL. Знание SQL, хотя бы его основ, и умение его применять для поиска и анализа данных является фундаментальной частью компьютерной грамотности даже рядовых пользователей.

Как устроена эта книга

Данная книга посвящена языку манипулирования реляционными базами данных, а не их проектированию и сопровождению. Однако успех изучения и применения SQL существенно зависит от понимания того, как устроена реляционная база данных. Теоретическим источником реляционных баз данных является теория отношений (реляционная теория). Основам этой теории посвящена *глава 1*. Несмотря на обилие символики, изложенный в ней материал достаточно прост и вполне доступен широкому кругу читателей. Однако те, кого интересуют непосредственно SQL и возможность быстрее получить практический результат, могут пропустить эту главу при первом чтении. Изложение последующих глав построено в основном на рассмотрении примеров.

Таблицы, из которых состоит любая реляционная база данных, представляют собой некоторые отношения, а отношения являются не чем иным, как множествами записей (строк). Все запросы к базе данных, направленные на извлечение из нее нужных сведений, интерпретируются как инструкции по выполнению тех или иных операций, являющихся в конечном счете операциями алгебры множеств. Более или менее серьезные базы данных состоят из нескольких таблиц, между которыми могут существовать связи. Рассмотрение в реляционной теории декомпозиции одной таблицы на несколько других позволит вам понять противоположный процесс — проектирование базы данных как композиции нескольких таблиц.

Главы 2—4 содержат минимальный набор сведений, позволяющий в большинстве случаев извлекать из базы данных необходимую информацию.

В *главе 2* кратко рассматривается история и составные части SQL, а также типы данных. При первом прочтении желательно получить хотя бы поверхностное представление о типах данных. Дело в том, что наборы типов данных SQL и типов столбцов таблиц, поддерживаемых той или иной СУБД, как правило, не совпадают. Однако между большинством из них есть соответствие. Кроме того, можно использовать функцию приведения к заданному типу.

В *главе 3* описываются запросы на выборку данных, которые наиболее часто используются на практике. Это так называемые простые запросы, не содержащие вложенных запросов. Здесь же рассматриваются условия, которые могут применяться не только в запросах на выборку, но и при модификации данных.

Глава 4 посвящена вычислениям в запросах с помощью итоговых (агрегатных) функций и условных выражений.

В *главе 5* рассматриваются так называемые сложные запросы, а также теоретико-множественные операции над наборами записей, соединения таблиц и рекурсивные запросы. Нередко для получения данных приходится вначале выполнить некий вспомогательный запрос, чтобы затем его результат использовать при формулировке условия выборки данных в основном запросе. Вспомогательный запрос включается в выражение основного запроса, который называют сложным (содержащим подзапрос). Теоретико-множественные операции позволяют из нескольких наборов записей получить другой набор записей — объединение, пересечение или разность исходных наборов. Операция соединения таблиц дает в результате таблицу, записи в которой получаются путем некоторой комбинации записей соединяемых таблиц.

Модификация (изменение, добавление и удаление) данных описывается в *главе 6*.

Рано или поздно возникает задача создания новых или изменения структуры уже существующих таблиц. Средства SQL, позволяющие это сделать, рассмотрены в *главе 7*. Здесь же описываются представления — виртуальные таблицы, доступные многим пользователям.

При выполнении многоэтапных операций с базами данных, особенно в многопользовательском режиме, последовательности SQL-выражений объединяются в транзакции. Если какое-либо SQL-выражение в транзакции по какой-то причине не выполнилось, отменяется действие всех SQL-выражений в этой транзакции, а база данных возвращается в исходное состояние, в котором она находилась до начала транзакции. Иначе говоря, выполняется принцип "все или ничего". Транзакции описываются в *главе 8*.

В *главе 9* рассмотрены так называемые курсоры. Курсор позволяет сделать обработку одной или нескольких записей таблицы с помощью SQL-выражений и, таким образом, упрощает совместное использование SQL с другими языками программирования, на которых пишутся приложения баз данных. Выборка записей производится с помощью SQL-запроса, а проверка их содержимого — с помощью кода на процедурном языке.

Глава 10 посвящена постоянно хранимым модулям. На языке SQL можно написать процедуры и функции, которые будут содержать объявления переменных и составные команды SQL. Коды этих процедур и функций могут содержать управляющие структуры, а также обычные SQL-выражения. Функции и процедуры задаются в рамках так называемого модуля, который представляет собой контейнер для их размещения. Модуль создается специальной командой SQL и сохраняется в метаданных базы данных, т. е. становится ее компонентом, подобно таблицам, индексам и т. д. Таким образом, однажды создав модуль с процедурами и функциями, вы можете затем в частных SQL-запросах использовать их вызовы.

В *главе 11* рассматриваются некоторые возможности SQL по администрированию базы данных, а именно по предоставлению прав доступа к ее объектам. Разграничение прав доступа является важным средством защиты базы данных от неправильного использования содержащейся в ней информации различными категориями пользователей.

В конце некоторых глав приводятся задачи для самоконтроля усвоения прочитанного материала. Список зарезервированных слов SQL приведен в приложении.

Как выполнять примеры SQL-выражений

Чтобы выполнить примеры и решить задачи, приведенные в книге, необходимо установить на компьютере какую-нибудь СУБД, которых существует множество. Наиболее простым вариантом является Microsoft Access. Однако не все возможности SQL-92 поддерживаются этой СУБД. Я рекомендую установить какой-нибудь SQL-сервер, например, Microsoft SQL Server 2000 или PostgreSQL. Далее мы кратко рассмотрим создание базы данных и SQL-выражений с помощью SQL Server 2000 и Access 2003.

Работа с Microsoft SQL Server 2000

Установка Microsoft SQL Server 2000

Процедура инсталляции SQL Server 2000 с установочного CD-диска довольно проста. Если инсталлятор не запустился автоматически, то следует запустить программу \x86\setup\setupsql.exe, расположенную на установочном CD. Далее мастер предоставит вам возможность указать параметры установки. Один из возможных вариантов установки, пригодный для большинства новичков, работающих на локальном компьютере, выглядит следующим образом:

1. **Local Computer** (Локальный компьютер).
2. **Server and Client Tools** (Инструменты сервера и клиента). При этом будет инсталлирован сам сервер и средства администрирования.
3. Настройка учетных записей служб:
   - **Use the same account for each service. Auto start SQL Server Service** (Использовать одну учетную запись для всех служб с их автоматическим запуском при загрузке операционной системы);
   - **Use the Local System account** (Использовать локальную учетную запись);

- **Auto Start Service** (Автоматический запуск служб при загрузке операционной системы).

4. **Typical** (Установка всех компонентов).

После инсталляции сервера баз данных в меню Windows **Пуск** | **Программы** | **Microsoft SQL Server** запустите утилиту Enterprise Manager, которая предоставляет пользовательский интерфейс для настройки, управления и доступа к данным на сервере (рис. В1). В левой части окна утилиты отображается древовидный список, в котором щелчком левой кнопкой мыши следует раскрыть узел **Microsoft SQL Servers/SQL Server Group**. Если установка прошла успешно, то в этом узле должен находиться узел, имя которого состоит из имени вашей учетной записи, за которым следует "(Windows NT)". На рис. В1 это **DVV (Windows NT)**. В правой части окна отображаются элементы того узла, который выделен в левой части окна (в дереве).

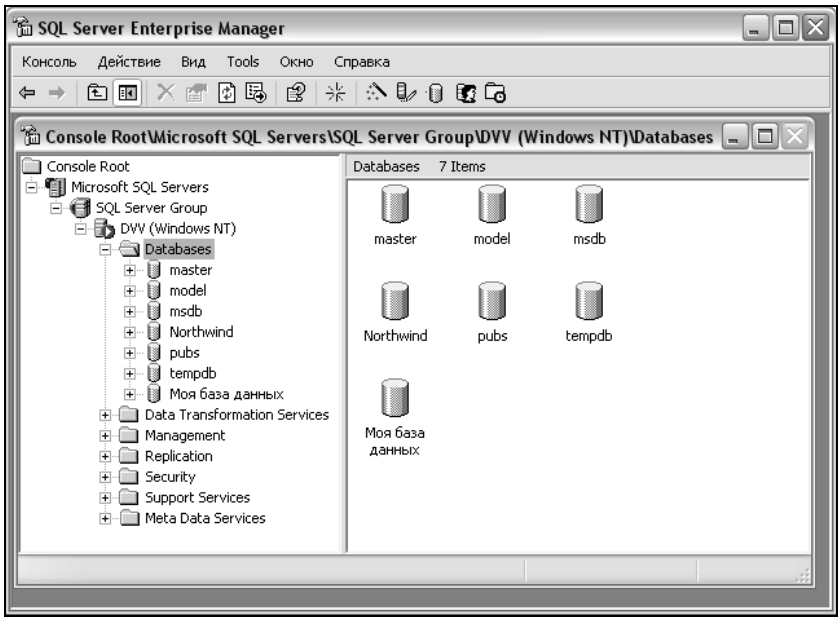


Рис. В1. Окно утилиты Enterprise Manager

Создание базы данных в Enterprise Manager

Чтобы выполнить учебные примеры SQL, необходимо создать учебную базу данных, на которой вы будете экспериментировать. Для этого в древовидном списке окна утилиты Enterprise Manager раскройте узел, соответствующий вашей учетной записи (на рис. B1 **DVV (Windows NT)**). Среди содержащихся в нем подузлов имеется **Databases** (Базы данных). Щелкните на нем правой кнопкой мыши и в раскрывшемся контекстном меню выберите **New Database** (Новая база данных). В результате откроется диалоговое окно **Database Properties** (Свойства базы данных). В поле **Name** (Имя) введите имя базы данных (например, "Моя база данных") и щелкните на кнопке **OK**. В результате в узле **Databases** появится узел с именем, совпадающим с именем базы данных.

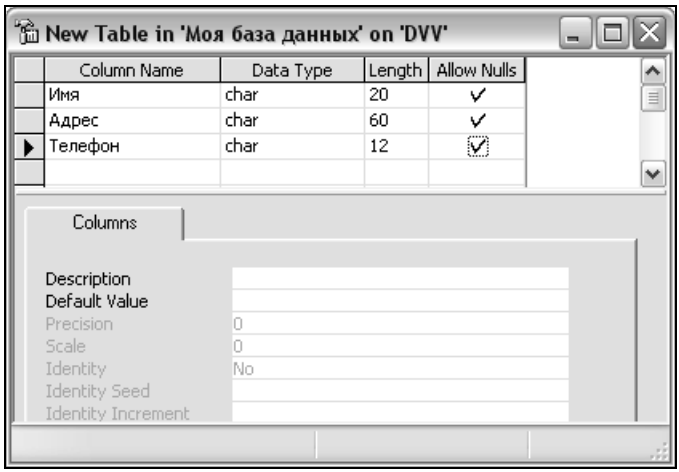


Рис. B2. Окно **New Table** для определения столбцов таблицы

Итак, создана пустая база данных, которая пока не содержит ни одной таблицы с пользовательскими данными. Теперь создайте какую-нибудь таблицу, например, `Клиенты` с символьными столбцами `Имя`, `Адрес`, `Телефон`. Для этого щелкните правой кнопкой мыши на имени вашей базы данных в древовидном списке и в контекстном меню выберите **Создать** | **Table**. В результате откроется окно **New Table** (Новая таблица), показанное на рис. B2.

В этом окне следует ввести имена столбцов (**Column Name**), тип данных (**Data Type**) и длину данных (**Length**). В рассматриваемом примере все столбцы являются символьными (строковыми, текстовыми), т. е. имеют тип `char`.

После завершения определения столбцов (структуры) создаваемой таблицы закройте окно, после чего появится запрос о сохранении введенных данных. Подтвердите необходимость сохранения и в открывшемся диалоговом окне введите имя созданной таблицы (например, `Клиенты`), а затем щелкните на кнопке **OK**.

Если в древовидном списке раскрыть узел с именем вашей базы данных и щелкнуть на узле **Tables** (Таблицы), то в правой части окна Enterprise Manager отобразятся все таблицы, входящие в эту базу данных. Среди множества служебных таблиц вы найдете и свою только что созданную таблицу.

Вы можете изменить структуру созданной таблицы или/и изменить содержащиеся в ней данные.

Для изменения структуры таблицы щелкните правой кнопкой мыши на ее имени и в контекстном меню выберите **Design Table** (Разработка таблицы). В результате откроется окно, аналогичное показанному на рис. В2. Теперь можно добавить новые или удалить имеющиеся столбцы, изменить их имена и/или другие параметры.

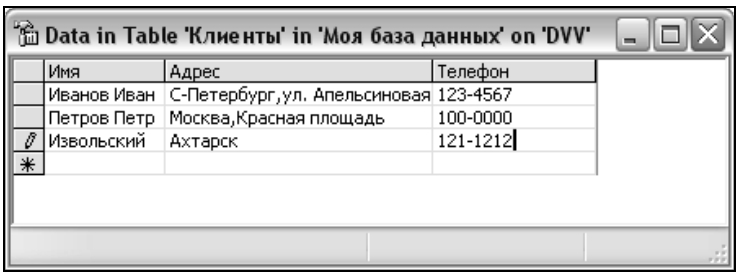


Рис. В3. Окно для модификации данных таблицы

Для ввода, редактирования и удаления данных в таблице щелкните правой кнопкой мыши на ее имени и в контекстном меню выберите **Open Table** | **Return all rows** (Открыть таблицу | Вернуть все записи). В результате откроется окно, показанное на рис. В3.

В этом окне можно вводить и изменять значения столбцов, добавлять и удалять строки (записи). Поскольку новая таблица пуста, то начать следует с добавления записей.

Аналогичным образом можно создать и другие таблицы базы данных.

Создание и выполнение SQL-выражений в SQL Query Analyser

Для создания, редактирования и выполнения выражений на языке SQL рекомендуется использовать утилиту SQL Query Analyser. Эту утилиту можно вызвать из меню **Tools** (Инструменты), расположенного в верхней части окна Enterprise Manager. В результате откроется окно, показанное на рис. В4. Здесь можно ввести, как в обычном текстовом редакторе, SQL-выражение.

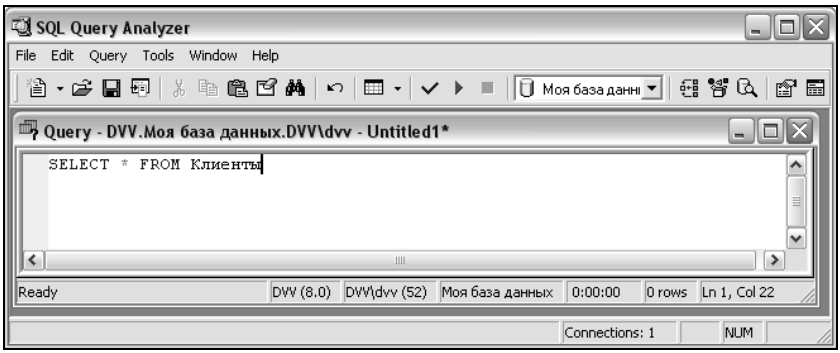


Рис. В4. Окно утилиты SQL Query Analyser

Прежде чем выполнить введенное SQL-выражение, рекомендуется его проверить, щелкнув на кнопке **Parse Query** (Анализировать запрос) с изображением галочки или нажав клавиши <Ctrl>+<F5>. Для выполнения выражения достаточно щелкнуть на кнопке с изображением стрелки вправо или нажать клавишу <F5>.

При желании созданное SQL-выражение можно сохранить в текстовом файле с расширением sql. Сохраненные в файлах SQL-выражения можно снова открыть в окне SQL Query Analyser. Соответствующие команды выбираются из меню **File**.

Работа с Microsoft Access 2003

Создание базы данных

При запуске программы Access открывается главное окно, в правой части которого находится панель **Приступая к работе**. На этой панели в разделе **Открыть** выберите команду **Создать файл**. В результате отобразится панель **Создание файла**. В разделе **Создание** на этой панели выберите команду **Новая база данных**. В открывшемся диалоговом окне **Файл новой базы данных** введите имя файла создаваемой базы данных (например, "Моя база данных") и сохраните его в какой-нибудь папке. Имя файла базы данных в Access имеет расширение mdb.

Итак, пустая база данных создана. В главном окне Access откроется окно **Моя база данных: база данных**. В этом окне выберите **Создание таблицы в режиме конструктора**. Откроется окно **Таблица1: таблица**, показанное на рис. В5. В этом окне определяется структура (свойства столбцов) таблицы — имена, типы, размеры и другие параметры столбцов.

По завершении определения столбцов (структуры) создаваемой таблицы закройте это окно, после чего появится запрос о сохранении введенных данных. Подтвердите сохранение и в открывшемся диалоговом окне введите имя созданной таблицы (например, `Клиенты`) и щелкните на кнопке **OK**.

В окне **Моя база данных: база данных** появится пиктограмма с именем созданной таблицы. Вы можете изменить структуру созданной таблицы или/и изменить содержащиеся в ней данные.

Для изменения структуры таблицы щелкните правой кнопкой мыши на ее имени и в контекстном меню выберите **Конструктор**. В результате откроется окно, аналогичное показанному на рис. В5. Теперь можно добавлять новые или удалять имеющиеся столбцы, изменять их имена и/или другие параметры.

Двойной щелчок на пиктограмме с именем таблицы откроет ее в режиме редактирования данных (рис. В6). В этом окне можно вводить и изменять значения столбцов, добавлять и удалять строки (записи). Поскольку новая таблица пуста, то следует начать с добавления записей.

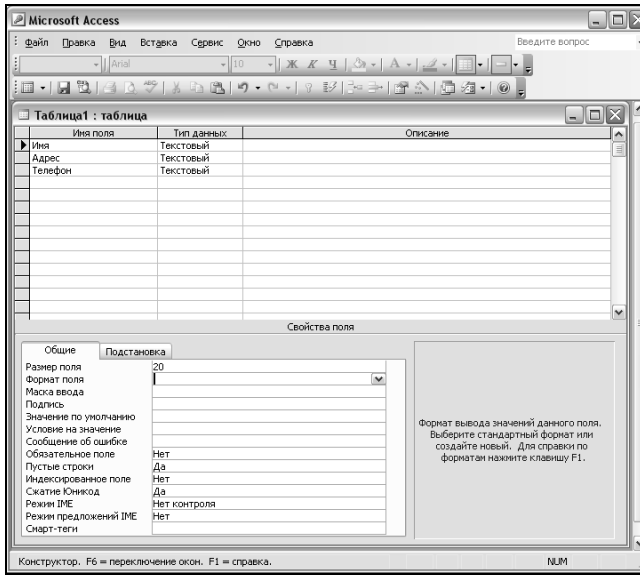


Рис. В5. Окно **Таблица1: таблица** для определения столбцов таблицы

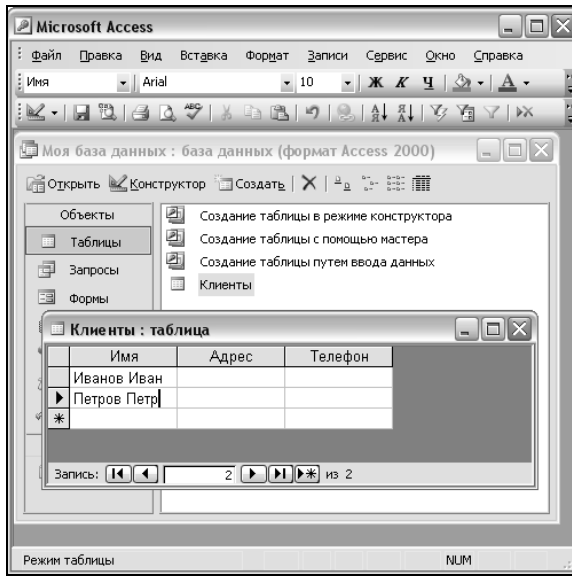


Рис. В6. Окно для модификации данных таблицы

Аналогичным образом можно создать и другие таблицы базы данных.

Создание и выполнение SQL-выражений

В окне **Моя база данных: база данных** (см. рис. В6) в списке объектов выберите **Запросы**, а в области задач, расположенной в правой части этого окна, выполните двойной щелчок на опции **Создание запроса в режиме конструктора**. В результате откроются два окна: **Запрос1: запрос на выборку** и **Добавление таблицы**. Окно **Добавление таблицы** закройте, после чего на экране останутся окна, показанные на рис. В7.

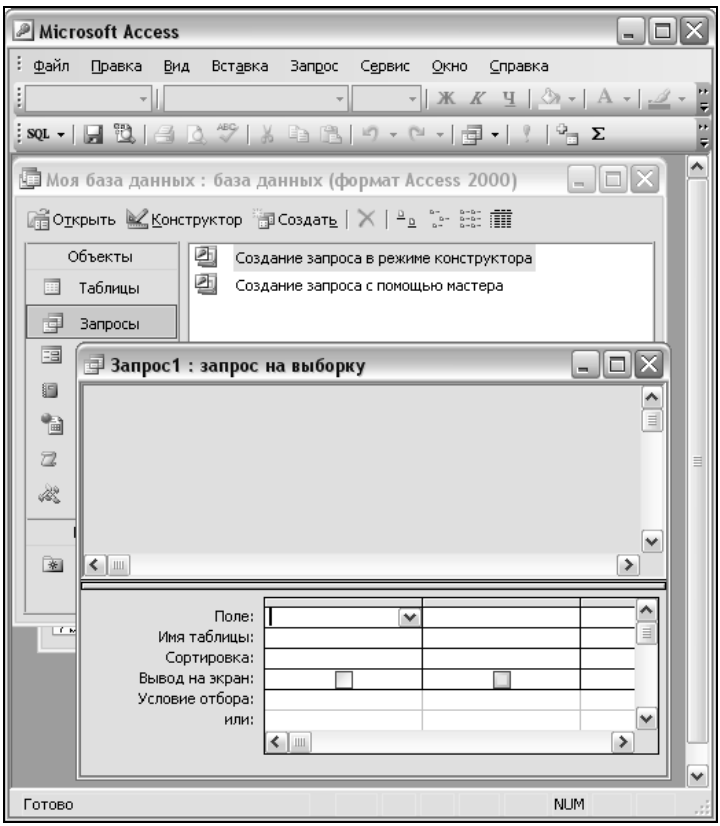


Рис. В7. Окно для ввода SQL-выражений в режиме конструктора

В меню **Вид** главного окна Access выберите команду **Режим SQL** или щелкните правой кнопкой мыши на верхней области окна **Запрос1: запрос на выборку** и в раскрывшемся контекстном меню выберите **Режим SQL**. В результате окно запросов примет вид, как показано на рис. В8. Это текстовый редактор, в котором можно вводить и редактировать выражения на языке SQL. При создании нового запроса в области ввода появляется строка "`SELECT;`", т. е. ключевое слово запроса, на выборку данных. Если вам необходимо выбрать данные, то следует дописать недостающие элементы SQL-выражения. Для создания других запросов (например, модификации данных, создания таблиц и др.) следует удалить ключевое слово `SELECT` и написать вместо него требуемый оператор.

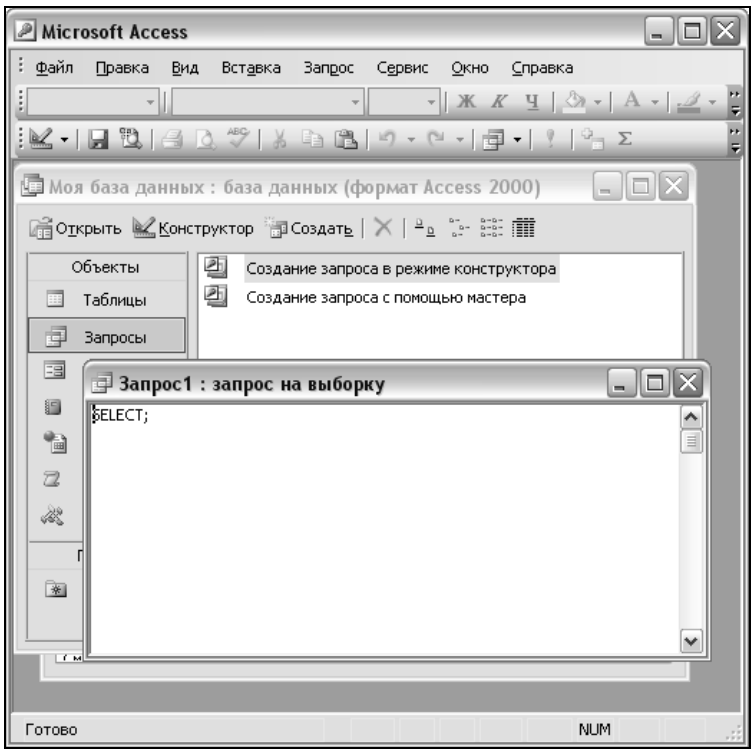


Рис. В8. Окно для ввода SQL-выражений в режиме SQL

Созданные запросы сохраняются в базе данных под именами, задаваемыми пользователем. При закрытии редактора запросов появляется предложение сохранить созданный запрос. В случае положительного ответа открывается диалоговое окно, в котором можно ввести имя запроса. Если выражение запроса было задано корректно, то его имя появится в списке запросов в окне **Моя база данных: база данных**. В противном случае появится сообщение об ошибке.

Чтобы отредактировать созданный ранее запрос, достаточно щелкнуть правой кнопкой мыши на его имени и в контекстном меню выбрать **Конструктор**.

Чтобы выполнить запрос, достаточно сделать двойной щелчок на его имени в списке запросов или в контекстном меню, раскрываемом щелчком правой кнопкой мыши на имени запроса, выбрать команду **Открыть**.

Глава 1

Основы реляционных баз данных

Программные средства работы с реляционными базами данных, в отличие от многих других, базируются на довольно строгой теории отношений, которая, в свою очередь, основана на теории множеств. Чтобы понять суть задачи создания реляционной базы данных, а также операций над данными, достаточно рассмотреть на теоретическом (абстрактном) уровне всего лишь несколько основных положений теории отношений. Это быстрее и, в конечном счете, лучше, как мне представляется, чем изучать большое количество конкретных примеров и ситуаций.

Таблицы, из которых состоит любая реляционная база данных, представляют собой некоторые отношения, а отношения являются не чем иным, как множествами. Все запросы к базе данных, направленные на извлечение из нее нужных записей, интерпретируются как инструкции по выполнению тех или иных операций, являющихся, в конечном счете, операциями алгебры множеств и исчисления предикатов.

В этой главе мы рассмотрим основные вопросы, относящиеся к реляционным базам данных, с более общей точки зрения. Зато все последующие главы будут посвящены конкретным приемам использования языка SQL.

1.1. Множества

Множество является настолько общим понятием, что не имеет определения, которое можно было бы выразить в еще более

общих и простых понятиях. Поэтому в математической теории множеств определение (в математическом смысле) понятия множества отсутствует. Вместе с тем, в его основе лежит некий образ, который Георг Кантор (один из создателей теории множеств) описал как собрание определенных и различных между собой объектов интуиции или интеллекта, мыслимое как единое целое. Эти объекты называются элементами множества.

Примечание

В математике многое не так, как в других науках. Если мы обратим внимание на теорию, скажем, "блям-блямчиков", то с точки зрения математической культуры совсем не важно, что такое "блям-блямчики". Иначе говоря, строгое в математическом смысле определение "блям-блямчиков" не входит в теорию. "Блям-блямчики" — лишь наименование предмета теории, а ее содержание — выявление свойств и отношений "блям-блямчиков". Только изучив эту теорию, мы получим более или менее полное представление о том, что такое "блям-блямчики". Это представление возникает из понимания того, каким образом объект изучения и его компоненты связаны между собой и объектами окружающего мира. Так, например, К. Поппер (исследователь логических аспектов языка) удачно заметил, что смысл слов определяется через их использование в речи (текстах). Тем не менее, любая книга по теории, в том числе и математической, обычно начинается с пространных описаний предмета и стремится дать более четкое его "определение". Однако это по существу лишь стимуляция интуиции читателя, направленная на то, чтобы он настроился на предмет изучения и преодолел возможные психологические барьеры, возникающие при встрече с новым. По большому счету стимуляция интуиции не имеет непосредственного отношения к самой теории. Другими словами, интерпретация теории не является частью этой теории. Это задача ее применения к жизни.

Существенным для канторовского понимания множества является то, что собрание объектов само по себе рассматривается как один объект. На природу объектов, которые могут входить в множество, не накладывается никаких ограничений. Это могут быть числа, наборы символов, люди, атомы и т. п.

Множества могут быть конечными и бесконечными. *Конечные* множества содержат элементы, которые можно сосчитать или перечислить. Это означает, во-первых, что имеется *принципиальная*

возможность сопоставить каждому элементу множества некоторое натуральное число (1, 2, 3, ...), и, во-вторых, этот пересчет когда-нибудь закончится. Так, например, бесконечное множество всех целых чисел можно начать перечислять, но этот процесс никогда не закончится: для любого целого числа можно создать следующее, прибавив к нему 1. А множество всех действительных чисел, также являющееся бесконечным, даже начать перечислять невозможно. Интересно, что этот факт был установлен Кантором только лишь в конце XIX века и произвел на математиков ошеломляющее впечатление.

Существуют и конечные множества, перечислить элементы которых *практически* невозможно. Я хочу обратить ваше внимание на слово "практически". Представителем таких множеств может служить, например, множество всех погибших в Куликовской битве, а также множество всевозможных последовательностей нулей и единиц длиной 100. Мы уверены, что эти множества конечны. Количество погибших не может быть бесконечным, но мы либо не знаем, как их перечислить, либо решение этой задачи требует неимоверно больших затрат ресурсов. Количество последовательностей из нулей и единиц длиной 100 равно 2^{100}. Это число больше количества атомов в видимой части вселенной, и, следовательно, не хватит жизни многих поколений людей, чтобы перечислить (или сгенерировать) такие последовательности даже с помощью самого быстродействующего компьютера. Однако в математике практическая и теоретическая невозможности чего-либо — это различные вещи. Практические трудности игнорируются математиками, а выявление принципиальной недостижимости (фундаментальных пределов) чего-либо является для них чрезвычайно ценным результатом. Математическая теория множеств рассматривается сейчас как фундамент всей математики, а также как пространство и средство изучения бесконечного. Это очень интересно, но данная книга не об этом.

В компьютерной практике и, в частности, в базах данных мы имеем дело с конечными множествами, хотя иногда и очень большими. Поэтому мы застрахованы от логических неопределенностей и тупиков, которые могут встретиться в области бес-

конечных множеств. Однако нам предстоит преодолевать практические трудности, связанные с очень большими конечными множествами, но это задача скорее технологии, чем математики. Теперь обратимся к формальным аспектам конечных множеств, чтобы кратко и ясно представить себе то, с чем в большом многообразии частностей мы имеем дело в жизни.

Итак, *множество* является тем, что имеет элементы, а *элементы* — это то, что либо входит, либо не входит в данное множество. В множестве не может быть двух или более одинаковых элементов, а порядок расположения элементов в множестве не имеет никакого значения. Заметим, что этим множества отличаются от массивов — объектов языков программирования и компьютерных программ. Массивы в программах так или иначе упорядочены и могут содержать одинаковые элементы. Вместе с тем, ничто не мешает рассматривать массивы как модель (некоторое приближение) множеств. Важно лишь помнить, что массив — это некоторое программное воплощение идеи конечного множества, имеющее дополнительные свойства, которые не обязано иметь множество.

Обозначим некоторое множество символом A, а все элементы, входящие в это множество, — символами $a_1, a_2, \ldots, a_N$. Тогда запись $A = \{a_1, a_2, \ldots, a_N\}$ означает, что множество A содержит только те элементы $a_1, a_2, \ldots, a_N$, которые указаны в фигурных скобках. Напомню, что все элементы различны, а их порядок в фигурных скобках не имеет значения.

Какой-либо элемент x может принадлежать или не принадлежать множеству A. Чтобы указать, что элемент x принадлежит множеству A, пишут $x \in A$, а если x не принадлежит A, то пишут $x \notin A$. Множество может и не содержать элементов, тогда его называют пустым и обозначают как $\varnothing$.

Простейший способ для определения конкретного множества состоит в том, чтобы явно указать все элементы, принадлежащие этому множеству. Это так называемый *экстенсиональный* способ определения множества. Если количество элементов, входящих в множество A, невелико, то этот способ вполне применим на практике: достаточно написать выражение вида $A = \{a_1, a_2, \ldots, a_N\}$.

Однако при достаточно больших множествах этот способ не подходит. Тогда используют так называемый *интенсиональный* (неявный) способ определения множеств. Он основан на использовании некоторой функции (алгоритма), которая определяет для каждого элемента, принадлежит ли он данному множеству или нет. Пусть для некоторого множества A определена такая функция $P_A(x)$. Если вместо переменной x подставить в ее выражение конкретный элемент a, то результатом вычисления $P_A(a)$ будет либо ИСТИНА (`true`), либо ЛОЖЬ (`false`), в зависимости от того, принадлежит или нет элемент a множеству A. Таким образом, функция $P_A(x)$ принимает в качестве аргументов элементы из некоторой области (универсума) и возвращает одно из двух значений, которые мы здесь обозначаем как ИСТИНА и ЛОЖЬ (т. е. функция $P_A(x)$ является двузначной). С другой стороны, мы можем сами определить какую-нибудь двузначную функцию $Q(x)$, принимающую в качестве аргумента значения из некоторого универсума. Тогда эта функция будет определять некоторое множество, а именно множество всех тех элементов универсума, для которых эта функция возвращает значение ИСТИНА. Указанные двузначные функции в математической логике называются *предикатами*.

В качестве примера рассмотрим выражение $x < 5$. Это типичное выражение сравнения. Здесь через x обозначена переменная, значения которой берутся из множества всех действительных чисел. Тогда это выражение может принимать значения ИСТИНА или ЛОЖЬ в зависимости от того, какое значение будет подставлено вместо x. Вопреки сложившейся математической практике мы могли бы записать данное выражение и в таком виде: $<5(x)$. Здесь <5 — обозначение предиката "быть меньше 5". Данный предикат определяет множество всех действительных чисел, которые меньше числа 5. Обратите внимание, что мы некоторым конечным образом определили бесконечное множество.

Аналогично мы можем определить с помощью предиката $Красный(x)$ множество всех красных элементов. Разумеется, мы должны иметь алгоритм вычисления данного предиката, т. е. алгоритм определения, является ли предъявленный элемент x красным или нет.

В явном виде (экстенсионально) пустое множество обозначается как { }. Интенсионально пустое множество определяется через некий предикат, который ложен для всех элементов универсума.

Примечание

Предикаты в математической логике являются двузначными функциями. Однако как в математике, так и на практике нередко встречаются ситуации, к которым больше подходит многозначная и, в частности, трехзначная логика. Так, для некоторого элемента *x* результатом вычисления значений некоторого предиката *P*(*x*) может быть не только ИСТИНА или ЛОЖЬ, но и некоторое третье значение, например, `NULL`. Последнее интерпретируется как "не определено" или "не известно". Такая логика в том или ином виде принята при поддержке современных баз данных. Например, столбец таблицы базы данных может содержать помимо наборов каких-то символов особое значение `NULL`, которое понимается как "еще не введенное", "пока неизвестное" и т. п. Однако замечу, что понимание трехзначной логики лучше усваивается при понимании классической двузначной логики.

Между множествами определяется *отношение включения*. Так, множество A включается в множество B (это утверждение записывается как $A \subseteq B$), если каждый элемент множества A принадлежит и множеству B. В этом случае говорят, что множество A является подмножеством множества B. Два множества равны ($A = B$), если $A \subseteq B$ и $B \subseteq A$, т. е. оба множества включаются друг в друга и, следовательно, состоят из одних и тех же элементов. Очевидно, что любое множество является своим подмножеством, т. е. для любого множества A выполняется отношение $A \subseteq A$. Однако в общем случае из того, что $A \subseteq B$, еще не следует, что $B \subseteq A$.

Пустое множество $\varnothing$ включается в любое множество, т. е. для любого множества A имеет место включение $\varnothing \subseteq A$. Этот факт, хоть он и очевиден, можно доказать. Предположим, что $\varnothing \subseteq A$ ложно. Но это может быть только в том случае, когда существует некий элемент x, такой, что $x \in \varnothing$ и $x \notin A$. Однако это не возможно, т. к. пустое множество $\varnothing$ не имеет элементов по определению.

Не следует смешивать отношение принадлежности $\in$ между элементами и множествами и отношение включения $\subseteq$ между множествами. Так, например, если $x \in A$ и $A \in B$, то это еще не означает, что $x \in B$. Другой пример: если $A = \{B, a, c\}$ и $B = \{a, d\}$, то $B \in A$, но не верно, что $B \subseteq A$. Верно лишь, что $\{B\} \subseteq A$, т. е. множество, состоящее из элемента B, является подмножеством множества A. Иначе говоря, элементы некоторого множества A сами могут быть множествами, но элементы последних не обязательно являются элементами множества A. Так, в рассматриваемом примере $a \in A$, $a \in B$, $d \in B$, но $d \notin A$. Заметим также, что элемент и множество, содержащее единственный этот элемент (т. е. x и $\{x\}$), — разные понятия.

Над множествами можно выполнять различные операции: объединение ($\bigcup$), пересечение ($\bigcap$), вычитание ($-$) и декартово произведение ($\times$).

Объединение множеств A и B есть множество, обозначаемое как $A \bigcup B$, которое содержит все элементы множества A и все элементы множества B. Пусть, например, $A = \{a, b, c, x, y\}$, $B = \{a, b, x, y, z\}$. Тогда $A \bigcup B = \{a, b, c, x, y, z\}$. Напомню, что в любом множестве все элементы должны быть различны. Другими словами, множество $A \bigcup B$ содержит только такие элементы, которые принадлежат множеству A или множеству B (возможно, и обоим одновременно). В частности, $A \bigcup \varnothing = A$.

Пересечение множеств A и B есть множество, обозначаемое как $A \bigcap B$, которое содержит все элементы, принадлежащие как множеству A, так и множеству B. Если таких элементов нет, то пересечение множеств пусто ($A \bigcap B = \varnothing$). Например, пусть $A = \{a, b, c, x, y\}$, $B = \{a, b, x, y, z\}$. Тогда $A \bigcap B = \{a, b, x, y\}$. В частности, $A \bigcap \varnothing = \varnothing$.

Вычитание из множества A множества B дает в результате множество, называемое *разностью* этих множеств, и обозначается как $A - B$. Это множество содержит все элементы множества A, которые не принадлежат множеству B. Например, пусть $A = \{a, b, c, d, x, y\}$, $B = \{a, b, x, y, z\}$. Тогда $A - B = \{c, d\}$. В частности, $A - \varnothing = A$. Если множества A и B не пересекаются

(т. е. $A \cap B = \varnothing$), то $A - B = A$. Множество $A - B$ называют также *дополнением* множества B до множества A.

Декартово произведение множеств будет подробно рассмотрено далее в *разд. 1.2*.

Обозначим через I множество всех элементов, которое называется *универсумом*. Предполагается, что любой элемент, с которым мы имеем дело, принадлежит универсуму. Тогда любое множество A является подмножеством универсума: $A \subseteq I$. Разность $I - A$ называют дополнением множества A до универсума (или просто дополнением) и обозначают через $\overline{A}$.

Примечание

В математической литературе для обозначения операции вычитания множеств часто используют обратный слэш (\).

Далее приведены основные равенства, используемые при преобразовании выражений с несколькими операциями над множествами:

- $\overline{\overline{A}} = A$ — закон двойного дополнения (отрицания);
- $A \cup \overline{A} = I$ — закон исключения третьего;
- $A \cap \overline{A} = \varnothing$ — закон противоречия;
- $A \cup \varnothing = A$;
- $A \cap I = A$;
- законы Моргана:

$$\overline{(A \cap B)} = \overline{A} \cup \overline{B};$$

$$\overline{(A \cup B)} = \overline{A} \cap \overline{B};$$

- законы коммутативности (перестановки):

$$A \cup B = B \cup A;$$

$$A \cap B = B \cap A;$$

- законы ассоциативности (группировки):

$$A \cup (B \cup C) = (A \cup B) \cup C;$$

$$A \cap (B \cap C) = (A \cap B) \cap C;$$

- законы дистрибутивности (сочетания операций):

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C);$$
$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C).$$

Чтобы доказать равенство двух множеств, обычно показывают, что первое множество включается во второе, а второе — в первое. Кроме того, в самом доказательстве относительно множеств используют предложения типа "пусть некоторый элемент x принадлежит такому-то множеству". Поскольку рассмотрение ситуации начинается с достаточно произвольного элемента, то в итоге мы получаем утверждение относительно не только этого элемента, но и всего множества, которому этот "произвольный" элемент принадлежит. Это общие правила доказательства равенств и включения множеств.

В качестве упражнения докажем одно из приведенных ранее равенств, а именно один из законов дистрибутивности:

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C).$$

Для этого достаточно доказать два включения:

$$A \cup (B \cap C) \subseteq (A \cup B) \cap (A \cup C)$$

и

$$(A \cup B) \cap (A \cup C) \subseteq A \cup (B \cap C).$$

1. Доказательство включения $A \cup (B \cap C) \subseteq (A \cup B) \cap (A \cup C)$.

   Выберем произвольный элемент $x \in A \cup (B \cap C)$. Тогда $x \in A$ или $x \in B \cap C$. Если $x \in A$, то верно и то, что $x \in A \cup B$ и $x \in A \cup C$, а следовательно, x есть элемент пересечения этих множеств. Если же $x \in B \cap C$, то $x \in B$ и $x \in C$. Следовательно, и в этом случае $x \in A \cup B$ и $x \in A \cup C$. Таким образом, $x \in (A \cup B) \cap (A \cup C)$. Поскольку мы рассматривали произвольный элемент $x \in A \cup (B \cap C)$, то требуемое включение множеств доказано.

2. Доказательство включения $(A \cup B) \cap (A \cup C) \subseteq A \cup (B \cap C)$.

   Пусть $x \in (A \cup B) \cap (A \cup C)$. Тогда $x \in A \cup B$ и $x \in A \cup C$. Следовательно, $x \in A$ или же $x \in B$ и $x \in C$, т. е. $x \in A \cup (B \cap C)$.

На этом доказательство равенства двух множеств заканчивается.

Итак, мы рассмотрели понятие множества и основные операции над множествами. Теперь кратко рассмотрим понятия высказывания и предиката. Высказывание, с точки зрения математической логики, не зависимо от его информационного содержания и смысла, либо истинно, либо ложно. Например, высказывание "дважды два равно четырем" считается истинным, а высказывание "все птицы могут летать" — ложным (поскольку есть, например, страус, который не летает). Однако мы можем составить выражение, в котором будут фигурировать некие изначально неопределенные объекты, вроде переменных в математических выражениях. Например, выражение $x<10$ может быть как истинным, так и ложным в зависимости от того, что имеется в виду под переменной x (т. е. каково ее значение). Выражение "нечто является красным" также не может принять конкретного значения истинности, пока мы не определим, что понимается под словом "нечто". Так, если в выражение $x<10$ подставить число 3, то оно примет вид $3<10$ и будет истинным. Если же в это выражение подставить число 12345, то получится ложное высказывание. Выражения с переменными, возвращающие одно из двух возможных значений (ИСТИНА или ЛОЖЬ) после подстановки вместо переменных конкретных значений, называются *предикатами*. Пусть имеется некоторое выражение с переменными $x1, x2, \ldots, xn$. Обозначим его как $P(x1, x2, \ldots, xn)$, подчеркивая тем самым, что это просто некоторая функция от n аргументов. Особенностью этой функции является только то, что она двузначная — принимает одно из двух возможных значений (ИСТИНА или ЛОЖЬ).

Как в обычной речи, так и в математической логике часто используются так называемые общеутвердительные и частноутвердительные высказывания. Предложение "все люди млекопитающие" является примером общеутвердительного высказывания, а предложение "некоторые люди — женщины" — пример частноутвердительного высказывания. Заметим, что оба эти высказывания являются истинными. Разумеется, мы могли бы придумать и какие-нибудь ложные высказывания. В этих высказываниях, как нетрудно заметить, используются слова "все" и "некоторые", иг-

рающие определенную функциональную роль. Они указывают, в насколько большой области значений переменных предикат является истинным. Поэтому эти слова называются *кванторами*. Слово "все" означает, что предикат истинен для всех (каждого, любого) элемента универсума, а слово "некоторые" — по крайней мере для какого-нибудь одного элемента. В математической логике эти кванторы обозначаются через $\forall$ и $\exists$ и называются *кванторами всеобщности* и *существования* соответственно.

Выражение $\forall x P(x)$ понимается как высказывание, что для всех элементов универсума предикат $P(x)$ истинен. Выражение $\exists x P(x)$ означает, что предикат $P(x)$ истинен для некоторого элемента универсума. Высказывания $\forall x P(x)$ и $\exists x P(x)$ уже не зависят от значения x и могут быть как истинными, так и ложными. Например, высказывание $\forall x(\sin(x) = 1)$ ложно, а высказывание $\exists x(\sin(x) = 1)$ истинно.

К предикатам, как и к высказываниям, применимы операторы отрицания ($\bar{}$), конъюнкции ($\&$) и дизъюнкции ($\vee$). В результате применения этих операторов получается совсем другой предикат.

Оператор отрицания изменяет значение предиката на противоположное. Так, если предикат $P(x)$ истинен, то $\bar{P}(x)$ ложен, и наоборот.

Оператор конъюнкции выполняет роль логического союза И, а дизъюнкции — логического союза ИЛИ. Так, для двух предикатов $P(x)$ и $Q(x)$ предикат $P(x) \& Q(x)$ истинен только тогда, когда истинны одновременно оба предиката; предикат $P(x) \vee Q(x)$ истинен только тогда, когда истинен хотя бы один из предикатов.

Из вышеизложенного следует, что выражение $\forall x P(x)$ эквивалентно выражению $P(a1) \& P(a2) \& ... \& P(an)$, где $a1$, $a2$, ... , an — элементы универсума. Аналогично, выражение $\exists x P(x)$ эквивалентно выражению $P(a1) \vee P(a2) \vee ... \vee P(an)$. Нетрудно заметить, что если истинно высказывание $\forall x P(x)$, то истинно и высказывание $\exists x P(x)$, хотя обратное в общем случае не верно (если верно частное утверждение, то соответствующее общее утверждение может быть и не верным). Например, утверждение

"некоторые люди умны" истинно, но утверждение "все люди умны" скорее ложно, чем истинно.

До сих пор мы рассматривали предикаты только от одной переменной. Однако предикаты могут содержать и несколько переменных. Например, предикат $x^2 - y = y^2 - x$ представляет собой равенство двух числовых выражений с двумя переменными — x и y. Следующее выражение является истинным высказыванием:

$$\forall x \exists y (x^2 - y = y^2 - x).$$

Изменение типа квантора в выражении может изменить его значение истинности. Так, например, следующее высказывание ложно:

$$\exists x \forall y (x^2 - y = y^2 - x).$$

Однако для любого предиката $P(x, y)$ если истинно высказывание $\exists y \forall x P(x, y)$, то истинно и высказывание $\forall x \exists y P(x, y)$. Иначе говоря, перестановка кванторов всеобщности и существования вместе с соответствующими им переменными не изменяет значения истинности высказывания.

Нетрудно заметить аналогию между операциями над множествами ($-$, $\cup$, $\cap$) и над предикатами ($\neg$, $\&$, $\vee$). Эта аналогия обусловлена тем, что, во-первых, множества можно задать не только явным образом путем перечисления их элементов, но и через предикаты и, во-вторых, любой предикат задает множество всех тех элементов, для которых он истинен.

Пусть $P(x)$ — некоторый предикат. Тогда справедливы следующие равенства:

$$\overline{\forall x P(x)} = \exists x \overline{P(x)};$$
$$\overline{\exists x P(x)} = \forall x \overline{P(x)}.$$

Пусть, например, P обозначает свойство "быть красным". Тогда первое из двух указанных равенств означает, что высказывание "не верно, что все x красные" эквивалентно высказыванию "некоторые x не красные". Второе равенство утверждает эквивалентность высказываний "не верно, что есть красный объект" и "все объекты не красные".

1.2. Отношения

1.2.1. Общие сведения

В предыдущем разделе мы рассмотрели понятия множества и предиката. Любой предикат определяет некоторое множество элементов из какого-то универсума, а именно множество тех элементов, для которых он истинен. Таким образом, предикат можно интерпретировать как двузначную функцию, определяющую, обладает ли элемент неким свойством, например, свойством принадлежности данному множеству.

Предикат $P(x)$ от одной переменной x определяет множество элементов, при подстановке которых вместо x этот предикат становится истинным высказыванием. Однако предикаты могут содержать и несколько переменных: $P(x1, x2, \ldots, xn)$. Такие предикаты определяют множество элементов вида $(x1, x2, \ldots, xn)$, т. е. последовательностей элементов универсума. Такие последовательности еще называют кортежами. Например, предикат $x^2 + y^2 = 1$ определяет множество точек (их координат), лежащих на окружности единичного радиуса с центром в начале системы координат.

В общем случае элементы $x1, x2, \ldots, xn$ кортежей могут выбираться не из одного множества-универсума, а из разных. Пусть переменные $x1, x2, \ldots, xn$ в предикате $P(x1, x2, \ldots, xn)$ пробегают значения из множеств $A1, A2, \ldots, An$ соответственно. Тогда говорят, что этот предикат определяет отношение между элементами множеств $A1, A2, \ldots, An$ или просто между множествами $A1, A2, \ldots, An$. Это отношение обозначим через $P(A1, A2, \ldots, An)$, т. е. так же, как и предикат.

Итак, отношение есть множество, элементами которого являются кортежи, которые, в свою очередь, состоят из элементов других множеств. При этом между любым кортежем отношения и самим отношением имеет место отношение принадлежности $\in$, но не включения $\subseteq$. Кортеж представляет собой не множество, а упорядоченную последовательность элементов.

Рассмотрим в качестве примера отношение ОТЦЫ_И_ДЕТИ. Это отношение между множеством МУЖЧИНЫ и множеством ЛЮДИ:

ОТЦЫ_И_ДЕТИ (МУЖЧИНЫ, ЛЮДИ). Мы не будем сейчас определять предикат, задающий это отношение. Алгоритм определения отцовства может быть различным. Например, можно ограничиться только опросом всех мужчин на предмет, каких детей они имеют, провести генетическую экспертизу и т. п. Как бы то ни было, в результате мы получим множество всех кортежей вида $(x1, x2)$, в которых через $x1$ и $x2$ обозначены люди, такие, что $x1$ является отцом для $x2$. В качестве имен людей следует выбрать их уникальные идентификаторы. Фамилия, имя и отчество для этой цели вряд ли подойдут. Возможно, окажется достаточным использовать дополнительные паспортные данные, отпечатки пальцев или снимок радужной оболочки глаза. Очевидно, что множества отцов и женщин включаются в множество людей, но ни одна женщина не принадлежит множеству отцов. Поэтому в рассматриваемом отношении ОТЦЫ_И_ДЕТИ не будет ни одного кортежа, в котором на первом месте стояло бы имя женщины. Кроме того, ни один человек не может быть отцом для самого себя. Поэтому в отношении не будет ни одного кортежа, в котором первый и второй элемент совпадают. Эти и, возможно, другие особенности, характеризующие отношение, называются ограничениями его целостности.

Задавая отношение ОТЦЫ_И_ДЕТИ, мы можем, по крайней мере теоретически, поступить следующим образом. Сначала возьмем множество всевозможных кортежей $(x1, x2)$, в которых $x1 \in$ МУЖЧИНЫ и $x2 \in$ ЛЮДИ, а затем выберем из этого множества только те кортежи, в которых $x1$ является отцом для $x2$. Полученное множество и будет представлять собой интересующее нас отношение ОТЦЫ_И_ДЕТИ.

Множество всех возможных кортежей $(x1, x2)$, в которых $x1 \in A1$ и $x2 \in A2$, называется *декартовым произведением множеств* $A1$ и $A2$ и обозначается как $A1 \times A2$. Количество элементов в декартовом произведении равно произведению количеств элементов в множествах $A1$ и $A2$. Например, если $A1 = \{a, b, c\}$ и $A2 = \{x, y\}$, то:

$$A1 \times A2 = \{(a, x), (a, y), (b, x), (b, y), (c, x), (c, y)\}.$$

Рассмотрим еще один пример. Пусть X, Y — множества действительных чисел, составляющих отрезки числовых осей прямо-

угольной системы координат, тогда множество $X \times Y$ можно представить как множество точек прямоугольника с абсциссами из множества X и ординатами из Y. Кривая, нарисованная в этом прямоугольнике, задает некоторую зависимость (отношение) $y = f(x)$ между X и Y, которую можно представить множеством пар координат точек, лежащих на кривой. Очевидно, это множество является подмножеством декартового произведения $X \times Y$ и может быть представлено в виде двухстолбцовой таблицы. В одном столбце этой таблицы записываются значения x, а в другом — соответствующие значения y функциональной зависимости $y = f(x)$.

Декартово произведение для n множеств $A1, A2, \ldots, An$ обозначается как $A1 \times A2 \times \ldots \times An$ и состоит из всевозможных кортежей вида $(x1, x2, \ldots, xn)$, в которых $x1 \in A1$, $x2 \in A2, \ldots, xn \in An$. Например, пусть товары на складе имеют характеристики, выбираемые из множеств НАИМЕНОВАНИЕ, КОЛИЧЕСТВО, ЦЕНА и ПОСТАВЩИК. Тогда таблица, содержащая сведения о всех товарах на складе, представляет отношение между указанными характеристиками, а все ее строки — кортежи, принадлежащие некоторому подмножеству декартового произведения:

НАИМЕНОВАНИЕ × КОЛИЧЕСТВО × ЦЕНА × ПОСТАВЩИК.

Итак, некоторое отношение $R(A1, A2, \ldots, An)$ есть подмножество декартового произведения $A1 \times A2 \times \ldots \times An$, т. е. имеет место включение $R(A1, A2, \ldots, An) \subseteq A1 \times A2 \times \ldots \times An$. При $n = 1$ $R(A1)$ представляет собой просто подмножество множества $A1$ и называется *унарным*. При $n = 2$ отношение называется *бинарным*. Так, отношения равенства, порядка и сходства являются примерами бинарных отношений. При $n = 3$ отношение называется *тернарным*. Нередко n-арные отношения называют также n-местными.

1.2.2. Способы представления отношений

К представлениям отношений мы приходим, когда требуется наглядность или определенное удобство для хранения данных и манипулирования ими. Рассмотрим несколько вариантов.

В вырожденном случае унарных отношений мы имеем дело с обычными множествами, которые естественным образом представляются списком своих элементов или же одностолбцовой таблицей, в каждой строке которой указан некоторый элемент множества. При этом не следует забывать, что множество не предполагает никакого порядка элементов. Поэтому два списка, содержащие одни и те же элементы, но упорядоченные различным образом, являются эквивалентными с точки зрения теории отношений. Иначе говоря, они представляют одно и то же отношение.

Бинарные (двухместные) отношения можно представить в виде графов, а также двухстолбцовых таблиц. В графовом представлении каждой вершине соответствует некоторый элемент одного из двух множеств. От одной вершины к другой проводится стрелка, если первый элемент находится в рассматриваемом отношении со вторым. Например, отношение достижимости населенных пунктов из данного пункта с помощью авиарейсов является бинарным отношением между множеством АВИАРЕЙСЫ и множеством ПУНКТЫ_НАЗНАЧЕНИЯ. Мы проводим из вершины графа достижимости, соответствующей, например, рейсу Р123, стрелку в вершину "Москва", если этим рейсом можно добраться до данного пункта. Очевидно, информацию о достижимости пунктов авиарейсами можно представить и в виде двухстолбцовой таблицы. В первом столбце указываются авиарейсы, а во втором — пункты назначения. В каждой строке указывается, каким авиарейсом и куда можно долететь. Каждая строка таблицы представляет собой кортеж отношения.

Тернарные (трехместные) отношения можно представить в виде графов, а также трехстолбцовых таблиц. В графовом представлении элементами множеств помечаются не только вершины, но и стрелки. Пусть, например, отношение содержит сведения не только о том, каким авиарейсом в какие пункты можно попасть, но и за какое время. Время полета можно указывать непосредственно около стрелки, соединяющей номер авиарейса и пункт назначения. Представление этого отношения с помощью таблицы очевидно.

Бинарные и тернарные отношения, атрибуты которых принимают значения из числовых множеств, представляются еще и в виде графиков — кривых и поверхностей в той или иной системе координат.

Отношения, арность которых больше 3, представить графически проблематично. Даже если как-то исхитриться это сделать, то результат вряд ли будет наглядным. Остается табличный способ, как достаточно универсальный для представления отношений.

При табличном представлении каждому кортежу отношения взаимооднозначно соответствует строка (запись) в таблице. Но не всякая таблица представляет некоторое отношение. *Таблица* — это прямоугольная сетка, в ячейках которой может находиться все, что угодно. При этом различные строки этой таблицы могут содержать в соответствующих столбцах одинаковые данные. Другими словами, таблицы могут содержать идентичные строки. В этом случае совокупность всех строк таблицы не является множеством, а значит, не представляет собой отношения. Напомню, что множество — это совокупность различных элементов. Таким образом, любое отношение (в определенном ранее смысле) можно представить в виде таблицы, но обратное утверждение, вообще говоря, не верно — не всякая таблица представляет какое-нибудь отношение.

При представлении отношения явным образом в виде таблицы, ее нельзя рассматривать просто как вместилище данных, в которое можно добавлять новые записи или удалять старые. Добавление, удаление и изменение данных в такой таблице приводит либо к другому отношению, либо к никакому отношению (если в таблице окажутся одинаковые записи).

Табличное представление отношений — это чрезвычайно плодотворный технологический прием, обеспечивший создание и широкое практическое применение баз данных. То обстоятельство, что отношения — просто множества, позволило изучать проблемы реляционных баз данных на теоретическом уровне, в мире математики (алгебра множеств и исчисление предикатов), а не только в мире технологии. В результате разработка СУБД, кон-

кретных баз данных, языков манипулирования данными, в том числе и SQL, водрузилась на твердый теоретический фундамент.

Далее мы будем рассматривать отношения, представляя их в виде таблиц. Введем несколько терминов, которые обычно применяются в теории отношений. Так, говоря о некотором отношении $R(A1, A2, \ldots, An)$, мы имеем в виду следующее.

- Отношение имеет имя R. Например, сведения о товарах, хранящихся на складе, образуют некоторое отношение, которому можно дать имя СКЛАД.
- Множества $A1, A2, \ldots, An$, для которых определено отношение R, имеют различные имена, называемые *атрибутами* отношения. Мы будем считать, что $A1, A2, \ldots, An$ — это атрибуты отношения. Совокупность всевозможных значений любого множества с именем Ai $(i = 1, 2, \ldots, n)$ называется *доменом* атрибута Ai. Заметим еще раз, что в отношении не может быть двух и более одинаковых атрибутов, хотя их домены могут быть и одинаковыми (в смысле равенства множеств). Например, отношение СКЛАД может быть определено для атрибутов НАИМЕНОВАНИЕ, КОЛИЧЕСТВО, ЦЕНА и ПОСТАВЩИК. Структуру этого отношения можно записать так: СКЛАД (НАИМЕНОВАНИЕ, КОЛИЧЕСТВО, ЦЕНА, ПОСТАВЩИК). Каждый атрибут имеет свой домен — множество возможных значений. Например, доменом атрибута КОЛИЧЕСТВО может быть множество неотрицательных чисел. При табличном представлении отношения каждому атрибуту соответствует заголовок столбца, а домену — множество значений в этом столбце.
- Порядок перечисления атрибутов в структуре отношения не имеет значения. Иначе говоря, перестановка атрибутов местами оставляет само отношение прежним, хотя вид таблицы, представляющей это отношение, меняется. Например, отношения СКЛАД (НАИМЕНОВАНИЕ, КОЛИЧЕСТВО, ЦЕНА, ПОСТАВЩИК) и СКЛАД (ПОСТАВЩИК, НАИМЕНОВАНИЕ, КОЛИЧЕСТВО, ЦЕНА) считаются эквивалентными.
- Элементами отношения являются кортежи — последовательности значений атрибутов отношения. В отношении не может

быть двух и более одинаковых кортежей, а порядок расположения кортежей не имеет значения. При табличном представлении отношения каждому кортежу соответствует строка таблицы. Строки таблицы мы будем называть *записями*.

1.2.3. Операции над отношениями

Коль скоро отношение — это множество (а именно множество кортежей или, другими словами, записей), то к ним применимы все теоретико-множественные операции, рассмотренные ранее. Однако в реляционной теории особую роль играют специальные операции, лежащие в основе языка SQL:

- селекция;
- проекция;
- естественное соединение.

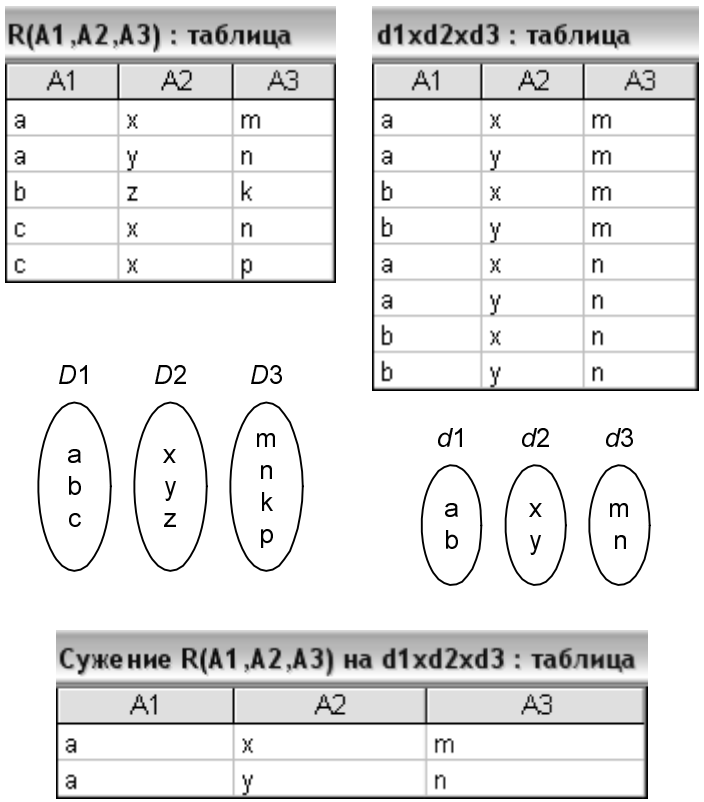
Эти операции выражаются некоторым образом через обычные операции над множествами, такие как объединение, пересечение, вычитание и декартово произведение. Рассмотрим их более подробно.

Селекция

Операция селекции (выборки, ограничения) отношения выделяет из него некоторое подмножество кортежей, которые удовлетворяют некоторому условию. При работе с базами данных выборка из всей имеющейся совокупности только требуемых данных — это наиболее часто применяемая операция. Конкретное определение этой операции зависит от вида условия. Одной из частных разновидностей операции селекции является операция сужения отношения, которую мы здесь и рассмотрим.

Пусть имеется некоторое отношение $R(A1, A2, \ldots, An)$. Обозначим домены атрибутов $A1, A2, \ldots, An$ этого отношения через $D1, D2, \ldots, Dn$ соответственно. Напомню, что домен атрибута — это множество его допустимых значений. Далее обозначим через $d1, d2, \ldots, dn$ какие-нибудь подмножества доменов $D1, D2, \ldots, Dn$ соответственно (т. е. $di \subset Di$, $i = 1, 2, \ldots, n$). Тогда *сужением* отношения $R(A1, A2, \ldots, An)$ на множество $d1 \times d2 \times \ldots \times dn$

называется отношение $R(A1, A2, \ldots, An) \bigcap d1 \times d2 \times \ldots \times dn$. По существу, данная операция просто выделяет из исходного отношения только те кортежи, в которых элементы (значения атрибутов) принадлежат указанным подмножествам соответствующих доменов. Заметим, что в частном случае, когда $di = Di$ для всех $i = 1, 2, \ldots, n$, сужение отношения $R(A1, A2, \ldots, An)$ на множество $d1 \times d2 \times \ldots \times dn$ равно исходному отношению $R(A1, A2, \ldots, An)$.



R(A1,A2,A3) : таблица

A1	A2	A3
a	x	m
a	y	n
b	z	k
c	x	n
c	x	p

d1xd2xd3 : таблица

A1	A2	A3
a	x	m
a	y	m
b	x	m
b	y	m
a	x	n
a	y	n
b	x	n
b	y	n

Сужение R(A1,A2,A3) на d1xd2xd3 : таблица

A1	A2	A3
a	x	m
a	y	n

Рис. 1.1. Пример сужения отношения

На рис. 1.1 в качестве примера показано некоторое отношение $R(A1, A2, A3)$, домены $D1$, $D2$ и $D3$ атрибутов $A1$, $A2$ и $A3$, неко-

торые подмножества $d1$, $d2$ и $d3$ этих доменов, декартово произведение $d1 \times d2 \times d3$ и результат сужения исходного отношения на множество кортежей $d1 \times d2 \times d3$.

Рассмотрим в качестве примера отношение ЗАРПЛАТА (СОТРУДНИК, ВЫПЛАТА), которое можно представить в виде двухстолбцовой таблицы. В этой таблице в столбце СОТРУДНИК указаны имена сотрудников некоторой фирмы или какого-то ее подразделения, а в столбце ВЫПЛАТА — денежная сумма, причитающаяся соответствующему сотруднику. Допустим, нас интересуют выплаты только Иванову и Петрову — элементам домена атрибута СОТРУДНИКИ. Обозначим через $D_{\text{СОТРУДНИК}}$ множество всех значений столбца СОТРУДНИК. Это и есть домен атрибута СОТРУДНИК. Аналогично, обозначим через $D_{\text{ВЫПЛАТА}}$ домен атрибута ВЫПЛАТА. Тогда интересующее нас отношение можно определить следующим образом:

$$\text{ЗАРПЛАТА}(\text{СОТРУДНИК}, \text{ВЫПЛАТА}) \bigcap \{\text{Иванов}, \text{Петров}\} \times D_{\text{ВЫПЛАТА}}.$$

Здесь через {Иванов, Петров} обозначено требуемое подмножество домена атрибута СОТРУДНИК.

На естественном языке запрос на получение указанного множества кортежей выглядит очень просто: "выбрать кортежи, в которых значение атрибута СОТРУДНИК равно "Иванов" или "Петров". Эквивалентный запрос можно сформулировать и так: "выбрать кортежи, в которых значение атрибута СОТРУДНИК равно "Иванов", и те кортежи, в которых значение атрибута СОТРУДНИК равно "Петров". Обратите внимание на использование союзов "или" и "и" в этих формулировках запросов. Если в первой формулировке "или" заменить на "и", то искомое множество кортежей будет заведомо пустым, поскольку атрибут СОТРУДНИК в одном и том же кортеже не может иметь два и более различных значения. Вторая формулировка допускает замену "и" на "или", поскольку в ней речь идет о различных значениях атрибута в различных кортежах. Применяя в обычной речи союзы "и" и "или", всегда следует отдавать себе отчет в том, что имеется в виду — объединение или пересечение соответствующих множеств.

Примечание

На языке SQL данный запрос формулируется следующим образом:

```
SELECT * FROM ЗАРПЛАТА WHERE СОТРУДНИК='Иванов' OR СОТРУДНИК=
'Петров';
```

По-русски это выглядит так: "выбрать кортежи (записи) из таблицы `ЗАРПЛАТА`, в которых атрибут (столбец) `СОТРУДНИК` имеет значение `'Иванов'` или `'Петров'`".

Проекция

Операция проекции отношения заключается в удалении из него указанных атрибутов. Пусть дано некоторое отношение $R(A1, A2, \ldots, An)$. Обозначим через **A** множество $\{A1, A2, \ldots, An\}$ всех атрибутов отношения, а через **X** — некоторое его подмножество (т. е. $\mathbf{X} \subseteq \mathbf{A}$). Тогда операция проекции отношения $R(\mathbf{A})$ на множество атрибутов **X** приводит к отношению $R[\mathbf{X}]$, кортежи которого получаются из кортежей исходного отношения путем удаления значений тех атрибутов, которые не принадлежат **X**. Для любого кортежа z из отношения $R(\mathbf{A})$ его проекция $z[\mathbf{X}]$ на множество атрибутов **X** определяется аналогично: в этом кортеже следует оставить значения только атрибутов из **X**. Обратите внимание, что квадратные скобки здесь указывают лишь на то, что рассматривается проекция отношения или кортежа на заключенные в эти скобки атрибуты.

На рис. 1.2 в качестве примера показано некоторое отношение $R(A1, A2, A3)$ и его проекция $R[A1, A3]$ на множество атрибутов $\{A1, A3\}$.

R(A1,A2,A3) : таблица

A1	A2	A3
a	x	m
a	y	n
b	z	k
c	x	n
c	x	p

Проекция R(A1,A2,A3) на {A1,A3}

A1	A3
a	m
a	n
b	k
c	n
c	p

Рис. 1.2. Пример проекции отношения

Примечание

На языке SQL проекция отношения (таблицы) на заданное множество атрибутов (столбцов) производится с помощью оператора:

```
SELECT список_столбцов FROM имя_таблицы;
```

На практике часто операция проекции используется совместно с операцией селекции (выборки) кортежей. В этом случае применяется следующее выражение на языке SQL:

```
SELECT список_столбцов FROM имя_таблицы WHERE условие_выборки;
```

Естественное соединение

Предположим, имеются два отношения, содержащие кроме прочих и одинаковые атрибуты. Рассмотрим, например, две таблицы *R*1 (ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА) и *R*2 (ПРЕПОДАВАТЕЛЬ, КАФЕДРА).

- Отношение *R*1 содержит сведения об именах преподавателей и об учебных дисциплинах, занятия по которым они проводят. Вообще говоря, один и тот же преподаватель может заниматься несколькими дисциплинами, а по одной и той же дисциплине могут проводить занятия разные преподаватели.
- Отношение *R*2 содержит сведения о приписке преподавателей к кафедрам. Предполагается, что каждый преподаватель может быть приписан только к одной кафедре.

Естественно, может возникнуть задача сведения этих двух отношений в одно (соединить две таблицы в одну) — *R*3 (ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА, КАФЕДРА). На рис. 1.3 показаны примеры отношений *R*1 и *R*2, а также результат их соединения *R*3. Важным обстоятельством, позволившим выполнить соединение двух отношений, является наличие у них общего атрибута ПРЕПОДАВАТЕЛЬ. Обратите внимание, что проекция отношения *R*3 на атрибуты {ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА} в точности равна отношению *R*1, а проекция на атрибуты {ПРЕПОДАВАТЕЛЬ, КАФЕДРА} — отношению *R*2. Это означает, что при соединении отношений *R*1 и *R*2 в отношение *R*3 не было привнесено ничего нового и ничего не было потеряно. Другими словами, отношение *R*3 в точности представляет информацию, содержащуюся в отношениях *R*1 и *R*2.

Рассмотренная операция соединения отношений называется *операцией естественного соединения*. Теперь уточним ее определение.

R1 : таблица

ПРЕПОДАВАТЕЛЬ	ДИСЦИПЛИНА
Иванов	Математический анализ
Иванов	Геометрия
Иванов	Теория вероятностей
Михайлов	Органическая химия
Михайлов	Неорганическая химия
Сергеев	Квантовая механика
Сергеев	Атомная физика
Сергеев	Физика твердого тела
Сидоров	Атомная физика
Сидоров	Кваковая механика
Сидоров	Термодинамика
Петров	Математический анализ
Петров	Математическая логика
Петров	Теория множеств
Федоров	Топология
Федоров	Математическая логика

R2 : таблица

ПРЕПОДАВАТЕЛЬ	КАФЕДРА
Иванов	101
Михайлов	103
Петров	101
Сергеев	102
Сидоров	102
Федоров	101

R3=R1*R2 : запрос на выборку

ПРЕПОДАВАТЕЛЬ	ДИСЦИПЛИНА	КАФЕДРА
Иванов	Геометрия	101
Иванов	Математический анализ	101
Иванов	Теория вероятностей	101
Михайлов	Неорганическая химия	103
Михайлов	Органическая химия	103
Петров	Теория множеств	101
Петров	Математическая логика	101
Петров	Математический анализ	101
Сергеев	Квантовая механика	102
Сергеев	Атомная физика	102
Сергеев	Физика твердого тела	102
Сидоров	Атомная физика	102
Сидоров	Кваковая механика	102
Сидоров	Термодинамика	102
Федоров	Топология	101
Федоров	Математическая логика	101

Рис. 1.3. Пример соединения отношений

Пусть даны два отношения $R1(\mathbf{A1})$ и $R2(\mathbf{A2})$, где **A1**, **A2** — множества атрибутов (а не одиночные атрибуты). Естественным соединением этих отношений $R1(\mathbf{A1}) * R2(\mathbf{A2})$ называется отноше-

ние $R3(\mathbf{A1} \cup \mathbf{A2})$, содержащее те и только те кортежи z, для которых выполняются одновременно следующие два условия:

- $z[\mathbf{A1}] \in R1(\mathbf{A1})$;
- $z[\mathbf{A2}] \in R2(\mathbf{A2})$.

Напомню, квадратные скобки указывают на то, что рассматривается проекция на заключенные в них атрибуты. Операция естественного соединения здесь обозначена через символ (*).

Операция естественного соединения двух отношений может быть выражена через операции декартового произведения, проекции и пересечения. Рассмотрим три случая.

- Множества атрибутов отношений $R1(\mathbf{A1})$ и $R2(\mathbf{A2})$ не равны, но пересекаются (т. е. $\mathbf{A1} \neq \mathbf{A2}$, $\mathbf{A1} \cap \mathbf{A2} \neq \varnothing$). В данном случае:

$$R1(\mathbf{A1}) * R2(\mathbf{A2}) = (R1(\mathbf{A1}) \times R2(\mathbf{A2} - \mathbf{A1})) \cap$$
$$\cap (R1(\mathbf{A1} - \mathbf{A2}) \times R2(\mathbf{A2})).$$

- Множества атрибутов отношений не пересекаются (т. е. $\mathbf{A1} \cap \mathbf{A2} = \varnothing$). В данном случае естественное соединение равно декартовому произведению исходных отношений:

$$R1(\mathbf{A1}) * R2(\mathbf{A2}) = R1(\mathbf{A1}) \times R2(\mathbf{A2}).$$

- Множества атрибутов отношений равны (т. е. $\mathbf{A1} = \mathbf{A2}$). Тогда естественное соединение равно пересечению исходных отношений:

$$R1(\mathbf{A1}) * R2(\mathbf{A2}) = R1(\mathbf{A1}) \cap R2(\mathbf{A2}).$$

На рис. 1.4 показаны два исходных отношения, их декартово произведение и естественное соединение. В таблице для декартового произведения отмечены галочками те кортежи, из которых получены кортежи естественного соединения.

Примечание

Язык SQL предоставляет специальные средства для различных типов соединения таблиц. Естественное соединение отношений *R1* (ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА) и *R2* (ПРЕПОДАВАТЕЛЬ, КАФЕДРА),

показанных на рис. 1.3, можно выполнить с помощью следующего SQL-выражения:

```
SELECT R1.*, R2.КАФЕДРА FROM R1, R2 WHERE R1.ПРЕПОДАВАТЕЛЬ=
R2.ПРЕПОДАВАТЕЛЬ
```

По-русски это выглядит так: "выбрать все столбцы таблицы `R1` и столбец `КАФЕДРА` таблицы `R2` из декартового произведения `R1` и `R2` при условии, что преподаватели из этих таблиц одинаковы".

Здесь подразумевается именно декартово произведение, т. к. после ключевого слова `FROM` указаны две таблицы. В SQL также можно использовать оператор `NATURAL JOIN` для явного указания операции естественного соединения.

R1 : таблица

A1	A2
a1	b1
a1	b2
a2	b1

R2 : таблица

A2	A3
b1	c1
b1	c2
b2	c1

R1xR2 : запрос на выборку

	A1	R1.A2	R2.A2	A3
✔	a1	b1	b1	c1
	a1	b2	b1	c1
✔	a2	b1	b1	c1
✔	a1	b1	b1	c2
	a1	b2	b1	c2
✔	a2	b1	b1	c2
	a1	b1	b2	c1
✔	a1	b2	b2	c1
	a2	b1	b2	c1

R3=R1*R2 : запрос на выборку

A1	A2	A3
a2	b1	c1
a1	b1	c1
a2	b1	c2
a1	b1	c2
a1	b2	c1

Рис. 1.4. Естественное соединение отношений

При работе с базами данных кроме естественного соединения используются и другие способы комбинации таблиц. Однако в реляционной теории понятие естественного соединения играет исключительно важную роль, поскольку через него определяется понятие корректной декомпозиции одной таблицы на несколько других. Об этом будет рассказано в следующем разделе.

1.3. Декомпозиция отношений

Таблица базы данных, представляющая некоторое отношение, может иметь очень большие размеры по количеству записей (строк) и столбцов, а также по общему объему содержащихся в них данных. Эффективность работы с таблицами по мере роста их объема уменьшается. Это обстоятельство вынуждает искать способы декомпозиции одной большой таблицы на несколько таблиц меньших размеров.

Так, в рассмотренном в предыдущем разделе отношении *R*3 (ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА, КАФЕДРА) (см. рис. 1.3) все атрибуты имеют повторяющиеся значения, а атрибут ПРЕПОДАВАТЕЛЬ однозначно определяет атрибут КАФЕДРА, поскольку каждый преподаватель приписан только к одной кафедре. Наличие такой зависимости между атрибутами ПРЕПОДАВАТЕЛЬ и ДИСЦИПЛИНА наталкивает на мысль, что декомпозиция возможна.

Отношения *R*1 (ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА) и *R*2 (ПРЕПОДАВАТЕЛЬ, КАФЕДРА) можно рассматривать как результат декомпозиции отношения *R*3, т. е. как операцию, обратную естественному соединению. Обратите внимание, что общий объем данных в отношениях *R*1 и *R*2 меньше объема данных в отношении *R*3. Кроме того, в отношении *R*2 атрибут ПРЕПОДАВАТЕЛЬ не имеет повторяющихся значений.

Очевидно, что отношения, получающиеся в результате декомпозиции исходного отношения, являются проекциями последнего на некоторые подмножества атрибутов. Напомню, что проекция отношения на заданные атрибуты получается из этого отношения путем удаления из него всех атрибутов, кроме заданных. Заметим также,

что для декомпозиции отношения на две проекции необходимо, чтобы оно было определено не менее чем для трех атрибутов.

Декомпозиция может быть корректной или некорректной. Другими словами, она может быть обратимой или необратимой. Обратимость декомпозиции означает, что она является эквивалентным преобразованием исходной информации (форма изменяется, а содержание нет). Необратимость декомпозиции означает неэквивалентность преобразования: в содержание либо что-то вносится, либо что-то теряется. Далее мы дадим определение корректной декомпозиции и покажем пример некорректной декомпозиции.

1.3.1. Корректная декомпозиция

Декомпозиция отношения на некоторые свои проекции называется *корректной*, если исходное отношение можно восстановить по этим проекциям с помощью операции естественного соединения. Точнее, отношение $R(\mathbf{A})$ корректно декомпозируется на свои проекции $R[\mathbf{X}]$ и $R[\mathbf{Y}]$ ($\mathbf{X} \subseteq \mathbf{A}$, $\mathbf{Y} \subseteq \mathbf{A}$), если выполняется равенство $R(\mathbf{A}) = R[\mathbf{X}] * R[\mathbf{Y}]$.

Например, декомпозиция отношения $R3$ (ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА, КАФЕДРА) (см. рис. 1.3) на свои проекции:

$R3$ [ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА] = $R1$ (ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА)

и

$R3$ [ПРЕПОДАВАТЕЛЬ, КАФЕДРА] = $R2$ (ПРЕПОДАВАТЕЛЬ, КАФЕДРА)

является корректной.

Это утверждение просто следует из того, что отношение $R3$ было получено из отношений $R1$ и $R2$ путем их естественного соединения *(см. разд. 1.2.3)*.

1.3.2. Пример некорректной декомпозиции

На рис. 1.5 показано некоторое отношение $R(A1, A2, A3)$, три его проекции $R[A1, A2]$, $R[A2, A3]$ и $R[A1, A3]$, а также естественное соединение этих трех проекций $R[A1, A2] * R[A2, A3] * R[A1, A3]$. Нетрудно заметить, что естественное соединение содержит кор-

теж $(0, 0, 0)$, которого нет в исходном отношении $R(A1, A2, A3)$. Таким образом, данная декомпозиция некорректна. Можно показать, что это отношение вообще нельзя корректно декомпозировать, т. к. в естественном соединении всегда будет появляться кортеж $(0, 0, 0)$.

R(A1,A2,A3) : таблица

A1	A2	A3
1	0	0
0	1	0
0	0	1

Проекция на A1,A2

A1	A2
1	0
0	1
0	0

Проекция на A2,A3

A2	A3
0	0
1	0
0	1

Проекция на A1,A3

A1	A3
1	0
0	0
0	1

Естественное соединение трех проекций

A1	A2	A3
1	0	0
0	1	0
0	0	1
0	0	0

Рис. 1.5. Пример некорректной декомпозиции отношения

Далее, говоря о декомпозиции отношения, мы будем всегда иметь в виду корректную декомпозицию.

1.3.3. Зависимости между атрибутами

Итак, отношение не всегда можно декомпозировать на две своих проекции. Критерием возможности декомпозиции является наличие в отношении некоторых зависимостей между его атрибутами. Это общий критерий. Если в чем-либо обнаруживаются какие-нибудь зависимости, то существует принципиальная возможность представить это нечто в более компактном виде.

Чтобы отношение можно было декомпозировать на две своих проекции, достаточно существования в нем так называемых функциональных зависимостей. Необходимым и достаточным условием декомпозиции отношения является наличие в нем многозначных зависимостей. Рассмотрим эти зависимости по порядку.

Функциональные зависимости

Функциональные зависимости просты для понимания и обычно легко обнаруживаются в отношении. Между атрибутами *A* и *B* существует функциональная зависимость, если любое значение атрибута *A* однозначно определяет значение атрибута *B*.

Рассмотрим несколько примеров отношений, показанных на рис. 1.3.

- В отношении *R*2 (ПРЕПОДАВАТЕЛЬ, КАФЕДРА) между атрибутами ПРЕПОДАВАТЕЛЬ и КАФЕДРА имеется функциональная зависимость, поскольку каждый преподаватель может быть приписан лишь к одной кафедре (предполагается, что совместительство не допускается). Другими словами, по значению атрибута ПРЕПОДАВАТЕЛЬ можно однозначно определить, на какой кафедре он работает. Однако между атрибутами КАФЕДРА и ПРЕПОДАВАТЕЛЬ в отношении *R*2 нет функциональной зависимости: на одной и той же кафедре могут работать несколько преподавателей.
- В отношении *R*1 (ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА) нет ни одной функциональной зависимости: один и тот же преподаватель может заниматься несколькими дисциплинами, а одну и ту же дисциплину могут вести несколько преподавателей.
- В отношении *R*3 (ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА, КАФЕДРА) существуют две функциональных зависимости:
  - между атрибутами ПРЕПОДАВАТЕЛЬ и КАФЕДРА;
  - между атрибутами ДИСЦИПЛИНА и КАФЕДРА.

Вообще говоря, функциональная зависимость может иметь место не только между отдельными атрибутами, но и между подмножествами атрибутов отношения. Для этого общего случая дадим определение функциональной зависимости.

Пусть дано некоторое отношение $R(\mathbf{A})$ с атрибутами из множества **A**, т. е. **A** — это множество атрибутов, а не отдельный атрибут. Обозначим через **X** и **Y** некоторые подмножества атрибутов ($\mathbf{X} \subseteq \mathbf{A}$, $\mathbf{Y} \subseteq \mathbf{A}$). В отношении $R(\mathbf{A})$ выполняется функциональная зависимость между атрибутами **X** и **Y**, обозначаемая как $\mathbf{X} \rightarrow \mathbf{Y}$, если для любых кортежей $z1$ и $z2$ этого отношения из равенства их проекций на множество атрибутов **X** следует равенство их проекций на множество атрибутов **Y**. Иначе говоря, функциональная зависимость $\mathbf{X} \rightarrow \mathbf{Y}$ выполняется, если справедливо следующее высказывание: для любых $z1$ и $z2$ из равенства $z1[\mathbf{X}] = z2[\mathbf{X}]$ следует равенство $z1[\mathbf{Y}] = z2[\mathbf{Y}]$.

Если выполняется функциональная зависимость $\mathbf{X} \rightarrow \mathbf{Y}$, то говорят, что атрибуты **X** функционально (однозначно) определяют атрибуты **Y**.

Например, в отношении $R3$ (ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА, КАФЕДРА), показанном на рис. 1.3, имеются следующие функциональные зависимости:

- {ПРЕПОДАВАТЕЛЬ} –> {КАФЕДРА} — преподаватель функционально определяет кафедру;
- {ДИСЦИПЛИНА} –> {КАФЕДРА} — дисциплина функционально определяет кафедру;
- {ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА} –> {КАФЕДРА} — преподаватель и дисциплина функционально определяют кафедру.

Рассмотрим еще один пример. Предположим, что отношение ПРОДАЖИ (ТОВАР, КОЛИЧЕСТВО, ЦЕНА, СТОИМОСТЬ) содержит сведения о проданных товарах. Стоимость товара однозначно определяется его количеством и ценой, поскольку существует простая формула: *стоимость* = *количество* × *цена*. Поэтому в данном отношении имеется функциональная зависимость {КОЛИЧЕСТВО, ЦЕНА} –> {СТОИМОСТЬ}.

Если в отношении $R(\mathbf{A})$ выполняется функциональная зависимость $\mathbf{X} \rightarrow \mathbf{Y}$ ($\mathbf{X} \subseteq \mathbf{A}$, $\mathbf{Y} \subseteq \mathbf{A}$), то это отношение декомпозируется на две своих проекции $R[\mathbf{X} \cup \mathbf{Y}]$ и $R[\mathbf{X} \cup (\mathbf{A} - \mathbf{Y})]$. Иначе говоря, справедливо следующее равенство:

$$R(\mathbf{A}) = R[\mathbf{X} \cup \mathbf{Y}] * R[\mathbf{X} \cup (\mathbf{A} - \mathbf{Y})].$$

Здесь через (*) обозначена операция естественного соединения. Таким образом, исходное отношение $R(\mathbf{A})$ можно восстановить с помощью операции естественного соединения двух своих проекций — $R[\mathbf{X} \cup \mathbf{Y}]$ и $R[\mathbf{X} \cup (\mathbf{A} - \mathbf{Y})]$.

Например, в отношении *R*3 (ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА, КАФЕДРА), показанном на рис. 1.3, имеется функциональная зависимость {ПРЕПОДАВАТЕЛЬ} –> {КАФЕДРА}. Следовательно, это отношение можно декомпозировать на две проекции:

- *R*3 [ПРЕПОДАВАТЕЛЬ, КАФЕДРА];
- *R*3 [ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА].

В данном примере можно принять такие обозначения:

- $\mathbf{A}$ = {ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА, КАФЕДРА};
- $\mathbf{X}$ = {ПРЕПОДАВАТЕЛЬ};
- $\mathbf{Y}$ = {КАФЕДРА};
- $\mathbf{X} \cup \mathbf{Y}$ = {ПРЕПОДАВАТЕЛЬ, КАФЕДРА};
- $\mathbf{A} - \mathbf{Y}$ = {ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА};
- $\mathbf{X} \cup (\mathbf{A} - \mathbf{Y})$ = {ПРЕПОДАВАТЕЛЬ} $\cup$ {ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА} = {ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА}.

Отношение ПРОДАЖИ (ТОВАР, КОЛИЧЕСТВО, ЦЕНА, СТОИМОСТЬ), имеющее функциональную зависимость {КОЛИЧЕСТВО, ЦЕНА} –> {СТОИМОСТЬ}, можно декомпозировать на такие проекции:

- ПРОДАЖИ [КОЛИЧЕСТВО, ЦЕНА, СТОИМОСТЬ];
- ПРОДАЖИ [ТОВАР, КОЛИЧЕСТВО, ЦЕНА].

В любом отношении существует хотя бы одна функциональная зависимость. Например, для произвольного отношения $R(\mathbf{A})$ всегда выполняется зависимость $\mathbf{A} \rightarrow \mathbf{A}$, поскольку по определению в любом отношении не может быть одинаковых кортежей и, следовательно, любой кортеж однозначно определяет сам себя. Такая функциональная зависимость является тривиальной. Точнее, функциональная зависимость $\mathbf{X} \rightarrow \mathbf{Y}$ называется тривиальной, если $\mathbf{Y} \subseteq \mathbf{X}$.

Наличие в отношении нетривиальной функциональной зависимости является достаточным, но не необходимым условием коррект-

ной декомпозиции этого отношения на две проекции. Это означает, что если функциональная зависимость есть, то декомпозиция возможна. Однако корректная декомпозиция может быть выполнена и для некоторых отношений, в которых нет ни одной функциональной зависимости. Например, на рис. 1.4 было показано отношение $R3 = R1 * R2$, в котором нет ни одной нетривиальной функциональной зависимости. Однако оно может быть корректно декомпозировано на две своих проекции $R1$ и $R2$ (см. рис. 1.4).

Приведем некоторые свойства функциональных зависимостей.

- Пусть $\mathbf{Y} = \{A1, \ldots, Ak\}$ — некоторое подмножество атрибутов отношения. Если в этом отношении выполняется функциональная зависимость $\mathbf{X} \rightarrow \mathbf{Y}$, то выполняется и множество функциональных зависимостей $\mathbf{X} \rightarrow \{Ai\}$, где $i = 1, \ldots, k$. Другими словами, коль скоро выполняется функциональная зависимость между множествами атрибутов **X** и **Y**, то выполняются и зависимости между **X** и одиночными атрибутами из **Y**. Обратное утверждение тоже верно: если в отношении выполняются функциональные зависимости $\mathbf{X} \rightarrow \{Ai\}$ ($i = 1, \ldots, k$), то выполняется и функциональная зависимость $\mathbf{X} \rightarrow \mathbf{Y}$, где $\mathbf{Y} = \{A1, \ldots, Ak\}$.
- В отношении $R(\mathbf{A})$ функциональная зависимость $\mathbf{X} \rightarrow \mathbf{Y}$ выполняется тогда и только тогда, когда она выполняется в проекции $R[\mathbf{Z}]$, где $\mathbf{X} \cup \mathbf{Y} \subseteq \mathbf{Z}$.

Многозначные зависимости

Обобщением понятия функциональной (однозначной) зависимости является понятие *многозначной* зависимости. Наличие в отношении многозначной зависимости является необходимым и достаточным условием его декомпозиции на две проекции. Напомню, что выполнение функциональной зависимости обеспечивает лишь достаточное условие декомпозиции. Вот почему некоторые отношения без функциональных зависимостей все-таки могут быть декомпозированы.

Многозначная зависимость имеет смысл только для отношений с тремя и более атрибутами. Проще всего понятие многозначной

зависимости объяснить для отношения с тремя атрибутами. Пусть дано некоторое отношение $R(A1, A2, A3)$, для которого выполняются следующие условия:

- атрибут $A1$ не определяет однозначно атрибут $A2$, т. е. нет функциональной зависимости $\{A1\} -> \{A2\}$;
- атрибуты $A2$ и $A3$ не зависят друг от друга, т. е. в отношении $R(A1, A2, A3)$ любое значение $A2$ может сочетаться в кортежах с любым значением $A3$.

Тогда в отношении $R(A1, A2, A3)$ выполняется многозначная зависимость между атрибутами $A1$ и $A2$, которая обозначается как $\{A1\} ->> \{A2\}$. При этом говорят, что атрибут $A1$ многозначно определяет атрибут $A2$.

Это частное определение многозначной зависимости, возможно, и не очень понятное. Поэтому поясним его на примере.

Пусть дано отношение R (ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА, УВЛЕЧЕНИЕ), которое содержит сведения о преподавателях, дисциплинах, занятия по которым они проводят, и личных увлечениях (хобби). Предполагается, что один преподаватель может проводить занятия по нескольким дисциплинам, т. е. атрибут ПРЕПОДАВАТЕЛЬ не определяет однозначно атрибут ДИСЦИПЛИНА. Далее, мы считаем, что нет никакой связи между дисциплинами и увлечениями преподавателей. Иначе говоря, если известно, что преподаватель читает студентам, например, математический анализ, то из этого нельзя однозначно определить, чем конкретно он увлекается. И наоборот, если мы знаем о конкретном увлечении преподавателя классической музыкой, то не можем однозначно сказать, какой именно дисциплиной он занимается. Все это означает, что в рассматриваемом отношении значения атрибутов ДИСЦИПЛИНА и УВЛЕЧЕНИЕ могут сочетаться произвольным образом. Итак, атрибуты ДИСЦИПЛИНА и УВЛЕЧЕНИЕ не зависят друг от друга, а атрибут ПРЕПОДАВАТЕЛЬ не однозначно определяет атрибут ДИСЦИПЛИНА. Пример рассмотренного отношения показан на рис. 1.6.

Поскольку в отношении R постулируется независимость атрибутов ДИСЦИПЛИНА и УВЛЕЧЕНИЕ, то в этом отношении каждое сочетание значений атрибутов ПРЕПОДАВАТЕЛЬ и ДИСЦИПЛИНА должно

сочетаться с каждым значением атрибута УВЛЕЧЕНИЕ. В примере отношения, показанном на рис. 1.6, преподаватель Иванов проводит занятия по двум дисциплинам и имеет два хобби. Поэтому ему в таблице посвящено четыре записи (кортежа). Михайлов занят двумя дисциплинами и имеет одно увлечение, поэтому информация о нем представлена в отношении двумя записями. В данном отношении имеется многозначная зависимость ПРЕПОДАВАТЕЛЬ –>> ДИСЦИПЛИНА или, другими словами, атрибут ПРЕПОДАВАТЕЛЬ многозначно определяет атрибут ДИСЦИПЛИНА. Заметим, что мы бы не говорили о многозначной зависимости ПРЕПОДАВАТЕЛЬ –>> ДИСЦИПЛИНА, если бы между атрибутами ДИСЦИПЛИНА и УВЛЕЧЕНИЕ существовала какая-нибудь зависимость.

R : таблица

ПРЕПОДАВАТЕЛЬ	ДИСЦИПЛИНА	УВЛЕЧЕНИЕ
Иванов	Математический анализ	Классическая музыка
Иванов	Математический анализ	Футбол
Иванов	Геометрия	Классическая музыка
Иванов	Геометрия	Футбол
Михайлов	Органическая химия	Поп-музыка
Михайлов	Неорганическая химия	Поп-музыка

Рис. 1.6. Отношение с многозначной зависимостью ПРЕПОДАВАТЕЛЬ –>> ДИСЦИПЛИНА

Очевидно, что с точки зрения хранения информации такое представление отношения в виде одной таблицы избыточно. И действительно, его можно декомпозировать так, как показано на рис. 1.7.

Поскольку выполняется многозначная зависимость ПРЕПОДАВАТЕЛЬ –>> ДИСЦИПЛИНА, атрибуты ДИСЦИПЛИНА и УВЛЕЧЕНИЕ не зависят друг от друга. Поэтому имеется возможность представить связи ПРЕПОДАВАТЕЛЬ—ДИСЦИПЛИНА и ПРЕПОДАВАТЕЛЬ—УВЛЕЧЕНИЕ в виде отдельных отношений, причем так, чтобы исходное отношение *R* (ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА, УВЛЕЧЕНИЕ) полностью восстанавливалось с помощью операции естественного соединения.

R : таблица

ПРЕПОДАВАТЕЛЬ	ДИСЦИПЛИНА	УВЛЕЧЕНИЕ
Иванов	Математический анализ	Классическая музыка
Иванов	Математический анализ	Футбол
Иванов	Геометрия	Классическая музыка
Иванов	Геометрия	Футбол
Михайлов	Органическая химия	Поп-музыка
Михайлов	Неорганическая химия	Поп-музыка

Проекция на ПРЕПОДАВАТЕЛЬ, ДИСЦИПЛИНА

ПРЕПОДАВАТЕЛЬ	ДИСЦИПЛИНА
Иванов	Математический анализ
Иванов	Геометрия
Михайлов	Органическая химия
Михайлов	Неорганическая химия

Проекция на ПРЕПОДАВАТЕЛЬ, УВЛЕЧЕНИЕ

Преподаватель	УВЛЕЧЕНИЕ
Иванов	Классическая музыка
Иванов	Футбол
Михайлов	Поп-музыка

Рис. 1.7. Декомпозиция отношения, содержащего многозначную зависимость ПРЕПОДАВАТЕЛЬ –>> ДИСЦИПЛИНА

Обобщим вышесказанное. Отношение $R(A1, A2, A3)$, в котором выполняется многозначная зависимость $\{A1\} \text{–>>} \{A2\}$, можно декомпозировать на две проекции: $R[A1, A2]$ и $R[A1, A3]$. Это означает, что выполняется следующее равенство: $R(A1, A2, A3) = R[A1, A2] * R[A1, A3]$.

R(A1,A2,A3) : таблица

A1	A2	A3
a2	b1	c1
a2	b1	c2
a1	b2	c2
a1	b1	c1
a1	b1	c2
a1	b2	c1

Рис. 1.8. Отношение с многозначной зависимостью *A*1 –>> *A*2

Следующий пример отличается от рассмотренного ранее лишь уровнем абстракции. Пусть дано некоторое отношение $R(A1, A2, A3)$, которое представлено в виде таблицы, показанной на рис. 1.8. Это представление отношения. Вся информация о зависимостях между его атрибутами не постулируется какими-либо предложениями, а заключена в самом представлении отношения.

Иначе говоря, зависимости, если они существуют, могут быть выявлены из анализа самого содержимого отношения, хотя это и не всегда простая задача.

В данном отношении выполняется многозначная зависимость $A1 \text{–>>} A2$. Поэтому его можно декомпозировать на две проекции — $R[A1, A2]$ и $R[A1, A3]$, как показано на рис. 1.9.

R(A1,A2,A3) : таблица

A1	A2	A3
a2	b1	c1
a2	b1	c2
a1	b2	c2
a1	b1	c1
a1	b1	c2
a1	b2	c1

Проекция на A1, A2

A1	A2
a1	b1
a1	b2
a2	b1

Проекция на A1, A3

A1	A3
a1	c1
a1	c2
a2	c1
a2	c2

Рис. 1.9. Декомпозиция отношения *R*(*A*1, *A*2, *A*3) с многозначной зависимостью *A*1 –>> *A*2

В рассмотренном отношении (см. рис. 1.8) кроме многозначной зависимости $A1 \text{–>>} A2$ выполняется еще и многозначная зависимость $A1 \text{–>>} A3$. Следовательно, исходное отношение можно декомпозировать на проекции $R[A1, A3]$ и $R[A1, A2]$, т. е. на те же самые, что позволяет получить зависимость $A1 \text{–>>} A2$.

В общем случае многозначная зависимость определяется не только между одноэлементными множествами атрибутов.

Ранее уже отмечалось, что наличие многозначной зависимости в отношении является необходимым и достаточным условием декомпозиции отношения на две его проекции. Если в отношении $R(\mathbf{A})$ выполняется многозначная зависимость $\mathbf{X} \text{–>>} \mathbf{Y}$, то это

отношение можно декомпозировать на две проекции — $R[\mathbf{X}, \mathbf{Y}]$ и $R[\mathbf{X}, \mathbf{Z}]$. Это утверждение о том, что многозначная зависимость необходима для декомпозиции.

То, что наличие многозначной зависимости является еще и достаточным условием декомпозиции, вытекает из следующего. Пусть отношение $R(\mathbf{A})$ декомпозируется на две проекции — $R[\mathbf{X}]$ и $R[\mathbf{Y}]$ (это означает, что $R(\mathbf{A}) = R[\mathbf{X}] * R[\mathbf{Y}]$). Тогда в отношении $R(\mathbf{A})$ выполняются две многозначные зависимости: $\mathbf{X} \cap \mathbf{Y} \text{–>>} \mathbf{X} - \mathbf{Y}$ и $\mathbf{X} \cap \mathbf{Y} \text{–>>} \mathbf{Y} - \mathbf{X}$. Иначе говоря, если нам удалось декомпозировать отношение на две проекции, то это означает, что в отношении выполняются, по крайней мере, две многозначные зависимости.

На рис. 1.10 представлено некоторое отношение $R(A1, A2, A3)$ и две его проекции — $R[A1, A2]$ и $R[A2, A3]$. Нетрудно проверить, что выполняется равенство $R(A1, A2, A3) = R[A1, A2] * R[A2, A3]$. Следовательно, отношение декомпозируется на эти проекции. Но тогда в отношении $R(A1, A2, A3)$ выполняются две многозначные зависимости:

- $A2 \text{–>>} A1$, поскольку $\{A1, A2\} \cap \{A2, A3\} = \{A2\}$ и $\{A1, A2\} - \{A2, A3\} = \{A1\}$;
- $A2 \text{–>>} A3$, поскольку $\{A2, A3\} - \{A1, A2\} = \{A3\}$.

С учетом первой зависимости исходное отношение можно декомпозировать на проекции $R[A2, A1]$ и $R[A2, A3]$, показанные на рис. 1.10. С учетом второй зависимости это отношение можно декомпозировать на такие же проекции.

Нетрудно заметить, что в определении многозначной зависимости множества атрибутов $\mathbf{Y}$ и $\mathbf{Z}$ играют симметричные роли: если что-либо справедливо для $\mathbf{Y}$, то это же справедливо и для $\mathbf{Z}$, и наоборот. Поэтому можно сформулировать следующее утверждение: многозначная зависимость $\mathbf{X} \text{–>>} \mathbf{Y}$ выполняется в отношении $R(\mathbf{A})$ тогда и только тогда, когда в $R(\mathbf{A})$ выполняется многозначная зависимость $\mathbf{X} \text{–>>} \mathbf{Z}$.

Многозначные (а также и функциональные) зависимости сохраняются в проекциях отношения, которые не затрагивают их левой части. Другими словами, если в отношении $R(\mathbf{A})$ выполняется

многозначная зависимость $\mathbf{X} \text{–>>} \mathbf{Y}$, то в любой проекции $R[\mathbf{U}]$, для которой $\mathbf{X} \subseteq \mathbf{U}$, выполняется и многозначная зависимость $\mathbf{X} \text{–>>} \mathbf{Y} \bigcap \mathbf{U}$. Для функциональных зависимостей верно и обратное утверждение: если в проекции выполняется функциональная зависимость, то эта же зависимость выполняется и во всем отношении. Для многозначных зависимостей это утверждение не верно: из выполнимости многозначной зависимости в некоторой проекции отношения еще не следует ее выполнимость во всем отношении.

R(A1,A2,A3) : таблица

A1	A2	A3
a1	b1	c1
a1	b1	c2
a1	b2	c1
a2	b1	c1
a2	b1	c2

Проекция на A1, A2

A1	A2
a1	b1
a1	b2
a2	b1

Проекция на A2,A3

A2	A3
b1	c1
b1	c2
b2	c1

Рис. 1.10. Из данной декомпозиции отношения следует наличие в нем зависимостей *A2* –>> *A1* и *A2* –>> *A3*

1.3.4. Правила вывода зависимостей

Все утверждения, сформулированные для многозначных зависимостей, справедливы и для функциональных зависимостей. Поэтому далее мы будем рассматривать только многозначные зависимости.

Если в отношении было каким-либо образом замечено выполнение одних зависимостей, то другие могут быть выведены логически из тех, что обнаружены. В основе декомпозиции, как известно, находится та или иная зависимость. Таким образом, имея несколько зависимостей, мы имеем возможность выбора декомпозиции. Не всякая декомпозиция может оказаться удачной с точки

зрения экономного представления данных. Не стоит также забывать и о том, что результаты декомпозиции (проекции исходного отношения) могут использоваться как исходный материал для дальнейшей декомпозиции: то, что хорошо на первом этапе декомпозиции, может плохо сказаться на всей многоэтапной декомпозиции. Иначе говоря, оптимизация декомпозиции в целом не означает, что каждый отдельный ее этап должен быть наилучшим.

Данный раздел посвящен выводам одних зависимостей из факта наличия других и адресуется тем, кто хочет либо глубже познать реляционную теорию, либо поупражняться в логике и алгебре множеств. Этот материал не из легких и может быть пропущен при первом чтении данной главы. Так или иначе, я привожу здесь правила вывода зависимостей.

В систему правил вывода многозначных зависимостей входят четыре основных правила. Прежде чем их перечислить, договоримся об обозначениях. Через **A** будем обозначать множество всех атрибутов отношения, а через **X**, **Y** и **Z** — подмножества атрибутов ($\mathbf{X} \subseteq \mathbf{A}$, $\mathbf{Y} \subseteq \mathbf{A}$, $\mathbf{Z} \subseteq \mathbf{A}$). Разумеется, каждое из них может содержать и только единственный элемент — какой-нибудь один атрибут отношения. Для обозначения многозначной зависимости используется, как и раньше, символ (–>>). Однако все приведенные далее утверждения справедливы не только для многозначных зависимостей, но и для функциональных зависимостей.

Итак, для зависимостей между атрибутами отношения выполняются следующие правила вывода:

1. *Правило дополнения*. Если выполняется зависимость $\mathbf{X} \text{–>>} \mathbf{Y}$, то выполняется и зависимость $\mathbf{X} \text{–>>} \mathbf{Z}$ для всех **Z**, таких, что $\mathbf{Y} \cap \mathbf{Z} = \mathbf{X}$ и $\mathbf{X} \cup \mathbf{Y} \cup \mathbf{Z} = \mathbf{A}$.
2. *Правило рефлексивности*. Если $\mathbf{X} \subseteq \mathbf{Y}$, то $\mathbf{X} \text{–>>} \mathbf{Y}$.
3. *Правило пополнения*. Если выполняется зависимость $\mathbf{X} \text{–>>} \mathbf{Y}$ и для некоторых подмножеств атрибутов **V** и **W** справедливо включение $\mathbf{V} \subseteq \mathbf{W}$, то выполняется зависимость $\mathbf{X} \cup \mathbf{W} \text{–>>} \mathbf{Y} \cup \mathbf{V}$.
4. *Правило транзитивности*. Если выполняются две зависимости $\mathbf{X} \text{–>>} \mathbf{Y}$ и $\mathbf{Y} \text{–>>} \mathbf{Z}$, то выполняется и зависимость $\mathbf{X} \text{–>>} \mathbf{Z} - \mathbf{Y}$.

Из этих четырех основных правил можно логически вывести еще три дополнительных правила:

1. *Правило объединения*. Если выполняются две зависимости $\mathbf{X} \text{–>>} \mathbf{Y}$ и $\mathbf{X} \text{–>>} \mathbf{Z}$, то выполняется и зависимость $\mathbf{X} \text{–>>} \mathbf{Y} \cup \mathbf{Z}$.
2. *Правило разности*. Если выполняются две зависимости $\mathbf{X} \text{–>>} \mathbf{Y}$ и $\mathbf{X} \text{–>>} \mathbf{Z}$, то выполняется и зависимость $\mathbf{X} \text{–>>} \mathbf{Y} - \mathbf{Z}$.
3. *Правило пересечения*. Если выполняются две зависимости $\mathbf{X} \text{–>>} \mathbf{Y}$ и $\mathbf{X} \text{–>>} \mathbf{Z}$, то выполняется и зависимость $\mathbf{X} \text{–>>} \mathbf{Y} \cap \mathbf{Z}$.

Дополнительные правила удобно использовать для сокращения длины вывода зависимостей.

В качестве примера, а также для упражнения в обращении с зависимостями рассмотрим вывод из четырех основных правил: правила разности и правила пересечения.

Пусть даны две зависимости $\mathbf{X} \text{–>>} \mathbf{Y}$ и $\mathbf{X} \text{–>>} \mathbf{Z}$, тогда вывод правила разности состоит из следующих четырех шагов:

1. По правилу пополнения из $\mathbf{X} \text{–>>} \mathbf{Z}$ следует $\mathbf{X} \cup \mathbf{X} \text{–>>} \mathbf{X} \cup \mathbf{Z}$, т. е. $\mathbf{X} \text{–>>} \mathbf{X} \cup \mathbf{Z}$.
2. По правилу пополнения из $\mathbf{X} \text{–>>} \mathbf{Y}$ следует $\mathbf{X} \cup \mathbf{Z} \text{–>>} \mathbf{Y} \cup \mathbf{Z}$.
3. По правилу транзитивности из $\mathbf{X} \text{–>>} \mathbf{X} \cup \mathbf{Z}$ и $\mathbf{X} \cup \mathbf{Z} \text{–>>} \mathbf{Y} \cup \mathbf{Z}$ следует $\mathbf{X} \text{–>>} (\mathbf{Y} \cup \mathbf{Z}) - (\mathbf{X} \cup \mathbf{Z})$. Однако $(\mathbf{Y} \cup \mathbf{Z}) - (\mathbf{X} \cup \mathbf{Z}) = \mathbf{Y} - \mathbf{Z} - \mathbf{X}$, поэтому выполняется зависимость $\mathbf{X} \text{–>>} \mathbf{Y} - \mathbf{Z} - \mathbf{X}$.
4. Так как $(\mathbf{Y} - \mathbf{Z}) \cap \mathbf{X} \subseteq \mathbf{X}$, то по правилу пополнения из $\mathbf{X} \text{–>>} \mathbf{Y} - \mathbf{Z} - \mathbf{X}$ следует $\mathbf{X} \cup \mathbf{X} \text{–>>} (\mathbf{Y} - \mathbf{Z} - \mathbf{X}) \cup (\mathbf{Y} - \mathbf{Z}) \cap \mathbf{X}$. Но $\mathbf{X} \cup \mathbf{X} = \mathbf{X}$, а $(\mathbf{Y} - \mathbf{Z} - \mathbf{X}) \cup (\mathbf{Y} - \mathbf{Z}) \cap \mathbf{X} = \mathbf{Y} - \mathbf{Z}$, поэтому справедлива зависимость $\mathbf{X} \text{–>>} \mathbf{Y} - \mathbf{Z}$.

Теперь выведем правило пересечения. Поскольку $\mathbf{Y} \cap \mathbf{Z} = \mathbf{Y} - (\mathbf{Y} - \mathbf{Z})$, то правило пересечения получается из правила разности: из $\mathbf{X} \text{–>>} \mathbf{Y}$ и $\mathbf{X} \text{–>>} \mathbf{Y} - \mathbf{Z}$ следует $\mathbf{X} \text{–>>} \mathbf{Y} - \mathbf{Y} - \mathbf{Z}$ или $\mathbf{X} \text{–>>} \mathbf{Y} \cap \mathbf{Z}$. Правило объединения можно обосновать аналогичным образом.

1.3.5. Ключи

При синтезе и анализе отношений (таблиц базы данных) важную роль играет понятие *ключа* отношения (таблицы). Это понятие производное от понятия функциональной зависимости, которое было рассмотрено в предыдущем разделе. Можно считать, что ключ является аргументом (левой частью) функциональной зависимости между множествами атрибутов отношения.

Ключ (*key*) — это множество из одного или нескольких атрибутов, которое однозначно определяет (идентифицирует) всю запись в отношении. Таким образом, множество атрибутов **X** является ключом отношения $R(\mathbf{A})$, если в этом отношении есть функциональная зависимость $\mathbf{X} \rightarrow \mathbf{A}$. И наоборот, если в отношении $R(\mathbf{A})$ имеется функциональная зависимость $\mathbf{X} \rightarrow \mathbf{A}$, то множество атрибутов **X** является ключом этого отношения. Очевидно, что главным признаком ключа отношения является уникальность (неповторяемость) его значений.

Из самого определения отношения следует, что в любом отношении всегда найдется ключ. По крайней мере, ключом может быть множество всех атрибутов **A**, т. к. в любом отношении имеется тривиальная функциональная зависимость $\mathbf{A} \rightarrow \mathbf{A}$, поскольку каждая запись встречается в отношении всего один раз и, следовательно, однозначно идентифицирует сама себя. Однако при разработке таблиц базы данных обычно интересуются не тривиальными ключами, а состоящими из одного или небольшого количества атрибутов. Действительно, тривиальные зависимости не дают возможность получить декомпозицию отношения на две проекции.

Рассмотрим в качестве примера таблицу, представляющую отношение `Студенты` (`Фамилия`, `Имя`, `Отчество`, `Учебная_группа`). Какие атрибуты в этом отношении могут составить ключ? Предполагается, что каждый студент может находиться только в одной учебной группе. Казалось бы, что множество атрибутов {`Фамилия`, `Имя`, `Отчество`} должно однозначно идентифицировать студента, а следовательно, и группу. Это было бы верно, если бы мы исключили возможность существования в одной группе полных тезок (одно-

фамильцев с одинаковыми именами и отчествами). Хотя это и маловероятно, но все же возможно. Таким образом, множество атрибутов {Фамилия, Имя, Отчество} не подходит на роль нетривиального ключа. В подобных ситуациях создают специальный атрибут (столбец), предназначенный играть роль ключа. В нашем примере это может быть ID_студента (идентификатор студента). Значения этого атрибута могут быть какими угодно, но обязательно уникальными. В результате получается таблица Студенты (ID_студента, Фамилия, Имя, Отчество, Учебная_группа).

Студенты : таблица

ID_студента	Фамилия	Имя	Отчество	Учебная_группа
001	Иванов	Иван	Иванович	201
002	Петров	Петр	Петрович	201
003	Сидоров	Сидор	Сидорович	201
004	Федоров	Федор	Федорович	202
005	Иванов	Иван	Иванович	202
006	Иванов	Иван	Иванович	201
007	Николаев	Николай	Николаевич	202
008	Степанов	Степан	Степанович	203
009	Михайлов	Михаил	Михайлович	203
010	Иванов	Иван	Иванович	202

Студенты_список : таблица

ID_студента	Фамилия	Имя	Отчество
001	Иванов	Иван	Иванович
002	Петров	Петр	Петрович
003	Сидоров	Сидор	Сидорович
004	Федоров	Федор	Федорович
005	Иванов	Иван	Иванович
006	Иванов	Иван	Иванович
007	Николаев	Николай	Николаевич
008	Степанов	Степан	Степанович
009	Михайлов	Михаил	Михайлович
010	Иванов	Иван	Иванович

Группы : таблица

ID_студента	Учебная_группа
001	201
002	201
003	201
004	202
005	202
006	201
007	202
008	203
009	203
010	202

Рис. 1.11. Декомпозиция отношения Студенты с ключом ID_студента на две проекции

А если ID_студента является ключом, то в данном отношении имеется функциональная зависимость {ID_студента} –> {ID_студента, Фамилия, Имя, Отчество, Учебная_группа}, а также зависимость {ID_студента} –> {Фамилия, Имя, Отчество, Учебная_группа}. Последнюю зависимость, очевидно, можно использовать для декомпозиции исходного отношения на следующие два: Студенты_список (ID_студента, Фамилия, Имя, Отчество) и Группы (ID_студента, Учебная_группа). На рис. 1.11 показано исходное отношение Студенты и две его проекции.

Ключ, состоящий из одного атрибута, называют *простым*, а из нескольких — *составным*. Ключи, рассмотренные в данном разделе, также называют первичными (*primary key*).

1.4. Ограничения целостности отношений

До сих пор мы рассматривали отношения, представленные в виде таблиц своими кортежами (записями). Зависимости между атрибутами выражали некоторую целостность отношений, а корректная декомпозиция отношений на основе зависимостей рассматривалась как сохраняющая их целостность. Таким образом, мы изучали реляционные базы данных главным образом как теорию декомпозиции отношений, заданных экстенсионально, т. е. посредством наборов записей. На практике приходится проектировать базу данных как набор из нескольких таблиц, которые изначально пусты и только со временем заполняются конкретными данными. Иначе говоря, довольно сложные базы данных проектируются путем разработки множества отдельных таблиц с последующей установкой связей между ними, т. е. путем композиции частей в некое целое. При этом не исключается и декомпозиция таблиц на свои проекции.

Однако каждая таблица в отдельности, а также совокупность таблиц обычно являются не случайными комбинациями атрибутов, произвольно распределенными между различными таблицами. Как отдельные таблицы, так и вся их совокупность, называе-

мая *базой данных*, обладают некоторой целостностью, которая выражается через различного рода ограничения, накладываемые на значения столбцов и связи между ними. Таким образом, при проектировании базы данных мы заняты не обнаружением зависимостей для использования их при декомпозиции, а наоборот, заданием их с целью объединения таблиц в целостную систему связанных таблиц, содержащих непротиворечивые данные.

Далее мы рассмотрим несколько видов целостности и способы их задания.

1.4.1. Семантическая целостность

При проектировании базы данных стремятся к тому, чтобы каждая таблица соответствовала некоторому объекту внешнего мира. Существование такого объекта, разумеется, не зависит от базы данных. Если таблица полностью соответствует некоторому объекту, то говорят, что она обладает семантической целостностью и моделирует (представляет) этот объект. При этом каждая запись таблицы моделирует некий элемент объекта.

В таблице, обладающей семантической целостностью, должен быть первичный ключ. *Первичный ключ (см. разд. 1.3.5)* — это один столбец или группа столбцов, значения которых должны быть уникальными и определенными (не равными `NULL`). В языке SQL это ограничение целостности, выражающееся в виде ограничения на значения столбца или группы столбцов, задается ключевыми словами `PRIMARY KEY` *(см. разд. 7.1.1, 7.1.2)*.

1.4.2. Доменная целостность

С каждым атрибутом отношения связан домен — множество допустимых значений. При создании таблицы для каждого столбца кроме имени указывается и тип данных, которые он может содержать. Тип данных *(см. разд. 2.2)* ограничивает множество допустимых значений столбца, однако во многих случаях такого ограничения оказывается недостаточно. Например, если в числовом столбце `Возраст` таблицы `Сотрудники` указано значение 1000,

то мы не усомнимся, что это ошибочное значение. В той же таблице символьный столбец Должность может принимать значения из определенного списка, предусмотренного штатным расписанием организации, а не произвольную комбинацию символов или наименования должностей с орфографическими ошибками.

Ограничения на допустимые значения для столбца таблицы предназначены для поддержания доменной целостности. В языке SQL риск нарушить доменную целостность возникает при добавлении и обновлении записей с помощью операторов INSERT и UPDATE соответственно. Ограничения доменной целостности можно задать при создании таблицы с помощью оператора CREATE TABLE, а также предварительно, путем создания домена, применив оператор CREATE DOMAIN.

1.4.3. Ссылочная целостность

В хорошо спроектированной базе из нескольких таблиц они связаны друг с другом. Так столбец в одной таблице может ссылаться на столбец другой таблицы этой же базы данных. Подобные ссылки представляют собой ограничения ссылочной целостности базы данных и играют важную роль при поддержке ее общей целостности. Вместе с тем, наличие ссылок порождает так называемую *проблему аномалий модификации данных*.

Связи между таблицами обычно несимметричны: одна таблица зависит от другой. Допустим, в базе данных имеются две таблицы:

- Клиенты (Имя_клиента, Адрес, Телефон);
- Продажи (ID, Товар, Количество, Цена, Стоимость, Имя_клиента).

В таблице Клиенты столбец Имя_клиента является ключом (PRIMARY KEY), т. е. имеет уникальные и определенные значения. В таблице Продажи одноименный столбец не является ключом, его значения могут повторяться, т. к. один и тот же клиент может приобрести несколько товаров. Эта таблица связана с таблицей Клиенты по столбцу Имя_клиента, т. е. столбец Имя_клиента пер-

вой таблицы (Продажи) ссылается на одноименный столбец второй таблицы (Клиенты). Другими словами, данные таблицы находятся в родительско-дочернем отношении: таблица Клиенты родительская, Продажи дочерняя. Данная связь между таблицами организуется путем объявления столбца Имя_клиента таблицы Продажи внешним ключом (FOREIGN KEY), ссылающимся на первичный ключ в таблице Клиенты *(см. разд. 7.1.3)*.

Аномалии модификации могут возникнуть различными способами и при различных обстоятельствах, вызывая трудности. Предположим, какой-то клиент перестал вас интересовать (например, он перестал делать у вас покупки). Если запись о нем удалить из таблицы Клиенты, то в дочерней таблице Продажи останутся записи, ссылающиеся на отсутствующую запись в родительской таблице. Аналогичная ситуация возникает при попытке добавить в дочернюю таблицу запись, когда в родительскую таблицу еще не было сделано соответствующего добавления. О способах поддержки ссылочной целостности и поведении в случае возникновения аномалии модификации будет рассказано в *разд. 7.1.3*.

1.5. Нормализация таблиц

База данных может быть спроектирована хорошо или плохо. Сразу создать хороший проект довольно сложно, и на практике процесс проектирования обычно является итерационным: состав таблиц и их структура модифицируются в несколько этапов, пока не будет получен приемлемый результат.

В плохом проекте часто возникают аномалии модификации данных, устранить которые довольно трудно. Такие аномалии могут возникнуть даже в однотабличной базе данных.

Рассмотрим в качестве примера таблицу Продажи (Клиент, Товар, Количество, Цена), показанную на рис. 1.12.

В таблице, показанной на рис. 1.12, могут возникнуть аномалии модификации данных. Предположим, было решено удалить из нее запись о клиенте "ОАО "Рога и копыта", поскольку теперь он

ничего не приобретает в вашей фирме. Но тогда вы потеряете информацию и цене на "Хвосты". Если же вам потребуется добавить запись о каком-нибудь новом товаре, то необходимо добавить и сведения о покупателе и количестве этого товара. А если пока такого покупателя нет?

Продажи : таблица

Клиент	Товар	Количество	Цена
Иванов	Хлеб	2	24,50р.
Петров	Молоко	3	30,00р.
ОАО "Рога и копыта"	Хвосты	25	120,00р.
ЗАО "111"	Молоко	1	30,00р.
Сидоров	Хлеб	3	24,50р.

Рис. 1.12. Пример таблицы, в которой могут возникнуть аномалии модификации

Аномалия модификации, возникшая в рассмотренном примере, обусловлена тем, что данная таблица содержит информацию, относящуюся к различным темам. В ней есть сведения и о том, что приобрели покупатели, и о цене товаров. Лучше разбить ее на две таблицы, посвященные двум различным темам: Продажи_клиенты и Прайс_лист, показанные на рис. 1.13.

Продажи_клиенты : таблица

Клиент	Товар	Количество
Иванов	Хлеб	2
Петров	Молоко	3
ОАО "Рога и копыта"	Хвосты	25
ЗАО "111"	Молоко	1
Сидоров	Хлеб	3

Прайс_лист : таблица

Товар	Цена
Хлеб	24,50р.
Молоко	30,00р.
Хвосты	120,00р.

Рис. 1.13. Результат декомпозиции таблицы Продажи

Вообще говоря, любую таблицу, относящуюся к двум или более темам, следует разбить на две или более таблицы. При определенных условиях, о которых говорилось ранее, можно декомпозировать и таблицу, посвященную одной теме. В этом случае каждая из результатных таблиц будет соответствовать какой-то

части одной темы. Этот процесс декомпозиции и составляет суть нормализации. Обнаружив в таблице аномалию модификации, мы устраняем ее путем декомпозиции на две или более таблицы так, чтобы они были свободны от аномалий. Однако, производя декомпозицию, мы вынуждены задавать ограничения ссылочной целостности *(см. разд. 1.4.3)*.

Таблицы классифицируются по тем видам аномалий модификации, которым они подвержены. Это так называемые нормальные формы таблиц (отношений). В своей статье 1970 г. И. Кодд определил три источника аномалий и три формы таблиц, свободных от них. В последующие годы он и другие исследователи обнаружили другие виды аномалий и предложили формы таблиц, которые им не подвержены. Далее приводится список всех специальных нормальных форм:

1. Первая нормальная форма (1НФ).
2. Вторая нормальная форма (2НФ).
3. Третья нормальная форма (3НФ).
4. Нормальная форма Бойса—Кодда (НФБК).
5. Четвертая нормальная форма (4НФ).
6. Пятая нормальная форма (5НФ).

Все нормальные формы вложены друг друга в следующем смысле: таблица в 2НФ является также и таблицей в 1НФ; таблица в 3НФ является таблицей и в 2НФ, и в 1НФ, и т. д.

Каждая из перечисленных нормальных форм могла устранить определенные виды аномалий, и не было гарантии, что с их помощью можно устранить всевозможные аномалии, о которых пока просто не было известно. В 1981 г. Р. Фагин ввел новую нормальную форму, названную *доменно-ключевой* (ДКНФ), и доказал, что таблица в ДКНФ свободна от всех аномалий модификации и наоборот: таблица, свободная от любых аномалий модификации, находится в ДКНФ. До появления этой важной теоремы теоретики реляционных баз данных должны были продолжать поиск не выявленных видов аномалий и соответствующих им нормальных форм. Теперь же было доказано, что для

того, чтобы получить уверенность в отсутствии всех видов аномалий модификации, следует привести таблицу к ДКНФ.

Далее мы кратко рассмотрим первые три наиболее важные для практики нормальные формы, а также доменно-ключевую нормальную форму.

1.5.1. Первая нормальная форма

Любая таблица, удовлетворяющая определению отношения, находится в 1НФ. Вот основные характеристики таблицы в 1НФ:

- в каждой строке таблицы должны содержаться данные, соответствующие некоторому объекту или его части;
- в каждом столбце должны находиться данные, соответствующие одному из атрибутов отношения;
- в каждой ячейке таблицы должно находиться только единственное значение;
- у каждого столбца должно быть уникальное имя;
- все строки (записи) в таблице должны быть различными;
- порядок расположения столбцов и строк в таблице не имеет значения.

Таблица (отношение) в 1НФ свободна от некоторых аномалий, но все же подвержена многим другим. Например, таблица, показанная на рис. 1.12, находится в 1НФ, но, как уже было отмечено, подвержена аномалиям удаления и добавления записей.

1.5.2. Вторая нормальная форма

Каждая таблица в 1НФ должна иметь первичный ключ. Он может состоять из одного или более столбцов (атрибутов). В последнем случае ключ называется составным. Чтобы таблица была в 2НФ, все ее неключевые столбцы должны однозначно определяться всем ключом, т. е. всеми его компонентами, а не некоторыми из них.

Рассмотрим пример отношения `Секции` (`Имя`, `Секция`, `Плата`), показанный на рис. 1.14.

Секции : таблица

Имя	Секция	Плата
Иванов	Футбол	100
Иванов	Волейбол	120
Петров	Лыжи	170
Сидоров	Шахматы	200
Сидоров	Лыжи	170
Федоров	Лыжи	170
Федоров	Волейбол	120

Рис. 1.14. Отношение `Секции` (`Имя`, `Секция`, `Плата`)

Ключом в данном отношении является {`Имя`, `Секция`}, но оно содержит функциональную зависимость `Секция` –> `Плата`. Аргумент (левая часть) этой зависимости является лишь частью составного ключа. Отношение `Секции` имеет аномалии удаления и добавления. Так, если мы захотим удалить записи с именем "`Иванов`", то потеряем стоимость футбольной секции. Мы не сможем добавить запись о новой секции, пока в нее кто-нибудь не запишется. Данных аномалий можно было бы избежать, если бы атрибут `Плата` зависел от всего ключа (однозначно определялся всем ключом).

Отношение `Секции` (`Имя`, `Секция`, `Плата`) в 1НФ можно разбить на два отношения во 2НФ:

- `Секция_члены` (`Имя`, `Секция`);
- `Секция_плата` (`Секция`, `Плата`).

1.5.3. Третья нормальная форма

В отношениях могут быть так называемые *транзитивные зависимости*, являющиеся источником аномалий модификации данных, против которых 2НФ бессильна. Транзитивная зависимость имеет место тогда, когда один атрибут однозначно определяет второй, второй однозначно определяет третий и т. д.

Рассмотрим в качестве примера отношение `Гости` (`ID_гостя`, `Тип_номера`, `Плата`), представляющее сведения о проживающих в гостинице. Ключом в этом отношении является `ID_гостя`, `Плата` однозначно определяется атрибутом `Тип_номера` (например,

люкс, полулюкс и т. д.), т. е. имеется функциональная зависимость Тип_номера –> Плата. Поскольку каждый гость проживает только в одном номере определенного типа, в отношении есть и функциональная зависимость ID_гостя –> Тип_номера. Таким образом, возникает транзитивная (опосредованная) зависимость ID_гостя –> Плата. Так как ключ состоит из единственного атрибута ID_гостя, то отношение находится в 2НФ.

В рассматриваемом отношении существует аномалия удаления. Удалив запись, мы потеряем не только информацию о каком-то госте (где он проживает), но и сведения о том, сколько стоит номер соответствующего типа.

Чтобы устранить указанную аномалию, следует декомпозировать исходное отношение Гости (ID_гостя, Тип_номера, Плата) на два:

- Проживание (ID_гостя, Тип_номера);
- Тип_плата (Тип_номера, Плата).

Эти отношения будут находиться в 2НФ и не содержать транзитивных зависимостей.

Таким образом, отношение находится в 3НФ, если оно находится в 2НФ и не содержит транзитивных зависимостей.

1.5.4. Доменно-ключевая нормальная форма

Если таблица находится в 3НФ, то остается довольно мало шансов для возникновения аномалий модификации данных, но они все равно есть. Чтобы исключить все виды возможных аномалий, таблица должна находиться в доменно-ключевой нормальной форме (ДКНФ).

Понятие ДКНФ довольно просто: отношение находится в ДКНФ, если каждое ограничение, накладываемое на него, является логическим следствием определения доменов и ключей. Термин *ограничение* (*constraint*) здесь намеренно трактуется широко. Р. Фагин определяет ограничение как любое правило, регулирующее возможные статические значения атрибутов, достаточно точное,

чтобы можно было проверить его выполнимость. Правила редактирования, ограничения взаимосвязей и структуры отношений, функциональные и многозначные зависимости являются примерами таких ограничений. Отсюда исключаются ограничения, связанные с изменением данных (ограничения, зависящие от времени). Другими словами, отношение находится в ДКНФ, если выполнение ограничений на домены и ключи влечет за собой выполнение всех ограничений.

Однако в настоящее время не известен алгоритм преобразования отношения в ДКНФ. Неизвестно также, какие отношения в принципе могут быть приведены к ДКНФ. Поиск и создание отношений в ДКНФ сейчас является искусством, а не наукой. В литературе обычно приводятся только примеры отношений в ДКНФ, которые мы здесь рассматривать не будем.

1.5.5. Денормализация

Чтобы исключить как можно больше аномалий модификации данных, старайтесь как можно больше нормализовать таблицы базы данных. Лучше, если вы доведете их до ДКНФ, хотя на практике это редко происходит. Чаще ограничиваются второй или третьей нормальными формами. Занимаясь нормализацией, вы увеличиваете количество таблиц в базе данных, и при определенном их количестве эффективность работы может оказаться слишком низкой. Кроме того, формулировать SQL-запросы к базе данных тем легче, чем меньше в ней таблиц. Так что на любом этапе своего развития база данных может быть в какой-то степени денормализованной.

Глава 2

Основы SQL

2.1. Что такое SQL

Первые разработки систем управления реляционными базами данных (реляционных СУБД) были выполнены в компании IBM в начале 1970-х годов. Тогда же был создан язык данных, предназначенный для работы в этих системах. Экспериментальная версия этого языка называлась SEQUEL — от англ. Structured English QUEry Language (структурированный английский язык запросов). Однако официальная версия была названа короче — SQL (Structured Query Language). Точнее говоря, SQL — это подъязык данных, поскольку СУБД содержит и другие языковые средства.

В 1981 году IBM выпускает реляционную СУБД SQL/DS. К этому времени компания Relation Software Inc. (сегодня это Oracle Corporation) уже выпустила свою реляционную СУБД. Эти продукты сразу же стали стандартом систем, предназначенных для управления базами данных. В состав этих продуктов вошел и SQL, который фактически стал стандартом для подъязыков данных. Производители других СУБД выпустили свои версии SQL. В них имелись не только основные возможности продуктов IBM. Чтобы получить некоторое преимущество для "своей" СУБД, производители вводили некоторые расширения SQL. Вместе с тем, начались работы по созданию общепризнанного стандарта SQL.

В 1986 году Американский национальный институт стандартов (American National Standards Institute, ANSI) выпустил официальный стандарт SQL-86, который в 1989 году был обновлен и получил новое название SQL-89. В 1992 году этот стандарт был назван SQL-92 (ISO/IEC 9075:1992). Последней к моменту написания этой книги версией стандарта SQL является SQL:2003 (ISO/IEC 9075X:2003).

Любая реализация SQL в конкретной СУБД несколько отличается от стандарта, соответствие которому объявлено производителем. Так, многие СУБД (например, Microsoft Access 2003, PostgreSQL 7.3) поддерживают SQL-92 не в полной мере, а лишь с некоторым уровнем соответствия. Кроме того, они поддерживают и элементы, которые не входят в стандарт. Однако разработчики СУБД стремятся к тому, чтобы новые версии их продуктов как можно в большей степени соответствовали стандарту SQL.

Внимание

В данной книге описаны элементы SQL:2003, не все из которых поддерживаются существующими СУБД. Прежде чем применять их на практике, следует убедиться, что они будут работать в вашей СУБД. Об этом можно узнать из технической документации.

Большинство описанных в книге элементов соответствуют и более ранним версиям SQL, в частности, широко распространенному SQL-92.

SQL задумывался как простой язык запросов к реляционной базе данных, близкий к естественному (точнее, к английскому) языку. Предполагалось, что близость по форме к естественному языку сделает SQL средством, доступным для широкого применения обычными пользователями баз данных, а не только программистами. Первоначально SQL не содержал никаких управляющих структур, свойственных обычным языкам программирования. Запросы, синтаксис которых довольно прост, вводились прямо с консоли последовательно один за другим и в этой же последовательности выполнялись. Однако SQL так и не стал инструментом банковских служащих, продавцов авиа- и железнодорожных

билетов, экономистов и других служащих различных фирм, использующих информацию, хранимую в базах данных. Для них простой SQL оказался слишком сложным и неудобным, несмотря на свою близость к естественному языку вопросов.

На практике с базой данных обычно работают посредством приложений, написанных программистами на процедурных языках, например, на C, Visual Basic, Pascal, Java и др. Часто приложения создаются в специальных средах визуальной разработки, таких как Delphi, Microsoft Access, Visual dBase и т. п. При этом разработчику приложения практически не приходится писать коды программ, поскольку за него это делает система разработки. Во всяком случае, работа с программным кодом оказывается минимальной. Эти приложения имеют удобный графический интерфейс, не вынуждающий пользователя непосредственно вводить запросы на языке SQL. Вместо него это делает приложение. Впрочем, приложение может как использовать, так и не использовать SQL для обращения к базе данных. SQL не единственное, хотя и очень эффективное средство получения, добавления и изменения данных, и если есть возможность использовать его в приложении, то это следует делать.

Реляционные базы данных могут существовать и действительно существуют вне зависимости от приложений, обеспечивающих пользовательский интерфейс. Если по каким-либо причинам такого интерфейса нет, то доступ к базе данных можно осуществить с помощью SQL, используя консоль или какое-нибудь приложение, с помощью которого можно соединиться с базой данных, ввести и отправить SQL-запрос (например, Borland SQL Explorer).

Язык SQL считают декларативным (описательным) языком, в отличие от языков, на которых пишутся программы. Это означает, что выражения на языке SQL описывают, что требуется сделать, а не каким образом. Например, для того чтобы выбрать из таблицы `Сотрудники` сведения о фамилиях и должностях сотрудников 102 отдела, достаточно выполнить следующий запрос:

```
SELECT Фамилия, Должность FROM Сотрудники WHERE Отдел=102;
```

По-русски данное выражение звучит так:

ВЫБРАТЬ `Фамилия`, `Должность` ИЗ `Сотрудники` ПРИ УСЛОВИИ, ЧТО `Отдел` = 102;

Чтобы изменить значение `"Иванов"` на `"Петров"` столбца `Фамилия`, достаточно выполнить следующий запрос:

```
UPDATE Сотрудники SET Фамилия = 'Петров' WHERE Фамилия =
'Иванов';
```

По-русски данное выражение выглядит так:

ОБНОВИТЬ `Сотрудники` УСТАНОВИВ `Фамилия` РАВНЫМ `'Петров'` ГДЕ `Фамилия` = `'Иванов'`;

Вам не нужно подробно описывать действия, которые должна выполнить СУБД, чтобы выбрать из таблицы указанные в запросе данные. Вы просто описываете, что желаете получить. В результате выполнения запроса СУБД возвращает таблицу, содержащую запрошенные вами данные. Если в базе данных не оказалось данных, соответствующих запросу, то будет возвращена пустая таблица.

Однако последние версии SQL поддерживают операторы управления вычислениями, свойственные процедурным языкам управления (операторы условного перехода и цикла). Поэтому SQL сейчас это не чисто декларативный язык.

Кроме выборки, добавления, изменения и удаления данных из таблиц, SQL позволяет выполнять все необходимые действия по созданию, модификации и обеспечению безопасности баз данных. Все эти возможности распределены между тремя компонентами SQL:

- DML (Data Manipulation Language — язык манипулирования данными) предназначен для поддержки базы данных: выбора (`SELECT`), добавления (`INSERT`), изменения (`UPDATE`) и удаления (`DELETE`) данных из таблиц. Эти операторы (команды) могут содержать выражения, в том числе и вычисляемые, а также подзапросы — запросы, содержащиеся внутри другого запроса. В общем случае выражение запроса может быть настолько

сложным, что сразу и не скажешь, что он делает. Однако сложный запрос можно мысленно разбить на части, которые легче анализировать. Аналогично, сложные запросы создаются из относительно простых для понимания выражений (подзапросов). DDL посвящена большая часть данной книги *(см. гл. 3—6, 8—10)*;

- DDL (Data Definition Language — язык определения данных) предназначен для создания, модификации и удаления таблиц и всей базы данных. Примерами операторов, входящих в DDL, являются `CREATE TABLE` (создать таблицу), `CREATE VIEW` (создать представление), `CREATE SHEMA` (создать схему), `ALTER TABLE` (изменить таблицу), `DROP` (удалить) и др. Основные элементы DDL рассматриваются в *гл. 7*;
- DCL (Data Control Language — язык управления данными) предназначен для обеспечения защиты базы данных от различного рода повреждений. СУБД предусматривает некоторую защиту данных автоматически. Однако в ряде случаев следует предусмотреть дополнительные меры, предоставляемые DCL, которые будут рассмотрены в *гл. 8—11*.

Внимание

Ключевые слова в выражениях на языке SQL могут записываться как прописными, так и строчными буквами, в одну или несколько строк. В конце выражения должна стоять точка с запятой (`;`). Однако во многих системах, обеспечивающих ввод, редактирование и отправку на выполнение SQL-выражений, в случае ввода одного такого выражения допускается не указывать признак окончания (`;`). Если же требуется выполнить блок из нескольких SQL-выражений, то они обязательно должны быть разделены точкой с запятой.

Все ключевые слова SQL (команды, операторы и проч.) являются зарезервированными и не должны использоваться в качестве имен таблиц, столбцов, переменных и т. п. Вот пример неправильного использования ключевых слов:

```
SELECT SELECT FROM WHERE WHERE SELECT=UPDATE;
```

2.2. Типы данных

Во всех языках программирования, а также в SQL важное место занимают поддерживаемые типы данных. Данные, которые хранятся в памяти компьютера и подвергаются обработке, можно отнести к различным типам. Понятие типа данных возникает естественным образом, когда необходимо применить к ним операции обработки. Например, операция умножения применяется к числам, т. е. к данным числового типа. А что получится, если умножить слово "Вася" на число 25? Поскольку трудно дать вразумительный ответ на этот вопрос, то напрашивается вывод: некоторые операции не следует применять к разнотипным данным. Мы также не знаем, что должно получиться в результате умножения слов "Саша" и "Маша", поэтому заключаем, что определенные операции вообще не применимы к данным некоторых типов. С другой стороны, существуют операции, результат которых зависит от типа данных. Например, операция сложения, обозначаемая символом (+), может применяться и к двум числам, и к двум строкам, состоящим из произвольных слов. В первом случае результатом применения этой операции будет некоторое число, а во втором — строка, получающаяся путем приписывания второй строки к концу первой. В случае строк операцию сложения еще называют *склейкой* или *конкатенацией*. Операции, применимые к различным типам данных, но обозначаемые одним и тем же символом, называют *перегруженными*. Так, операция, обозначаемая символом (+), является перегруженной: применительно к числам она выполняет арифметическое сложение, а применительно к строкам символов — склейку.

Типы данных столбцов таблиц базы данных могут отличаться от типов данных, поддерживаемых SQL. Это обстоятельство необходимо учитывать при составлении SQL-запросов. Так, например, если в таблице `Сотрудники` столбец `Зарплата` определен по каким-либо причинам как символьный, то следующий запрос приведет к ошибке из-за несоответствия типов сравниваемых данных:

```
SELECT Фамилия FROM Сотрудники WHERE Зарплата > 25000;
```

Данный запрос пытается получить список всех сотрудников, у которых зарплата больше 25 000. Однако в таблице `Сотрудники` столбец `Зарплата` содержит символьные, а не числовые данные. В условии запроса (оператор `WHERE`) сравниваются символьное и числовое значения. На практике данная проблема легко преодолима с помощью функции `CAST()` приведения значения к требуемому типу *(см. разд. 2.2.9)*. Кроме того, между некоторыми типами столбцов таблицы и данных, участвующих в SQL-выражениях, существуют соответствия, допускающие их совместное использование. Такие типы называют *согласованными*.

Различные СУБД поддерживают несколько отличающиеся наборы типов данных для столбцов таблиц базы данных. Аналогичная ситуация и с типами данных, поддерживаемых различными версиями и реализациями SQL. Вместе с тем, всегда имеются типы данных, которые поддерживаются всеми реализациями SQL.

В спецификации SQL:2003 признаны пять предопределенных общих типов, внутри которых могут быть подтипы:

- строковый (символьный):
  - `CHARACTER` (или `CHAR`);
  - `CHARACTER VARYING` (или `VARCHAR`);
  - `CHARACTER LARGE OBJECT` (или `CLOB`);
- числовой:
  - точные числовые типы:
    1. `INTEGER`;
    2. `SMALLINT`;
    3. `BIGINT`;
    4. `NUMERIC`;
    5. `DECIMAL`;
  - приблизительные числовые типы:
    1. `REAL`;

2. DOUBLE PRECISION;
3. FLOAT;

- логический (булевский) — BOOLEAN;
- даты-времени:
  - DATE;
  - TIME WITHOUT TIME ZONE;
  - TIME WITH TIME ZONE;
  - TIMESTAMP WITHOUT TIME ZONE;
  - TIMESTAMP WITH TIME ZONE;
- интервальный.

Кроме того, существуют особые типы: ROW (запись), ARRAY (массив) и MULTISET (мультимножество).

2.2.1. Строки

Строковые данные (последовательности символов) имеют три главных строковых типа. Для столбца таблицы можно указать тип CHARACTER (*n*) или CHAR (*n*) (строка фиксированной длины), где *n* — максимальное количество символов, содержащихся в строке. Если (*n*) не указано, то предполагается, что строка состоит из одного символа. Если в столбец типа CHARACTER (*n*) вводится $m < n$ символов, то оставшиеся позиции заполняются пробелами.

Тип данных CHARACTER VARYING (*n*) или VARCHAR (*n*) (строка переменной длины) применяется тогда, когда вводимые данные имеют различную длину и нежелательно дополнять их пробелами. При этом сохраняется только то количество символов, которое ввел пользователь. В данном случае указание максимального количества символов обязательно (в отличие от CHARACTER).

Данные типов CHARACTER и CHARACTER VARYING могут участвовать в одних и тех же строковых операциях.

Тип данных CHARACTER LARGE OBJECT (CLOB — большой символьный объект) используется для представления очень больших символьных строк (например, статей, книг и т. п.). В некоторых СУБД данный тип называется MEMO, а в других — TEXT. С данны-

ми этого типа можно выполнять не все операции, предусмотренные для типов CHARACTER и CHARACTER VARYING. Так, их нельзя использовать в операциях сравнения, за исключением равенства и неравенства. Кроме того, столбцы этого типа не могут быть первичными и внешними ключами, а также быть объявлены как имеющие уникальные значения. Иначе говоря, при создании таблиц с помощью оператора CREATE и объявлении столбцов типа CLOB нельзя использовать ключевые слова PRIMARY KEY, FOREIGN KEY и UNIQUE.

В следующем примере создается таблица с обычным символьным столбцом и столбцом типа CLOB, значения которого могут содержать 100 000 символов:

```
CREATE TABLE myTable
(
    FIELD1 CHARACTER (60),
    FIELD2 CLOB (100000)
);
```

Здесь оператор CREATE TABLE создает таблицу с именем myTable, которая состоит из двух столбцов с именами FIELD1 и FIELD2, типы столбцов указаны рядом с их именами.

Различные языки используют различные наборы символов. Даже английский и немецкий наборы отличаются, не говоря уж о русском и китайском. Система может быть настроена на некоторый набор символов, принимаемый по умолчанию. Однако при этом можно использовать и другие национальные символьные наборы. Так, в следующем примере создается таблица, в которой столбец FIELD1 объявляется как строковый с набором символов, принятым по умолчанию, а столбец FIELD2 — как строковый с греческим набором символов:

```
CREATE TABLE myTable
(
    FIELD1 CHARACTER (60),
    FIELD2 CHARACTER VARYING (80) CHARACTER SET GREEK
);
```

Значения строкового типа в SQL-выражениях заключаются в одинарные кавычки. Например, `'Иванов Иван Иванович'`, `'12345 рублей'`, `'тел. (812) 123-4567'`. Пустая строка не содержит ни одного символа и имеет вид: `''`. Строка, содержащая один или более пробелов, не является пустой.

Иногда бывает так, что строковый столбец в таблице содержит только числа (точнее, строки, содержащие цифры, знаки числа и разделительные точки). Чтобы использовать такие данные в операциях с числами, необходимо привести данные одного типа к другому типу с помощью функции `CAST()` *(см. разд. 2.2.9)*.

Значения типа `CHARACTER` и `CHARACTER VARYING` (`VARCHAR`) совместимы в том смысле, что они могут участвовать как операнды в строковых операциях и операциях сравнения.

2.2.2. Числа

Числовой тип данных может быть двух видов — точный и приблизительный. Точные числовые типы позволяют точно выразить значение числа. Некоторые величины имеют очень большой диапазон значений, и в таких случаях достаточно ограничиться некоторым приближенным их представлением с учетом технических возможностей компьютера (размеров регистра).

К точным числовым относятся следующие пять типов:

- `INTEGER` — целое (без дробной части) число. Количество разрядов (точность) зависит от реализации SQL. В некоторых реализациях числа этого типа лежат в диапазоне от –2 147 483 648 до 2 147 483 647 (четырехбайтное целое число);
- `SMALLINT` — малое целое число. Количество разрядов зависит от реализации SQL, но не больше количества разрядов `INTEGER` в этой же реализации. В некоторых реализациях числа этого типа лежат в диапазоне от –32 768 до 32 767 (двухбайтное целое число);
- `BIGINT` — большое целое число. Количество разрядов зависит от реализации SQL и превышает количество разрядов числа типа `INTEGER`;

- `NUMERIC (x, y)` — число, в котором всего `x` разрядов (точность), из которых `y` разрядов (масштаб) отводится для дробной части. Если `y` не указано (`NUMERIC (x)`), то для дробной части отводится количество разрядов, установленное в системе по умолчанию. Если не указаны ни `x`, ни `y` (`NUMERIC`), то принимаются обе эти величины, установленные по умолчанию. Например, если указан тип `NUMERC (6, 2)`, то максимальное значение числа равно `9999.99`;
- `DECIMAL (x, y)` — десятичное число, в котором всего `x` разрядов, из которых `y` разрядов отводятся для дробной части. Если `x` или/и `y` не указаны, то принимаются значения по умолчанию. Этот тип очень похож на `NUMERIC`. Отличие состоит в том, что если в `DECIMAL (x, y)` указанные `x` и `y` меньше, чем допустимые реализацией SQL, то будут использоваться последние. Если `x` и `y` не указаны, то применяется система умолчаний. Например, вы задали для столбца тип `DECIMAL (6, 2)`. Если реализация SQL позволяет, то в этот столбец можно ввести числа, превышающие `9999.99`. В отличие от `DECIMAL (x, y)`, тип `NUMERIC (x, y)` жестко задает диапазон возможных значений числовой величины.

К приблизительным числовым типам относятся следующие три типа:

- `REAL` — вещественное число одинарной точности с плавающей разделительной точкой (эта точка "плавает", появляясь в различных местах числа). Например, `5.25`, `5.257`, `5.2573`. Точность представления числа зависит от реализации SQL и оборудования. Например, 32-битовый компьютер дает бóльшую точность, чем 16-битовый;
- `DOUBLE PRECISION` — вещественное число двойной точности с плавающей разделительной точкой. Точность представления числа зависит от реализации SQL и оборудования. Применяется для представления научных данных (например, результатов измерений) в широком диапазоне значений, т. е. как очень малых (близких к 0), так и очень больших;
- `FLOAT (x)` — вещественное число с плавающей разделительной точкой и минимальной точностью `x`, занимающее не более

8 байтов. Если компьютер может поддержать указанную точность, используя аппаратную одинарную точность, то система будет использовать арифметику одинарной точности. Если указанная точность требует арифметики с двойной точностью, то система будет использовать ее. Данный тип следует применять, если предполагается возможность переноса базы данных на другую аппаратную платформу, отличающуюся размерами регистров. Пример значения типа `FLOAT`: `5.318E-24` (т. е. `5.318`, умноженное на `10` в степени `-24`). Такую же форму представления имеют и числа типа `REAL` и `DOUBLE PRECISION`.

При создании таблиц целочисленные типы применяются для столбцов, содержащих разного рода идентификаторы, например, номера (коды) клиентов, товаров, заказов и т. п. Разумеется, если содержимое столбца должно быть целым числом (например, количество ящиков, бутылок, штук и т. п.), то тип этого столбца естественно определить как `INTEGER`, `SMALLINT` или `BIGINT`.

Допустим, в таблице `Клиенты` имеется столбец `ID_клиента`, содержащий уникальные идентификаторы клиентов. Если количество клиентов не превышает 32 000, то тип столбца можно определить как `SMALLINT`. Если в вашей таблице будут храниться сведения о сотнях тысяч клиентов, то тип столбца `ID_клиента` следует определить как `INTEGER`.

Если столбец в проектируемой таблице должен содержать числа с дробной частью, то для него можно задать какой-нибудь нецелочисленный тип. Если вы не уверены, что применить: точные числовые типы или приблизительные, выбирайте точные (`NUMERIC`, `DECIMAL`). Они требуют меньше ресурсов и дают точные результаты. Если в столбце предполагается хранить данные из очень широкого диапазона (и очень малые, и очень большие числа), то используйте приблизительные типы данных (`FLOAT`, `REAL`).

Обратите внимание, что строка, содержащая число (например, `'12345.47'`), является данным строкового, а не числового типа. Чтобы в SQL-выражениях сравнивать строковые и числовые данные, необходимо один из типов привести к другому с помощью функции `CAST()`.

2.2.3. Логические данные

В этой части математической логики, основоположником которой был английский математик Джон Буль, данные имеют только два значения — ИСТИНА и ЛОЖЬ, обозначаемые как `true` и `false` соответственно. Данные логического типа получаются в результате операций сравнения. Например, результатом вычисления выражения $3 < 5$ является ИСТИНА, а выражения $2 + 3 = 10$ — ЛОЖЬ.

В SQL тип данных `BOOLEAN` (булевский) имеет три значения — `true`, `false` и `unknown`. Значение `unknown` (неизвестное) было введено для обозначения результата, получающегося при сравнении со значением `NULL` (неопределенное). Если пользователь еще не ввел в ячейку таблицы никакого значения, то эта "пустая" ячейка содержит значение `NULL`, интерпретируемое как неизвестное или неопределенное значение.

Результатом любой операции сравнения `true` или `false` с `NULL` или с `unknown` всегда является `unknown`.

В SQL-выражениях логические значения заключаются в кавычки, например, `'TRUE'` или `'true'`.

2.2.4. Дата и время

Тип `DATA` (дата) предназначен для хранения значений даты, элементы которых расположены в следующем порядке: год (4 цифры), дефис (-), месяц (2 цифры), дефис, день (2 цифры). Таким образом, значения даты занимают 10 позиций, например, `2005-10-02`.

Данные этого типа могут содержать любую дату с 0001 года по 9999 год.

Для представления времени предусмотрены два типа:

- `TIME WITHOUT TIME ZONE` (время без часового пояса) предназначен для хранения значений времени, элементы которых расположены в следующем порядке: часы, двоеточие, минуты, двоеточие, секунды. Часы и минуты представляются двумя циф-

рами, а секунды могут быть представлены двумя и более цифрами (если требуется дробная часть), например `18:35:19.547`. Длина дробной части секунд зависит от реализации, но внутреннее представление времени должно иметь не менее 6 цифр. По умолчанию время данного типа представляют без дробной части секунд. Чтобы указать, что время должно быть представлено с *n* цифрами после разделительной точки, достаточно использовать такой синтаксис: `TIME WITHOUT TIME ZONE (n)`. Например, чтобы кроме секунд указывались еще и миллисекунды, следует определить тип как `TIME WITHOUT TIME ZONE (3)`. Длина данных рассматриваемого типа без дробной части равна 8 символам, а с дробной частью — 9 плюс количество цифр после разделительной точки. Для задания времени без указания часового пояса с использованием установок по умолчанию можно использовать короткий синтаксис — `TIME`;

- `TIME WITH TIME ZONE` (время с часовым поясом) — такой же тип данных, как и `TIME WITHOUT TIME ZONE`. Отличие заключается лишь в том, что к значению времени добавляется еще и информация о разности между местным и всемирным временем. Всемирное время (Universal Time Coordinated, UTC) — это время по Гринвичу, т. е. время нулевого меридиана, проходящего через г. Гринвич в Великобритании (Greenwich Mean Time, GMT). Значение разности между локальным и всемирным временем находится в диапазоне от –12:59 до 13:00. Длина данных рассматриваемого типа равна длине данных типа `TIME WITHOUT TIME ZONE` плюс 6, поскольку дополнительная информация о разности времен занимает 6 позиций (дефис, знак (+) или (–), 2 цифры для часов, двоеточие, 2 цифры для минут).

Для одновременного представления даты и времени служат следующие два типа:

- `TIMESTAMP WITHOUT TIME ZONE` (дата и время без часового пояса). Элементы данных этого типа имеют такие же характеристики, как и для данных типа `DATE` и `TIME WITHOUT TIME ZONE`, за исключением одного: данные типа `TIMESTAMP WITHOUT`

TIME ZONE по умолчанию имеют 6 цифр в дробной части секунд, а не 0, как в типе TIME WITHOUT TIME ZONE. Для указания количества цифр в дробной части используется синтаксис TIMESTAMP WITHOUT TIME ZONE (*n*). Если дробной части нет, то данные занимают 19 позиций: 10 позиций для даты, один пробел и 8 позиций для времени. Если определена дробная часть, то длина данных равна 20 плюс количество цифр в дробной части секунд;

- TIMESTAMP WITH TIME ZONE (дата и время с часовым поясом) — такой же тип данных, как и TIMESTAMP WITH TIME ZONE. Отличие состоит в том, что к значению времени добавляется еще и информация о разности между местным и всемирным временем (см. TIME WITH TIME ZONE). Дополнительная информация занимает 6 позиций. Данные типа TIMESTAMP WITH TIME ZONE без дробной части занимают 25 позиций, с дробной частью — 26 плюс количество цифр в дробной части секунд.

Чтобы представить в SQL-выражении дату, время или дату-время, необходимо использовать функцию CAST() приведения к заданному типу. Допустим, в таблице Продажи имеется столбец Дата типа DATA. Чтобы получить сведения из этой таблицы за период после 2005-09-30, следует выполнить такой запрос:

```
SELECT * FROM Продажи WHERE Дата > CAST('2005-09-30' AS DATE);
```

Здесь строка, содержащая дату, приводится к типу DATA, и полученный результат участвует в операции сравнения с данными столбца Дата.

В языке SQL имеются три функции, которые возвращают текущие дату и время:

- CURRENT_DATE — возвращает текущую дату (тип DATE). Например, 2005-06-18;
- CURRENT_TIME(*число*) — возвращает текущее время (тип TIME). Целочисленный параметр *число* указывает точность представления секунд. Например, при *число* = 2 секунды будут представлены с точностью до сотых (две цифры в дробной части): 12:39:45.27;

- `CURRENT_TIMESTAMP(`*число*`)` — возвращает дату и время (тип `TIMESTAMP`), например `2005-06-18 12:39:45.27`. Целочисленный параметр *число* указывает точность представления секунд.

2.2.5. Интервалы

Интервал представляет собой разность между двумя значениями типа дата-время. SQL поддерживает два типа интервалов: год-месяц и день-время. Интервал типа год-месяц — это количество лет и месяцев между двумя датами, а интервал день-время — количество дней, часов, минут и секунд между двумя моментами в пределах одного месяца. Нельзя смешивать вычисления, использующие интервал год-день, с вычислениями, в которых используется интервал день-время.

Интервал времени можно задать двумя способами: в виде начального и конечного моментов или в виде начального момента и длительности, например:

- `(TIME '12:25:30', TIME '14:30:00')` — интервал, заданный начальным и конечным моментами;
- `(TIME '12:45:00', INTERVAL '5' HOUR)` — интервал, заданный начальным моментом и длительностью в часах.

Чтобы задать значение типа интервал, используется такой синтаксис:

```
INTERVAL 'длина' YEAR | MONTH | DAY | HOUR | MINUTE | SECOND
```

Здесь *длина* — длина интервала, после которой указывается единица измерения (возможные значения указаны через вертикальную черту):

- `YEAR` — год;
- `MONTH` — месяц;
- `DAY` — день;
- `HOUR` — час;
- `MINUTE` — минута;
- `SECOND` — секунда.

Например, для задания интервала длиной 15 дней следует использовать выражение `INTERVAL '5' DAY`.

Еще один пример использования интервалов приведен в *разд. 3.2.1* (предикат `OVERLAPS`).

2.2.6. Специальные типы данных

К специальным типам данных относятся следующие типы:

- `ROW` — запись;
- `ARRAY` — массив;
- `MULTISET` — мультимножество.

Использование этих типов данных нарушает принцип, согласно которому таблицы реляционной базы данных должны находиться в первой нормальной форме. Напомню, что любой столбец таблицы в первой нормальной форме не может содержать структурированные данные, например, набор полей: значения столбцов должны быть едиными и неделимыми. Тем не менее, жизнь неоднократно предлагает искушения отступить от теории в пользу эффективности. Если вы не отступаете от первой нормальной формы, то возникающие недоразумения в поддержке реляционных баз данных вы можете объяснить нарушениями законов реляционной теории. В противном случае для получения объяснений требуются более тщательные расследования происшедшего. Да, вне теории трудно жить в крайних случаях, хотя и более удобно в обычных обстоятельствах. Рассмотрим специальные типы данных подробнее.

Тип `ROW` (запись) позволяет объявить набор полей (запись) в качестве значения столбца таблицы. Таким образом, строки таблиц могут содержать записи.

Во многих базах данных имеются таблицы, содержащие сведения об адресах. Адрес, как известно, состоит из нескольких элементов, таких как почтовый индекс, страна, город, улица и т. д. Если в таблице завести один символьный столбец `Адрес`, в котором поместить все элементы адреса просто как символьную строку,

то дальнейшая работа с отдельными элементами адреса будет довольно хлопотным делом (вам придется использовать функции разбора строк). Поэтому на практике для адреса обычно заводят несколько столбцов, по одному на каждый элемент (например, столбцы Город, Улица и т. д.). Если же использовать тип ROW, то можно хранить все элементы адреса как значения одного столбца. В следующем примере сначала на основе типа ROW создается тип addr, а затем он назначается столбцу Адрес при создании таблицы Клиенты:

```
CREATE ROW TYPE addr
(
    PostCode VARCHAR (9),
    City     VARCHAR (30),
    Street   VARCHAR (30),
    House    VARCHAR (10)
);
CREATE TABLE Клиенты
(
    ID_клиента  INTEGER PRIMARY KEY,
    Имя         VARCHAR (25),
    Адрес       addr,
    Телефон     VARCHAR (15)
);
```

Тип данных ROW появился впервые в SQL:1999, а раньше без него прекрасно обходились.

Тип ARRAY (массив) также впервые появился в SQL:1999 и нарушает принцип первой нормальной формы, но несколько иначе, чем тип ROW. Тип ARRAY дает возможность одному из обычных типов иметь множество значений внутри значения одного табличного столбца. Например, клиент может иметь несколько номеров телефонов и вы можете пожелать хранить их все как одно многозначное значение единственного столбца Телефон, а не как обычные значения нескольких столбцов Телефон1, Телефон2

и т. п. Вот пример, в котором каждая ячейка столбца Телефон может содержать до трех номеров телефона:

```
CREATE TABLE Клиенты
(
    ID_клиента  INTEGER PRIMARY KEY,
    Имя         VARCHAR (25),
    Адрес       addr,
    Телефон     VARCHAR (15) ARRAY [3]
);
```

Следующее SQL-выражение добавляет в таблицу Клиенты новую запись и вводит в нее значения столбцов:

```
INSERT INTO Клиенты (Имя, Телефон)
VALUES ('Петров Петр Петрович', {'444-4444', '123-4567',
'777-8899'});
```

Тип данных MULTISET (мультимножество) появился впервые в SQL:2003. Если массив — это упорядоченное множество элементов, однозначно связанных с их местом в массиве посредством индекса, то мультимножество — неупорядоченная совокупность элементов. Этот тип данных используется очень редко.

2.2.7. Пользовательские типы данных

Пользовательские типы данных (User Defined Types, UDT) определяются пользователем по своему усмотрению для каких-то своих целей. Данная возможность впервые появилась в SQL:1999. Одно из важнейших направлений применения заключается в устранении некоторых несоответствий между основными типами данных SQL и типами данных в базовом языке, на котором пишутся приложения. К созданию своих типов данных обычно прибегают программисты, а не рядовые пользователи баз данных, даже владеющие языком SQL. Тем не менее, им также не помешает ознакомиться в общих чертах с этой возможностью SQL.

Отдельные типы

Простейшей формой пользовательских типов являются так называемые отдельные (различающиеся) типы. Они создаются на основе ранее определенных типов, например, основных типов SQL. Синтаксис создания отдельного типа данных такой:

```
CREATE DISTINCT TYPE имяТипа AS определенныйТип;
```

Определим в качестве примера тип MONEY для хранения денежных сумм. С этой целью можно воспользоваться уже определенным числовым типом DECIMAL:

```
CREATE DISTINCT TYPE MONEY AS DECIMAL (9,2);
```

Все отдельные типы, даже созданные на базе одного и того же типа, не могут сравниваться ни друг с другом, ни с базовым типом. Таким образом, данные типа MONEY нельзя сравнивать с данными типа DECIMAL, а также с данными других числовых и нечисловых типов. Создав специальный тип для денег, мы тем самым исключили возможность сравнивать денежные суммы с количеством других вещей. По аналогии с типом MONEY можно создать типы данных для различных валют, чтобы исключить возможность сравнения их просто как обычных чисел:

```
CREATE DISTINCT TYPE EURO AS DECIMAL (9,2);
CREATE DISTINCT TYPE USD AS DECIMAL (9,2);
CREATE DISTINCT TYPE RU AS DECIMAL (9,2);
```

Чтобы сравнить денежные суммы в различных валютах, придется сначала выполнить конвертацию одной валюты в другую. В SQL это можно сделать с помощью функции приведения типов CAST().

Допустим, имеется таблица Прайс_лист (Товар, Цена_USD, Цена_RU), в которой столбцы Цена_USD и Цена_RU определены как столбцы типов USD и RU соответственно. Эти столбцы содержат цены товаров в американских долларах и рублях. Предположим, что курс доллара к рублю изменился, и мы хотим изменить цены в рублях. Если бы столбцы Цена_USD и Цена_RU были одного и того же типа, то данную операцию можно было бы выполнить с помощью следующего SQL-выражения:

```
UPDATE Прайс_лист SET Цена_RU = k * Цена_USD;
```

Здесь k — коэффициент конвертации.

Но поскольку столбцы Цена_USD и Цена_RU имеют различные типы, то необходимо выполнить такое выражение:

```
UPDATE Прайс_лист
SET CAST(Цена_RU AS DECIMAL (9,2) = k * CAST(Цена_USD AS
DECIMAL (9,2));
```

Структурированные типы

Другая форма пользовательских типов данных — структурированные типы. Структурированный тип определяется не на основе какого-то базового (ранее определенного типа), а путем задания своих атрибутов и методов. Эта конструкция пришла в SQL из объектно-ориентированного программирования (ООП). Она представляет средство некой технологии, отличающейся от традиционной технологии применения SQL, но близкой к современной технологии ООП.

При создании структурированного типа СУБД автоматически создает для него три функции:

- функцию-конструктор с именем, совпадающим с именем создаваемого типа;
- функцию-мутатор, с помощью которой можно изменить значение атрибута создаваемого типа;
- функцию-наблюдатель, позволяющую узнать значение атрибута создаваемого типа.

Между структурированными типами может быть задана иерархия — отношение вложенности. Тип, содержащий другие типы, называется супертипом, а вложенные в него типы — подтипами.

Рассмотрим пример, в котором создаются некий тип и его подтип данных, таблица со столбцом созданного типа, а также блок операторов SQL, которые производят изменения данных в таблице. Вначале создадим тип данных Book, содержащий сведения (атрибуты) о книгах, затем создадим его подтип myBooks, который наследует атрибуты своего супертипа Books:

```
CREATE TYPE Books AS
    Title      CHAR (50),
```

```
    Author      CHAR (20),
    Publisher   CHAR (20),
    Year        INTEGER,
    Volume      INTEGER,
NOT FINAL;
```

Здесь в типе `Books` (книги) определяются атрибуты (имена и типы данных): `Title` (наименование), `Author` (автор), `Publisher` (издательство), `Year` (год издания), `Volume` (количество страниц). Ключевые слова `NOT FINAL` (не конец) означают, что данный тип имеет хотя бы один подтип.

Определим подтип `myBooks` для хранения данных, например, о моих любимых книгах:

```
CREATE TYPE myBooks UNDER Books FINAL;
```

По-русски это выглядит как:

СОЗДАТЬ ТИП `myBooks` ПОД `Books` ЗАКОНЧИТЬ;

Это означает, что создается конечный подтип, который не содержит других типов. Если бы потребовалось создать еще один тип, вложенный в тип `myBooks`, то в SQL-выражении создания подтипа вместо ключевого слова `FINAL` следовало бы написать `NOT FINAL`.

Итак, создан супертип `Books` и его подтип `myBooks`. Супертип `Books` понадобится в дальнейшем, быть может, и для каких-то других целей, но сейчас он нам нужен как родительский тип для конкретного типа `myBooks`. Обратите внимание, что здесь `myBooks` является просто копией типа `Books`. При желании мы могли бы добавить в него дополнительные свойства, которых нет в супертипе, но в данном случае мы обойдемся без этого.

Создадим таблицу, которая использует тип `myBooks`:

```
CREATE TABLE Книги
(
    Книга  myBooks,
    Цена   NUMERIC (6,2)
);
```

Теперь добавим новые записи в созданную таблицу Книги:

```
BEGIN
    DECLARE x myBooks;  /* объявление переменной x
                           типа myBooks*/
    SET x = myBooks();  /* Выполняем функцию-конструктор */
    /* Вызов функций-мутаторов */
    SET x = x.Title('HTML, скрипты и стили');
    SET x = x .Author('Дунаев Вадим');
    SET x = x.Publisher('БХВ-Петербург');
    SET x =x.Year(2005);
    SET x.Volume(832);
    /* Добавление новой записи с установкой значений столбцов */
    INSERT INTO Книги (x, 350.50);
END;
```

Здесь ключевые слова BEGIN (начало) и END (конец) окаймляют блок SQL-команд, в котором вызываются функции-мутаторы для установки значений атрибутов типа данных myBook, а затем выполняется SQL-оператор INSERT для добавления новой записи с установкой значений столбцов. Ключевые слова BEGIN и END окаймляют SQL-операторы так называемой составной команды *(см. разд. 10.1)*. Комментарии заключаются в символы (/*) и (*/).

2.2.8. Неопределенные значения

Если в ячейку (поле) таблицы вы вводите какие-то значения (буквы, цифры, пробелы или еще что-нибудь), то эти ячейки приобретают так называемые *определенные* значения. С определенными значениями, принадлежащими тому или иному типу, можно выполнять какие-то операции. Однако вы можете создать запись (строку) таблицы, ничего не вводя в ее поля (столбцы). Такая запись ничего не содержит. Запись в таблице, которая создана, но в которую не введены конкретные значения, называется *пустой*. В каждой ячейке пустой записи содержится так называемое неопределенное значение, обозначаемое через NULL. Значение NULL понимается как неопределенное или неизвестное.

Итак, каждый раз, когда вы добавляете в таблицу новую запись, не вводя в нее каких-либо конкретных значений, СУБД устанавливает в ее полях значение NULL.

Значение NULL вы можете интерпретировать так, как вам хочется, например, "еще не введено", "пока не известно". Однако независимо от смысловой интерпретации СУБД будет интерпретировать значение NULL вполне определенно. Задумайтесь, подойдет ли эта интерпретация к вашим конкретным целям. В обычных случаях это самая хорошая интерпретация. Но дело в том, как влияет значение типа NULL на результат его участия в тех или иных операциях.

Важно понимать, что число 0 или пустая строка '' являются вполне определенными значениями и отличаются от NULL, хотя визуально пустая строка и неопределенное значение могут отображаться одинаково — в виде пустой ячейки таблицы. Многие функции SQL возвращают в зависимости от обстоятельств какое-то определенное значение либо NULL. Это необходимо учитывать при составлении SQL-выражений, поскольку комбинация NULL и других вполне конкретных значений может дать тот или иной результат в зависимости от конкретной операции.

Таблица базы данных может иметь некоторые записи, имеющие неопределенные значения. Такие значения появляются по различным причинам, как уже было отмечено, однако при этом могут возникать неоднозначные ситуации при извлечении данных. Чтобы их избежать, необходимо помнить, что при сравнении любого определенного значения с NULL результат равен false, т. е. любое определенное значение не равно NULL. То же самое происходит при сравнении двух величин, значениями которых являются NULL, т. е. сравнение двух неопределенных величин дает в результате false.

При проектировании баз данных можно указать, что некоторые столбцы не могут иметь значение NULL, т. е. их значения обязательно должны быть определены (при определении столбца в операторе CREATE TABLE указывается NOT NULL). Однако во многих случаях значения NULL являются очень полезными. Так,

например, вы можете создать таблицу и ввести в нее сведения, которыми вы располагаете в данный момент, оставив другие значения пока неопределенными. Затем, по мере поступления новой информации, вы можете изменить значения NULL на те, которые стали известны.

В следующем примере из таблицы Клиенты выбираются все записи, в которых не определено имя клиента:

```
SELECT * FROM Клиенты WHERE Имя IS NULL;
```

2.2.9. Преобразование типов

Чтобы выполнить какую-либо операцию над данными различных типов, необходимо сделать преобразование типов. Точнее, необходимо выполнить приведение данных одного типа к другому типу, чтобы участвующие в операции данные были либо однотипными, либо их типы были соответствующими.

Соответствующими типами являются:

- строковые CHARACTER и CHARACTER VARYING;
- все числовые типы;
- дата, время, дата-время и соответствующие интервалы.

Соответствующие типы не обязательно приводить друг к другу.

Приведение значения одного типа к другому осуществляется с помощью функции CAST():

```
CAST(выражение AS тип);
```

Например:

```
CAST('1234.52' AS NUMERIC (9,2));

CAST('2005-10-03' AS DATE);

CAST(CURRENT_TIMESTAMP (2) AS CHAR (20));

SELECT 'Цена: ' || CAST(Цена AS CHAR(5)) FROM Продажи;
```

В последнем примере пара вертикальных черт означает операцию конкатенации строк *(см. разд. 4.3)*. Другие примеры использования функции CAST() будут приведены в *гл. 4*.

Глава 3

Простые выборки данных

Предположим, что реляционная база данных, состоящая из одной или нескольких таблиц, создана и вы к ней уже подключились. В этом случае типичной практической задачей является получение (извлечение) нужных данных. Например, может потребоваться просто просмотреть все содержимое какой-либо таблицы из базы данных или некоторых ее полей. При этом, возможно, вы захотите получить не все записи, а лишь те, которые удовлетворяют заданным условиям. Однако чаще возникает более интересная и сложная задача извлечения данных сразу из нескольких таблиц. Данные из двух и более таблиц необходимо скомпоновать в одну таблицу, чтобы представить ее для обозрения, анализа или последующей обработки. Язык SQL предоставляет для этого широкие возможности, которые мы и рассмотрим в этой главе.

В результате выполнения выражения на языке SQL (SQL-выражения) создается таблица, которая либо содержит запрошенные данные, либо пуста, если данных, соответствующих запросу, не нашлось. Эта таблица, называемая еще результатной, существует только во время сеанса работы с базой данных и не присоединяется к числу таблиц, входящих в базу данных. Иначе говоря, она не хранится на жестком диске компьютера подобно исходным таблицам базы данных, и поэтому ее еще называют *виртуальной*.

Примечание

Термин "виртуальный" в современной обычной речи означает "мнимый", "возможный", т. е. не "реальный". Однако исконное значение этого слова как раз противоположное — "не номинальный", "фактический", "реальный". Данная ситуация аналогична употреблению слова "оригинальный". Этот термин исконно означает "настоящий", "первоначальный", "подлинный", "исходный". Однако в обычной речи чаще используются значения "отличающийся", "из ряда вон выходящий", "особенный".

Выборка данных из нескольких таблиц, их обработка, а также использование подзапросов (запросов, которые нужны в качестве промежуточных для получения окончательного результата) относятся к теме сложных запросов. Сложные запросы будут рассмотрены в последующих главах, а здесь мы остановимся на задаче выборки данных из одной таблицы при относительно простых условиях отбора, группировки и сортировки записей. Тем не менее, операторы SQL, применяемые в простых запросах на выборку данных, используются и в сложных запросах, направленных не только на получение, но и на изменение данных. Начните с простого, чтобы потом было легко понять сложное. Я имею в виду, что материал данной главы относится к фундаментальным темам SQL, хотя многие пользователи баз данных могут им и ограничиться. Последнее вполне вероятно, поскольку материал этой главы сам по себе исключительно практичен.

Все SQL-выражения, предназначенные для выборки данных из существующих таблиц базы данных, начинаются с ключевого слова (оператора) SELECT (выбрать). Для уточнения запроса служат дополнительные операторы, такие как FROM (из), WHERE (где) и др. Сейчас важно понять и запомнить, что результатом выполнения запроса, сформулированного в виде SQL-выражения, является таблица, содержащая запрошенные данные. Эта таблица виртуальна в том смысле, что только представляет результаты запроса и не принадлежит к базе данных. Замечу также, что SQL позволяет изменять существующую базу данных — создавать и добавлять к ней новые таблицы, а также модифицировать и удалять уже существующие. Но об этом в следующих главах.

3.1. Основное SQL-выражение для выборки данных

Чтобы выбрать из таблицы базы данных требуемые записи, следует, по крайней мере, указать столбцы и имя этой таблицы. Это требование было бы естественно сформулировать так:

ВЫБРАТЬ такие-то столбцы ИЗ такой-то таблицы;

Разумеется, вам может потребоваться выбрать не все записи таблицы, а лишь те, которые отвечают некоторому условию. На практике именно так и бывает. Отложим пока рассмотрение формирования условий отбора записей, а сконцентрируем внимание на выборке всех записей из заданной таблицы. SQL-запрос к базе данных, результатом которого является таблица, полученная из указанной в запросе, но отличающаяся от нее тем, что содержит лишь указанные столбцы, выглядит так:

```
SELECT списокСтолбцов FROM списокТаблиц;
```

Операторы `SELECT` (выбрать) и `FROM` (из) в SQL-выражении, определяющем выборку данных, являются обязательными, т. е. ни один из них нельзя пропустить. SQL-выражение, содержащее только эти операторы, является основным выражением, определяющим запрос к базе данных на выборку данных. В результате выполнения этого запроса создается виртуальная таблица, содержащая указанные столбцы и все записи исходной таблицы.

Примечание

Оператор `SELECT` осуществляет проекцию отношения, указанного в выражении `FROM`, на заданное множество атрибутов (столбцов), указанное в выражении `SELECT`. Так, например, если исходная таблица `R` содержит столбцы `A1`, `A2`, ... , `An` (другими словами, таблица представляет некоторое отношение *R* (*A*1, *A*2, ... , *An*) над атрибутами *A*1, *A*2, ... , *An*), то оператор:

```
SELECT A1,A2,Ak FROM R;
```

реализует проекцию *R* [*A*1, *A*2, *Ak*] этого отношения на атрибуты *A*1, *A*2, *Ak* (*k* = 1, 2, ... , *n*). Понятие проекции отношения и соответствующие обозначения были рассмотрены в *гл. 1*.

В выражении `FROM` указывается список имен таблиц базы данных, из которых требуется выбрать данные. В простейшем случае

списокТаблиц содержит лишь одно имя таблицы. Если же таблиц несколько, то их имена в списке разделяются запятыми. Если в выражении FROM указано более одной таблицы, то результатная таблица получается из декартового произведения перечисленных в списке таблиц. Иногда это используется для специальных целей, но чаще всего в выражении FROM указывается только одна таблица.

Список столбцов — это перечень имен столбцов, разделенных запятой, как они определены в таблице, указанной в выражении FROM. Разумеется, можно указать все или только некоторые столбцы. Если вы хотите получить все столбцы таблицы, то вместо списка столбцов достаточно указать символ (*). Если в выражении FROM указано несколько таблиц, то в выражении SELECT имена столбцов должны содержать префиксы, указывающие, к какой именно таблице они относятся. Префикс отделяется от имени столбца точкой. Например, выражение Клиенты.Адрес означает столбец Адрес из таблицы Клиенты.

Тривиальный запрос, возвращающий все данные (все столбцы и все записи) из одной таблицы, формулируется так:

```
SELECT * FROM имяТаблицы;
```

Основное SQL-выражение может быть дополнено другими операторами, уточняющими запрос. Более подробно они будут рассмотрены в *разд. 3.2*. Чаще всего употребляется оператор WHERE (где), с помощью которого можно задать условие выборки записей (строк таблицы). Таким образом, если выражение SELECT задает столбцы таблицы, указанной в операторе FROM, то выражение WHERE определяет записи (строки) из этой таблицы. Выражение, определяющее запрос на выборку данных, находящихся в некоторой таблице, имеет следующий вид:

```
SELECT * FROM имяТаблицы WHERE условиеПоиска;
```

Условие, указанное в выражении WHERE, принимает одно из двух логических значений: true (ИСТИНА) или false (ЛОЖЬ). Другими словами, это логическое выражение. При обработке запроса условие проверяется для каждой записи таблицы. Если оно истинно для данной записи, то она выбирается и будет представле-

на в результатной таблице. В противном случае запись не выбирается и в результатную таблицу не попадает. Если выражение `WHERE` не указано в SQL-выражении, то результатная таблица будет содержать все записи из таблицы, заданной в выражении `FROM`. Таким образом, выражение `WHERE` определяет *фильтр* записей. Фильтр что-то пропускает в результатную таблицу, а что-то отбрасывает.

Примечание

Фильтр — одно из основных понятий в области работы с базами данных. В литературе иногда можно встретить различные его трактовки. Так, "отфильтровать записи" может означать "получить записи", а может наоборот — "отбраковать записи". Вопрос в том, что проходит через фильтр, а что остается. В этой книге я придерживаюсь значения "получить записи". Поэтому здесь понятия "отфильтровать", "пропустить через фильтр" или "наложить фильтр" всегда означают "выбрать записи, удовлетворяющие условию фильтра".

Сразу за оператором `SELECT` до списка столбцов можно применять ключевые слова `ALL` (все) и `DISTINCT` (отличающиеся), которые указывают, какие записи представлять в результатной таблице. Если эти ключевые слова не используются, то подразумевается, что следует выбрать все записи, что также соответствует применению ключевого слова `ALL`. В случае использования `DISTINCT` в результатной таблице представляются только уникальные записи. При этом если в исходной таблице находятся несколько идентичных записей, то из них выбирается только первая.

Примечание

В Microsoft Access кроме ключевых слов `ALL` и `DISTINCT` после `SELECT` можно использовать ключевое слово `TOP` с дополнительными параметрами. Выражение `TOP` *n* требует, чтобы в выборку данных попали только первые *n* записей, удовлетворяющих заданному условию запроса. Это ограничение условия поиска нужных записей, формулируемого в выражении `WHERE`. Если исходная таблица очень большая, то `DISTINCT` может ускорить получение ответа.

В Access можно использовать и выражение `TOP` *n* `PERCENT`, чтобы указать, что *n* выражается в процентах от общего количества

записей. Не трудно понять, что использование такого выражения направлено не на ускорение поиска, а на получение таблицы, избавленной от лишних данных.

Заголовки столбцов в результатной таблице можно переопределить по своему усмотрению, назначив для них так называемые *псевдонимы*. Для этого в списке столбцов после соответствующего столбца следует написать выражение вида: `AS` *`заголовок_столбца`*. Например:

```
SELECT ClientName AS Клиент, Address AS Адрес FROM Клиенты;
```

Псевдонимы также можно задать и для каждой таблицы после ключевого слова `FROM`. Для этого достаточно указать псевдоним через пробел сразу после имени соответствующей таблицы. Псевдонимы таблиц, более короткие, чем их имена, удобно использовать в сложных запросах. Например:

```
SELECT T1.Имя, T2.Адрес FROM Клиенты T1, Контакты T2;
```

3.2. Уточнения запроса

Основное SQL-выражение для выборки данных, описанное в *разд. 3.1*, имеет вид:

```
SELECT списокСтолбцов FROM списокТаблиц;
```

Такой запрос возвращает таблицу, полученную из указанной в операторе `FROM` (или из декартового произведения указанных таблиц, если их несколько), путем выделения в ней только тех столбцов, которые определены в операторе `SELECT`. Я уже упоминал, что для выделения требуемых записей (строк) исходной таблицы используется выражение, следующее за ключевым словом (оператором) `WHERE`. Оператор `WHERE` является наиболее часто используемым, хотя и не обязательным в SQL-выражении. Именно из-за популярности его можно считать основным компонентом SQL-выражения. Кроме `WHERE`, в SQL-выражениях используются и другие операторы, позволяющие уточнить запрос.

Для уточнения запроса на выборку данных служит ряд дополнительных операторов:

- `WHERE` (где) — указывает записи, которые должны войти в результатную таблицу (фильтр записей);
- `GROUP BY` (группировать по) — группирует записи по значениям определенных столбцов;
- `HAVING` (имеющие, при условии) — указывает группы записей, которые должны войти в результатную таблицу (фильтр групп);
- `ORDER BY` (сортировать по) — сортирует (упорядочивает) записи.

Эти операторы не являются обязательными. Их можно совсем не использовать, или использовать лишь некоторые из них, или все сразу. Если применяются несколько операторов, то в SQL-выражении они используются в указанном в списке порядке. Таким образом, запрос данных из таблицы с применением всех перечисленных операторов уточнения запроса имеет следующий вид:

```
SELECT списокСтолбцов FROM имяТаблицы
WHERE условиеПоиска
GROUP BY столбецГруппировки
HAVING условиеПоиска
ORDER BY условиеСортировки;
```

Порядок перечисления операторов в SQL-выражении не совпадает с порядком их выполнения. Однако знание порядка выполнения операторов поможет вам избежать многих недоразумений. Итак, перечисленные операторы SQL-выражения выполняются в следующем порядке, передавая друг другу результат в виде таблицы:

1. `FROM` — выбирает таблицу из базы данных; если указано несколько таблиц, то выполняется их декартово произведение и результирующая таблица передается для обработки следующему оператору.
2. `WHERE` — из таблицы выбираются записи, отвечающие условию поиска, и отбрасываются все остальные.

3. GROUP BY — создаются группы записей, отобранных с помощью оператора WHERE (если он присутствует в SQL-выражении); каждая группа соответствует какому-нибудь значению столбца группирования. Столбец группирования может быть любым столбцом таблицы, заданной в операторе FROM, а не только тем, который указан в SELECT.
4. HAVING — обрабатывает каждую из созданных групп записей, оставляя только те из них, которые удовлетворяют условию поиска; этот оператор используется только вместе с оператором GROUP BY.
5. SELECT — выбирает из таблицы, полученной в результате применения перечисленных операторов, только указанные столбцы.
6. ORDER BY — сортирует записи таблицы. При этом в условии сортировки можно обращаться лишь к тем столбцам, которые указаны в операторе SELECT.

Допустим, среди таблиц вашей базы данных имеется таблица Клиенты, которая содержит столбцы с именами: Имя, Адрес, Сумма_заказа и, возможно, какие-то другие (рис. 3.1). Семантика этой таблицы тривиальна. В ней фиксируются данные о клиентах и денежные суммы, которые они заплатили вашей фирме, пользуясь ее услугами.

Клиенты : таблица

Имя	Адрес	Сумма_заказа	Регион	Телефон
Иванов	197371 г.Санкт-Петербург пр. Королева, д.30, кв. 777	1000	Северо-запад	111-1111
Петров	197371 г.Санкт-Петербург Коломяжский пр., д.14	450	Северо-запад	111-2222
Сидоров	Москва, ул. Тверская, д. 25, кв. 375	3000	Москва	222-3333
Васильев	Селижарово, ул. Укромная, д. 22	200	Тверская область	555-1234
Захаров	Санкт-Петербург Невский пр., д.152	8600	Северо-запад	112-4321
Павлов	Иркутск, ул. Ленина, д.2	300	Иркутская область	888-0987

Запись: 1 из 6

Рис. 3.1. Таблица Клиенты

Внимание

Таблица, показанная на рис. 3.1, будет использоваться в качестве исходной во всех примерах и задачах данной главы.

Предположим, нас интересуют не все данные этой таблицы, а только те, которые касаются клиентов, заплативших фирме более 500 (сейчас не важно, в какой валюте производились оплаты). Точнее, нам нужны имена и адреса клиентов, которые заплатили фирме более 500 денежных единиц. Таким образом, нам необходимо получить не все, что содержится в таблице `Клиенты`, а лишь некоторую ее часть, как по столбцам, так и по записям. Для этой цели подойдет следующее SQL-выражение:

```
SELECT Имя, Адрес FROM Клиенты WHERE Сумма_заказа > 500;
```

Это SQL-выражение представляет собой запрос, который на естественном языке выглядит приблизительно так:

ВЫБРАТЬ СТОЛБЦЫ `Имя`, `Адрес` ИЗ ТАБЛИЦЫ `Клиенты` ГДЕ `Сумма_заказа` > 500;

Здесь из таблицы `Клиенты` выбираются записи, в которых значение столбца `Сумма_заказа` превышает 500. При этом в результатной таблице будут представлены только два столбца таблицы `Клиенты`: `Имя` и `Адрес` (рис. 3.2).

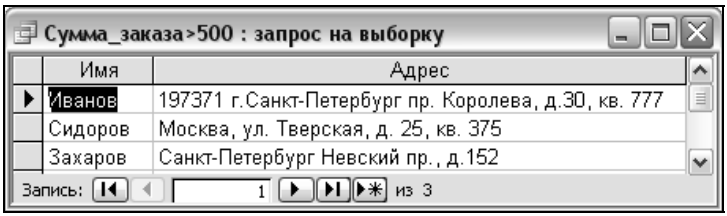


Рис. 3.2. Результат запроса `'WHERE Сумма_заказа > 500'`

Столбец `Сумма_заказа`, имеющийся в исходной таблице, в данном случае не выводится (отсутствует в результирующей таблице). Впрочем, если бы вам потребовалось увидеть конкретные суммы, превышающие 500, то для этого было бы достаточно указать в списке столбцов, следующем за оператором `SELECT`, еще и столбец `Сумма_заказа`.

Следующее SQL-выражение создает виртуальную таблицу, содержащую три столбца: `Регион`, `Имя` и `Адрес`. Из таблицы `Клиенты`

выбираются только те записи, в которых `Сумма_заказа` превышает 500, и они группируются по значениям столбцов `Регион`, `Имя` и `Адрес`. Это означает, что в результатной таблице записи, имеющие одинаковые значения в столбце `Регион`, будут расположены рядом друг с другом. Наконец, все записи в результатной таблице упорядочиваются по значениям столбца `Имя` (рис. 3.3).

```
SELECT Регион, Имя, Адрес FROM Клиенты
WHERE Сумма_заказа > 500
GROUP BY Регион, Имя, Адрес
ORDER BY Имя;
```

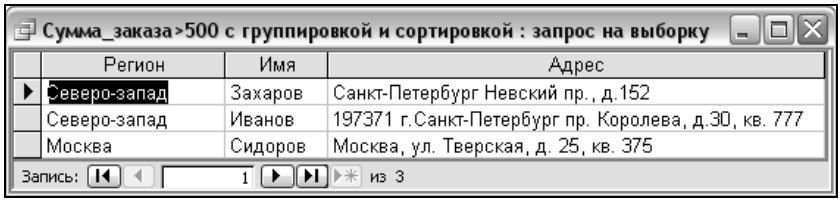


Рис. 3.3. Результат запроса `'WHERE Сумма_заказа > 500'` с группировкой `'GROUP BY Регион, Имя, Адрес'` и сортировкой `'ORDER BY Имя'`

Операторы группировки (`GROUP BY`) и сортировки (`ORDER BY`) будут подробно рассмотрены далее в *разд. 3.2.2—3.2.4*. Здесь я лишь показал на примере, что оператор выборки записей из исходной таблицы может соседствовать с другими SQL-операторами.

3.2.1. Оператор *WHERE*

Условия поиска в операторе `WHERE` (где) являются логическими выражениями, т. е. принимающими одно из двух возможных значений — `true` (ИСТИНА) или `false` (ЛОЖЬ). Например, выражение `Сумма_заказа > 500` является истинным (имеет значение `true`), если в текущей записи таблицы значение столбца `Сумма_заказа` превышает 500. В противном случае это выражение ложно (имеет значение `false`). Одно и то же логическое выражение может быть истинным для одних записей и ложным для других. Вообще говоря, в SQL логические выражения могут прини-

мать еще и неопределенное значение. Это происходит тогда, когда в выражении некоторые элементы имеют значение `NULL`. Таким образом, в SQL мы имеем дело не с классической двузначной, а с трехзначной логикой.

Напомню, что выражение, следующее за оператором `WHERE`, возвращает одно из трех значений: `true`, `false` или `NULL`. При выполнении запроса (SQL-выражения) логическое выражение `WHERE` применяется ко всем записям исходной таблицы. Если оно истинно для данной записи исходной таблицы, то эта запись выбирается и будет представлена в результатной таблице; в противном случае запись не попадет в результатную таблицу.

При составлении логических выражений используются специальные ключевые слова и символы операций сравнения, которые называют предикатами. Например, в выражении `Сумма_заказа > 500` применен предикат сравнения (>).

Внимание

Выражения с оператором `WHERE` используются не только при выборке данных (т. е. с оператором `SELECT`), но и при вставке, модификации и удалении записей. Таким образом, материал данного раздела имеет значение, выходящее за границы рассматриваемой темы о выборке данных.

Наиболее часто используются предикаты сравнения, такие как (=), (<), (>), (<>), (<=) и (>=). Однако имеются и другие. Далее приведен список всех предикатов:

- предикаты сравнения: (=), (<), (>), (<>), (<=), (>=);
- `BETWEEN`;
- `IN`, `NOT IN`;
- `LIKE`, `NOT LIKE`;
- `IS NULL`;
- `ALL`, `SOME`, `ANY`;
- `EXISTS`;
- `UNIQUE`;
- `DISTINCT`;

- OVERLAPS;
- MATCH;
- SIMILAR.

Предикаты сравнения

Предикаты сравнения, называемые также операторами сравнения, перечислены в табл. 3.1.

Таблица 3.1. *Предикаты сравнения*

Символ	Описание
=	Равно
<>	Не равно
<	Меньше
<=	Меньше или равно (не больше)
>	Больше
>=	Больше или равно (не меньше)

Примечание

В обычных процедурных языках программирования перечисленные символы сравнения называются операторами сравнения, а не предикатами. Если вы ознакомились с содержанием *гл. 1*, то нетрудно понять, что, например, выражение *x* > *y*, понимаемое как выражение сравнения величин *x* и *y* с использованием оператора (>) (больше), можно было бы записать как некую функцию в виде > (*x*, *y*). Поскольку эта функция возвращает логическое значение после подстановки вместо переменных *x* и *y* некоторых значений, то она и есть предикат. Выражение *x* > *y* также возвращает логическую величину после подстановки значений вместо переменных *x* и *y*. Таким образом, как бы мы ни называли все, что связано с символами сравнения, суть остается неизменной.

BETWEEN

Предикат BETWEEN (между) позволяет задать выражение проверки вхождения какого-либо значения в диапазон, определяемый граничными значениями. Например:

```
WHERE Сумма_заказа BETWEEN 100 AND 750
```

Здесь ключевое слово AND представляет собой логический союз И. Граничные значения (в примере это 100 и 750) входят в диапазон. Причем первое граничное значение должно быть не больше второго.

Эквивалентным приведенному является выражение с предикатами сравнения:

```
WHERE Сумма_заказа >= 100 AND Сумма_заказа <= 750
```

Кроме данных числового типа, в выражениях с BEETWEEN можно использовать данные следующих типов: символьные, битовые, даты-времени. Так например, чтобы выбрать записи, в которых имена клиентов находятся в диапазоне от А до Ж, можно использовать такое выражение:

```
SELECT Имя, Адрес FROM Клиенты
WHERE Имя BETWEEN 'A' AND 'Ж';
```

IN и NOT IN

Предикаты IN (в) и NOT IN (не в) применяются для проверки вхождения какого-либо значения в заданный список значений. Например, для выборки записей о клиентах из некоторых регионов можно использовать такое выражение:

```
SELECT Имя, Адрес FROM Клиенты
WHERE Регион IN ('Северо-запад', 'Ставропольский край',
'Иркутская область');
```

Если требуется получить данные о всех клиентах не из Москвы и Северо-Запада, то можно использовать предикат NOT IN:

```
SELECT Имя, Адрес FROM Клиенты
WHERE Регион NOT IN ('Москва', 'Санкт-Петербург');
```

LIKE и NOT LIKE

Предикаты LIKE (похожий) и NOT LIKE (не похожий) применяются для проверки частичного соответствия символьных строк. Например, столбец Телефон в некоторой таблице содержит полные номера телефонов, а вам требуется выбрать лишь те записи, в которых номера телефонов начинаются с 348 или содержат такое сочетание цифр.

Критерий частичного соответствия задается с помощью двух символов-масок: знака процента (%) и подчеркивания (_). Знак процента означает любой набор символов, в том числе и пустой, а символ подчеркивания — любой одиночный символ.

Например, чтобы выбрать записи о клиентах, у которых номера телефонов начинаются с 348, можно использовать такое выражение:

```
SELECT Имя, Адрес, Телефон FROM Клиенты
WHERE Телефон LIKE '348%';
```

Допустим, столбец Адрес содержит полный почтовый адрес (индекс, название города, улицы и т. д.). Если вам требуется выбрать записи о клиентах, проживающих в Санкт-Петербурге, то для этого подойдет следующее выражение:

```
SELECT Имя, Адрес, Телефон FROM Клиенты
WHERE Адрес LIKE '%Санкт-Петербург%';
```

Если вы хотите исключить всех клиентов, проживающих в Москве, то воспользуйтесь таким выражением:

```
SELECT Имя, Адрес, Телефон FROM Клиенты
WHERE Адрес NOT LIKE '%Москва%';
```

Возможно, вам потребуется выбрать записи, содержащие символы процента и/или подчеркивания. Тогда необходимо, чтобы такие символы воспринимались интерпретатором SQL не как символы-маски. Чтобы знак процента или подчеркивания воспринимался буквально, перед ним необходимо указать специальный управляющий символ. Этот символ можно определить произвольно, лишь бы он не встречался в качестве элемента данных. В следующем примере показано, как это можно сделать:

```
SELECT Имя, Адрес, Процент_скидки FROM Клиенты
WHERE Процент_скидки LIKE '20#%'
ESCAPE '#';
```

Здесь за ключевым словом ESCAPE указывается символ, который используется в качестве управляющего. Таким же способом можно отключить и сам управляющий символ.

IS NULL

Предикат `IS NULL` применяется для выявления записей, в которых тот или иной столбец не имеет значения. Например, для получения записей о клиентах, для которых не указан адрес, можно использовать следующее выражение:

```
SELECT Имя, Адрес, Регион FROM Клиенты
WHERE Адрес IS NULL;
```

Для получения записей, в которых столбец `Адрес` содержит некоторые определенные значения (т. е. отличные от `NULL`), можно использовать аналогичное выражение, но с логическим оператором `NOT` (не):

```
SELECT Имя, Адрес, Регион FROM Клиенты
WHERE Адрес IS NOT NULL;
```

Не следует использовать предикаты сравнения с `NULL`, такие как `Адрес = NULL`.

Предикаты для вложенных запросов

В выражении `WHERE`, кроме перечисленных предикатов, могут также использоваться выражения с оператором `SELECT`. Любое выражение, начинающееся с оператора `SELECT`, является запросом к базе данных. Если в выражении встречается еще хотя бы один оператор `SELECT`, то он задает запрос, вложенный в первый. Вложенные запросы также называют подзапросами.

Вложенный запрос является обычным запросом, таким же, как и рассмотренные ранее. Он возвращает таблицу (набор записей), которая, так или иначе, используется для формирования ответа на основной запрос. Так, например, подзапрос используется, когда для выборки данных в одной таблице необходимо выполнить проверки по другой таблице. Для этой цели подходят перечисленные далее специальные предикаты. В этой главе я лишь отмечу их основные характеристики (для справки), варианты применения будут рассмотрены в *гл. 5*.

ALL, SOME, ANY

Предикаты `ALL` (все), `SOME` (некоторый), `ANY` (любой) в действительности представляют собой кванторы, известные в математи-

ческой логике как кванторы всеобщности и существования. ALL — квантор всеобщности, а SOME и ANY, являющиеся синонимами в SQL, — кванторы существования. Заметим, что в переводе на русский слово ANY следовало бы понимать как квантор всеобщности ("любой" означает "все"), однако в английском языке есть различные варианты значений этого слова.

Применение ключевого слова ALL следует понимать как "для всех" или "для каждого". Ключевые слова SOME и ANY следует понимать как "хотя бы какой-нибудь один".

Как бы то ни было, в языке SQL ключевые слова SOME и ANY имеют одинаковый смысл, отличающийся от ALL. Примеры запросов с кванторами будут рассмотрены в *разд. 5.1*.

Примечание

Выражения с ключевыми словами ALL, SOME (ANY) соответствуют логическим выражениям с кванторами и, как таковые, могут называться предикатами.

EXISTS

Обработка данных часто состоит из нескольких этапов. Так, сначала производится некоторая выборка данных, а затем выполняются какие-то манипуляции с ней. Однако, выполняя запрос на выборку, мы далеко не всегда можем быть уверенными, что ответ содержит хотя бы одну непустую строку. Если ответ на запрос пуст, то бессмысленно производить дальнейшую обработку данных. Таким образом, полезно знать, содержит ли ответ на запрос какие-либо данные. Для этого предназначен предикат EXISTS (существует). Он становится истинным только тогда, кода результатная таблица, полученная в ответ на запрос, содержит хотя бы одну запись. Пример запроса с предикатом EXISTS будет рассмотрен в *разд. 5.1*.

UNIQUE

Предикат UNIQUE (уникальный) имеет такой же смысл, как и EXISTS, но при этом для его истинности требуется, чтобы все записи в результатной таблице не только существовали, но и были уникальны (т. е. не повторялись).

DISTINCT

Предикат `DISTINCT` (отличающийся, особый) почти такой же, как и `UNIQUE`. Отличие этих предикатов обнаруживается применительно к значениям `NULL`. Так, если в результатной таблице все записи уникальны (предикат `UNIQUE` истинен), то и предикат `DISTINCT` тоже истинен (т. е. если все записи уникальны, то они и отличающиеся). С другой стороны, если в результатной таблице имеются хотя бы две неопределенные записи, то предикат `DISTINCT` ложен, хотя предикат `UNIQUE` истинен.

OVERLAPS

Предикат `OVERLAPS` (перекрывает) используется для определения, перекрываются ли два интервала времени. Интервал времени можно задать двумя способами: в виде начального и конечного моментов или в виде начального момента и длительности. Далее приведены примеры задания интервала времени:

- `(TIME '12:25:30', TIME '14:30:00')` — интервал, заданный начальным и конечным моментами;
- `(TIME '12:45:00', INTERVAL '2' HOUR)` — интервал, заданный начальным моментом и длительностью в часах.

Выражение с предикатом `OVERLAPS` можно записать, например, так:

```
(TIME '12:25:30', TIME '14:30:00') OVERLAPS (TIME
'12:45:00', INTERVAL '2' HOUR)
```

Поскольку временные интервалы в данном примере пересекаются, то предикат `OVERLAPS` возвращает значение `true`.

Подробнее об интервалах было рассказано в *разд. 2.2.5*.

MATCH

Предикат `MATCH` применяется для проверки сохранения ссылочной целостности при модификации данных, т. е. при добавлении, изменении и удалении записей *(см. разд. 6.4)*.

SIMILAR

Предикат `SIMILAR` (подобный) применяется для проверки частичного соответствия символьных строк. Эту же задачу можно ре-

шить и с помощью предиката LIKE, однако в ряде случаев SIMILAR более эффективен.

Предположим, что в некоторой таблице имеется столбец OC, содержащий названия операционных систем. Нужно выбрать записи, соответствующие Windows NT, Windows XP и Windows 98. Тогда в выражении запроса можно использовать такой оператор WHERE:

```
WHERE OC SIMILAR TO '(Windows (NT|XP|98))';
```

Предикат впервые появился в SQL:1999.

3.2.2. Оператор *GROUP BY*

Оператор GROUP BY (группировать по) служит для группировки записей по значениям одного или нескольких столбцов. Если в SQL-выражении используется оператор WHERE, задающий фильтр записей, то оператор GROUP BY находится и выполняется после него. Для определения, какие записи должны войти в группы, служит оператор HAVING, используемый совместно с GROUP BY *(см. разд. 3.2.3)*. Если оператор HAVING не применяется, то группировке подлежат все записи, отфильтрованные оператором WHERE. Если WHERE не используется, то группируются все записи исходной таблицы. О порядке выполнения операторов уже говорилось в начале *разд. 3.2*.

Допустим, что на основе таблицы о клиентах (см. рис. 3.1) требуется сгруппировать данные о суммах заказов клиентов по регионам. Для этого можно воспользоваться следующим SQL-выражением:

```
SELECT Регион, Сумма_заказа FROM Клиенты
GROUP BY Регион;
```

На рис. 3.4 показана результатная таблица на фоне исходной таблицы Клиенты. Обратите внимание, что записи с одинаковыми названиями регионов расположены рядом друг с другом (в одной группе).

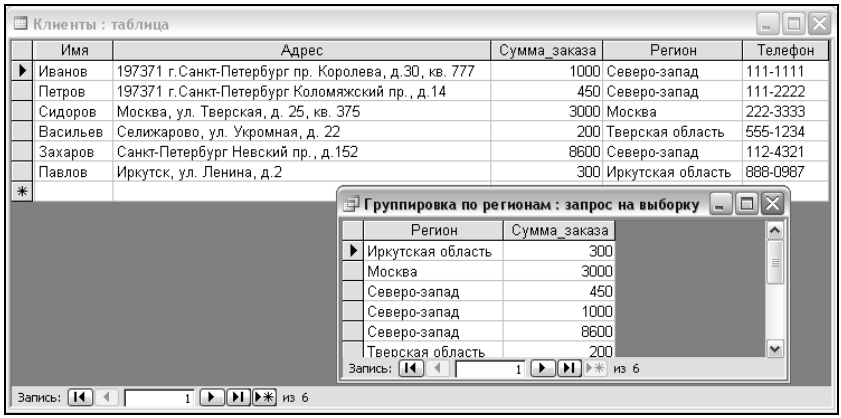


Рис. 3.4. Результат запроса сумм заказов с группировкой по регионам

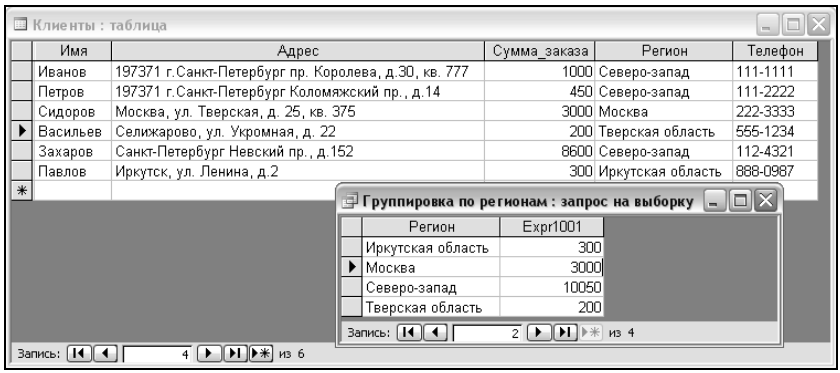


Рис. 3.5. Результат запроса итоговых сумм заказов по регионам

Чтобы получить таблицу, в которой суммы заказов подытожены по регионам, потребуется использовать итоговую функцию `SUM()` *(см. разд. 4.1)* и группировку по регионам:

```
SELECT Регион, SUM(Сумма_заказа) FROM Клиенты
GROUP BY Регион;
```

Здесь в выражении `SELECT` указаны обычный столбец таблицы `Клиенты` и итоговая функция `SUM()`, вычисляющая сумму значений столбца `Сумма_заказа`. Поскольку группировка задана по столбцу `Регион`, то функция `SUM(Сумма_заказа)` вычисляет суммы значений столбца `Сумма_заказа` для каждого значения столб-

ца Регион. На рис. 3.5 показана результатная таблица на фоне исходной таблицы Клиенты. Обратите внимание, что в этой таблице названия регионов не повторяются.

Оператор GROUP BY собирает записи в группы и упорядочивает (сортирует) группы по алфавиту (точнее, по ASCII-кодам символов). Это обстоятельство следует иметь в виду перед тем, как принять решение об использовании оператора сортировки ORDER BY *(см. разд. 3.2.4)*.

3.2.3. Оператор *HAVING*

Оператор HAVING (имеющие, при условии) обычно применяется совместно с оператором группировки GROUP BY и задает фильтр записей в группах. Правила его формирования такие же, что и для оператора WHERE.

Предположим, что из таблицы Клиенты требуется выбрать данные о регионах и суммах заказов, сгруппированные по регионам и такие, в которых сумма заказа превышает 500. Иначе говоря, требуется сгруппировать данные с ограничением записей, входящих в группы. Запрос, выполняющий это задание, имеет вид:

```
SELECT Регион, Сумма_заказа FROM Клиенты
GROUP BY Регион, Сумма_заказа
HAVING Сумма_заказа > 500;
```

Результат выполнения этого запроса представлен на рис. 3.6.

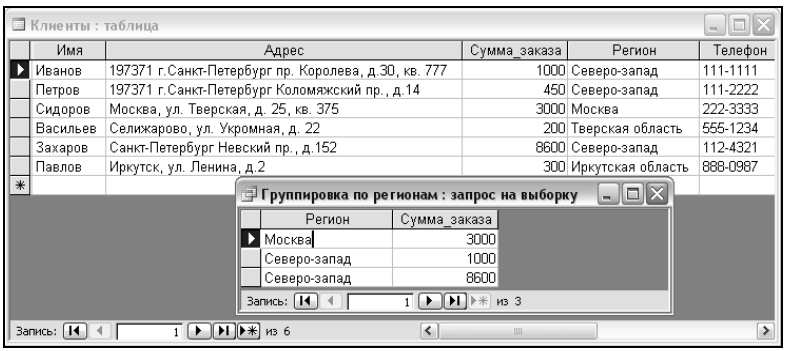


Рис. 3.6. Результат запроса с группировкой по регионам и ограничением по суммам заказов

Если в SQL-выражении оператора `GROUP BY` нет, то оператор `HAVING` применяется ко всем записям, возвращаемым оператором `WHERE`. Если же отсутствует и `WHERE`, то `HAVING` действует на все записи таблицы.

3.2.4. Оператор *ORDER BY*

Оператор `ORDER BY` (сортировать по) применяется для упорядочивания (сортировки) записей. Если он используется в запросе, то в самом конце запроса. Этот оператор сортирует строки всей таблицы или отдельных ее групп (в случае применения оператора `GROUP BY`). Если в выражении запроса оператора `GROUP BY` нет, то оператор `ORDER BY` рассматривает все записи таблицы как одну группу.

Вслед за ключевым словом `ORDER BY` указывается столбец, по значениям которого следует произвести сортировку. После имени столбца можно указать ключевое слово, задающее порядок (режим) сортировки:

- `ASC` — по возрастанию (*ascending*). Это значение принято по умолчанию, поэтому если необходима сортировка, например, в алфавитном порядке, то специально указывать порядок не требуется;
- `DESC` — по убыванию (*descending*).

Если в выражении `ORDER BY` указаны несколько столбцов сортировки, то сначала записи упорядочиваются по значениям первого столбца, затем для каждого значения первого столбца записи упорядочиваются по значениям второго столбца и т. д. Столбцы в списке разделяются, как обычно, запятыми. Таким образом, создается иерархическая система сортировки записей результатной таблицы.

В следующем примере данные исходной таблицы `Клиенты` сортируются по регионам и по именам клиентов (рис. 3.7). При этом сортировка по именам клиентов производится по убыванию, т. е. в порядке, противоположном алфавитному.

```
SELECT * FROM Клиенты
ORDER BY Регион, Имя DESC;
```

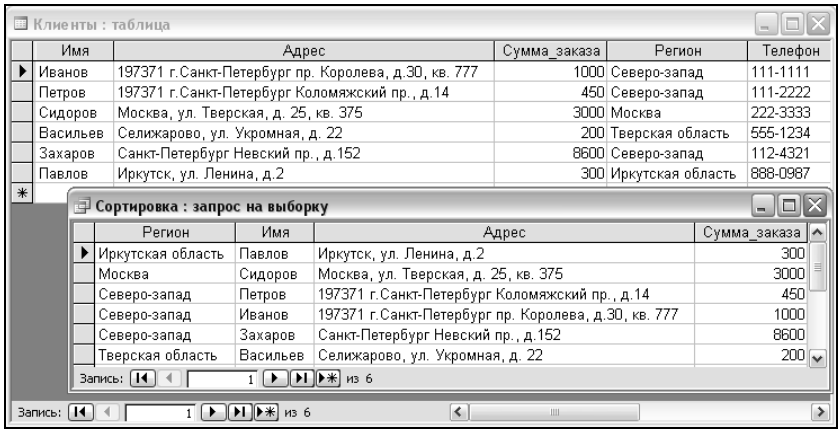


Рис. 3.7. Результат сортировки по регионам в алфавитном порядке и по именам клиентов в обратном порядке

3.2.5. Логические операторы

Логические выражения в операторах `WHERE` и `HAVING` могут быть сложными, т. е. состоять из двух и более простых выражений, соединенных между собой логическими операторами (союзами) `AND` и/или `OR`. Оператор `AND` выполняет роль логического союза И, а оператор `OR` — союза ИЛИ. Так, если `x` и `y` — два логических выражения, то составное выражение `x AND y` принимает значение `true` (ИСТИНА) только тогда, когда `x` и `y` одновременно истинны; в противном случае выражение `x AND y` принимает значение `false` (ЛОЖЬ). Выражение `x OR y` истинно, если хотя бы одно из выражений, `x` или `y`, истинно; если `x` и `y` одновременно ложны, то составное выражение `x OR y` ложно.

Логический оператор `NOT` применяется к одному выражению (возможно и к сложному), расположенному справа от него. Этот оператор меняет значение выражения на противоположное. Так, если выражение `x` имеет значение `true`, то выражение `NOT x` имеет значение `false`, и, наоборот, если `x` ложно, то `NOT x` истинно.

Предположим, из таблицы `Клиенты` (см. рис. 3.1) требуется выбрать записи о клиентах из Москвы и Северо-Запада. Соответствующий запрос имеет вид:

```
SELECT Регион, Имя, Сумма_заказа FROM Клиенты
WHERE Регион='Москва' OR Регион='Северо-Запад';
```

Обратите внимание, что здесь используется логический оператор `OR` (ИЛИ), а не `AND` (И), поскольку нам нужны клиенты, проживающие или в Москве, или на Северо-Западе. Если бы вместо оператора `OR` мы применили `AND`, то получили бы пустую таблицу, т. к. в исходной таблице нет ни одной записи, в которой один и тот же столбец имел бы различные значения.

Внимание

Будьте внимательны при формулировке запроса на естественном языке и при его переводе на SQL.

Следующее SQL-выражение эквивалентно рассмотренному ранее. Оно основано на применении оператора `IN` *(см. разд. 3.2.1)*:

```
SELECT Регион, Имя, Сумма_заказа FROM Клиенты
WHERE Регион IN ('Москва', 'Северо-Запад');
```

Если требуется получить данные о всех клиентах, которые не проживают ни в Москве, ни на Северо-Западе, то можно использовать такое SQL-выражение:

```
SELECT Регион, Имя, Сумма_заказа FROM Клиенты
WHERE NOT (Регион='Москва' OR Регион='Северо-Запад');
```

Это выражение эквивалентно следующим двум:

```
SELECT Регион, Имя, Сумма_заказа FROM Клиенты
WHERE Регион <> 'Москва' AND Регион <> 'Северо-Запад';
```

и:

```
SELECT Регион, Имя, Сумма_заказа FROM Клиенты
WHERE Регион NOT IN ('Москва', 'Северо-Запад');
```

3.3. Задачи

Выберите в качестве исходной таблицу `Клиенты`, показанную на рис. 3.1. Вы можете не копировать ее содержимое в точности,

а создать похожую таблицу самостоятельно. Важно, чтобы в таблице имелись символьные (текстовые) и числовые столбцы. Хорошо, если некоторые столбцы имели бы одинаковые значения, например, столбец `Регион`. В предлагаемых далее задачах требуется сформировать SQL-выражения, обеспечивающие некоторую выборку записей.

При формировании условий поиска (выборки) записей старайтесь абстрагироваться от конкретных данных в имеющейся таблице, поскольку пользователь может добавлять новые записи и модифицировать уже имеющиеся, делая это так, как ему хочется и как позволяют ограничения для данной таблицы. Например, столбец `Адрес` имеет символьный тип. Это означает, что он содержит произвольные символьные данные, структура которых поддерживается пользователем на семантическом уровне. В полях данного столбца может присутствовать или нет почтовый индекс, перед названием улицы может быть указано "ул." или "улица", или может быть ничего не указано, номер телефона может иметь различную значность, содержать или не содержать код региона, дефисы и т. п. При формулировании запроса в подобных ситуациях следует учесть как можно больше семантических нюансов.

Задача 3.1

Выберите записи, сгруппированные по регионам и исключающие Северо-Западный регион. Попробуйте сделать это по крайней мере двумя способами (с использованием `WHERE` и `HAVING`).

Задача 3.2

Выберите записи о клиентах, проживающих в городах, название которых оканчивается на "бург", а сумма заказа превышает 2000.

Задача 3.3

Выберите записи, в которых номера телефонов пятизначные. При этом следует предусмотреть, что номера телефонов могут содержать, а могут и не содержать дефисы, а сам номер может содержать или нет код региона, заключенный или не заключенный в круглые скобки.

Глава 4

Вычисления

В выражениях SQL-запросов нередко требуется выполнить предварительную обработку данных. С этой целью используются специальные функции и выражения.

4.1. Итоговые функции

Довольно часто требуется узнать, сколько записей соответствует тому или иному запросу, какова сумма значений некоторого числового столбца, его максимальное, минимальное и среднее значения. Для этого служат так называемые *итоговые* (статистические, агрегатные) функции. Итоговые функции обрабатывают наборы записей, заданные, например, выражением `WHERE`. Если их включить в список столбцов, следующий за оператором `SELECT`, то результатная таблица будет содержать не только столбцы таблицы базы данных, но и значения, вычисленные с помощью этих функций. Далее приведен список итоговых функций.

- `COUNT(`*`параметр`*`)` — возвращает количество записей, указанных в параметре. Если требуется получить количество всех записей, то в качестве параметра следует указать символ звездочки (`*`). Если в качестве параметра указать имя столбца, то функция вернет количество записей, в которых этот столбец имеет значения, отличные от `NULL`. Чтобы узнать, сколько

различных значений содержит столбец, перед его именем следует указать ключевое слово DISTINCT.

Например:

```
SELECT COUNT(*) FROM Клиенты;

SELECT COUNT(Сумма_заказа) FROM Клиенты;

SELECT COUNT(DISTINCT Сумма_заказа) FROM Клиенты;
```

Попытка выполнить следующий запрос приведет к сообщению об ошибке:

```
SELECT Регион, COUNT(*) FROM Клиенты;
```

- SUM(*параметр*) — возвращает сумму значений указанного в параметре столбца. Параметр может представлять собой и выражение, содержащее имя столбца.

  Например:

  ```
  SELECT SUM(Сумма_заказа) FROM Клиенты;
  ```

  Данное SQL-выражение возвращает таблицу, состоящую из одного столбца и одной записи и содержащую сумму всех определенных значений столбца Сумма_заказа из таблицы Клиенты.

  Допустим, что в исходной таблице значения столбца Сумма_заказа выражены в рублях, а нам требуется вычислить общую сумму в долларах. Если текущий обменный курс равен, например, 27,8, то получить требуемый результат можно с помощью выражения:

  ```
  SELECT SUM(Сумма_заказа*27.8) FROM Клиенты;
  ```

- AVG(*параметр*) — возвращает среднее арифметическое всех значений указанного в параметре столбца. Параметр может представлять собой выражение, содержащее имя столбца.

  Например:

  ```
  SELECT AVG(Сумма_заказа) FROM Клиенты;

  SELECT AVG(Сумма_заказа*27.8) FROM Клиенты
  WHERE Регион <> 'Северо_Запад';
  ```

- `MAX(`*параметр*`)` — возвращает максимальное значение в столбце, указанном в параметре. Параметр может также представлять собой выражение, содержащее имя столбца.

  Например:

```
SELECT MAX(Сумма_заказа) FROM Клиенты;

SELECT MAX(Сумма_заказа*27.8) FROM Клиенты WHERE
Регион <> 'Северо_Запад';
```

- `MIN(`*параметр*`)` — возвращает минимальное значение в столбце, указанном в параметре. Параметр может представлять собой выражение, содержащее имя столбца.

  Например:

```
SELECT MIN(Сумма_заказа) FROM Клиенты;

SELECT MIN(Сумма_заказа*27.8) FROM Клиенты WHERE
Регион <> 'Северо_Запад';
```

На практике нередко требуется получить итоговую таблицу, содержащую суммарные, усредненные, максимальные и минимальные значения числовых столбцов. Для этого следует использовать группировку (`GROUP BY`) и итоговые функции. В *разд. 3.2.2* был рассмотрен пример простой итоговой таблицы, полученной на основе таблицы `Клиенты` (см. рис. 3.1):

```
SELECT Регион, SUM(Сумма_заказа) FROM Клиенты
GROUP BY Регион;
```

Результатная таблица для данного запроса содержит имена регионов и итоговые (общие) суммы заказов всех клиентов из соответствующих регионов (рис. 3.5).

Теперь рассмотрим запрос на получение всех итоговых данных по регионам:

```
SELECT Регион, SUM(Сумма_заказа), AVG(Сумма_заказа),
MAX(Сумма_заказа), MIN(Сумма_заказа)

FROM Клиенты

GROUP BY Регион;
```

Исходная и результатная таблицы показаны на рис. 4.1. В данном примере только Северо-Западный регион представлен в исходной

таблице более чем одной записью. Поэтому в результатной таблице для него различные итоговые функции дают различные значения.

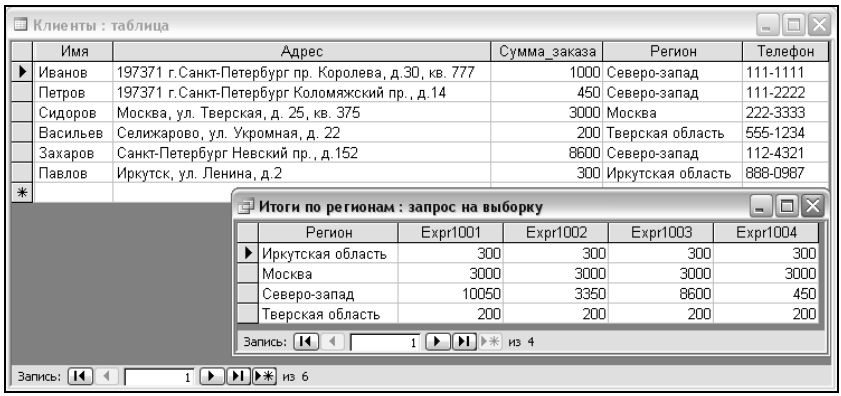


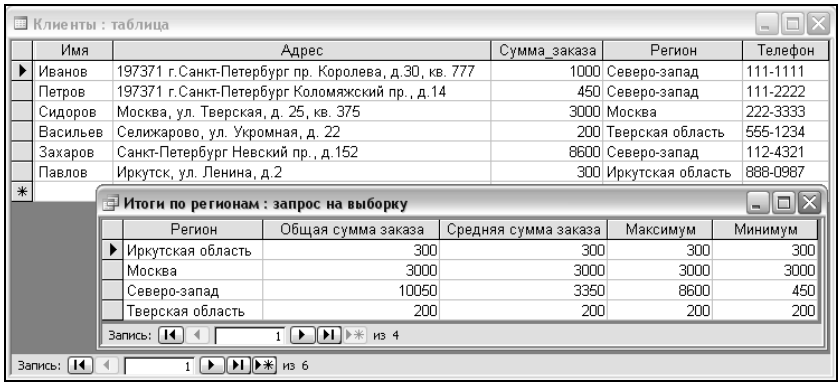
Рис. 4.1. Итоговая таблица сумм заказов по регионам

При использовании итоговых функций в списке столбцов в операторе `SELECT` заголовки соответствующих им столбцов в результатной таблице имеют вид `Expr1001`, `Expr1002` и т. д. (или что-нибудь аналогичное, в зависимости от реализации SQL). Однако заголовки для значений итоговых функций и других столбцов вы можете задавать по своему усмотрению. Для этого достаточно после столбца в операторе `SELECT` указать выражение вида:

```
AS заголовок_столбца
```

Ключевое слово `AS` (как) означает, что в результатной таблице соответствующий столбец должен иметь заголовок, указанный после `AS`. Назначаемый заголовок еще называют *псевдонимом*. В следующем примере (рис. 4.2) задаются псевдонимы для всех вычисляемых столбцов:

```
SELECT Регион,
SUM(Сумма_заказа) AS [Общая сумма заказа],
AVG(Сумма_заказа) AS [Средняя сумма заказа],
MAX(Сумма_заказа) AS Максимум,
MIN(Сумма_заказа) AS Минимум
FROM Клиенты
GROUP BY Регион;
```



Клиенты : таблица

Имя	Адрес	Сумма_заказа	Регион	Телефон
Иванов	197371 г.Санкт-Петербург пр. Королева, д.30, кв. 777	1000	Северо-запад	111-1111
Петров	197371 г.Санкт-Петербург Коломяжский пр., д.14	450	Северо-запад	111-2222
Сидоров	Москва, ул. Тверская, д. 25, кв. 375	3000	Москва	222-3333
Васильев	Селижарово, ул. Укромная, д. 22	200	Тверская область	555-1234
Захаров	Санкт-Петербург Невский пр., д.152	8600	Северо-запад	112-4321
Павлов	Иркутск, ул. Ленина, д.2	300	Иркутская область	888-0987

Итоги по регионам : запрос на выборку

Регион	Общая сумма заказа	Средняя сумма заказа	Максимум	Минимум
Иркутская область	300	300	300	300
Москва	3000	3000	3000	3000
Северо-запад	10050	3350	8600	450
Тверская область	200	200	200	200

Запись: 1 из 4

Запись: 1 из 6

Рис. 4.2. Итоговая таблица сумм заказов по регионам с применением псевдонимов столбцов

Псевдонимы, состоящие из нескольких слов, разделенных пробелами, заключаются в квадратные скобки.

Итоговые функции можно использовать в выражениях `SELECT` и `HAVING`, но их нельзя применять в выражении `WHERE`. Оператор `HAVING` аналогичен оператору `WHERE`, но в отличие от `WHERE` он отбирает записи в группах.

Допустим, требуется определить, в каких регионах более одного клиента. С этой целью можно воспользоваться таким запросом:

```
SELECT Регион, Count(*)
FROM Клиенты
GROUP BY Регион HAVING COUNT(*) > 1;
```

4.2. Функции обработки значений

При работе с данными часто приходится их обрабатывать (преобразовывать к нужному виду): выделить в строке некоторую подстроку, удалить ведущие и заключительные пробелы, округлить число, вычислить квадратный корень, определить текущее время и т. п.

В SQL имеются следующие три типа функций:

- строковые функции;
- числовые функции;
- функции даты-времени.

4.2.1. Строковые функции

Строковые функции принимают в качестве параметра строку и возвращают после ее обработки строку или NULL.

- SUBSTRING(*строка* FROM *начало* [FOR *длина*]) — возвращает подстроку, получающуюся из строки, которая указана в качестве параметра *строка*. Подстрока начинается с символа, порядковый номер которого указан в параметре *начало*, и имеет длину, указанную в параметре *длина*. Нумерация символов строки ведется слева направо, начиная с 1. Квадратные скобки здесь указывают лишь на то, что заключенное в них выражение не является обязательным. Если выражение FOR *длина* не используется, то возвращается подстрока от *начало* и до конца исходной строки. Значения параметров *начало* и *длина* должны выбираться так, чтобы искомая подстрока действительно находилась внутри исходной строки. В противном случае функция SUBSTRING вернет NULL.

  Например:

  SUBSTRING('Дорогая Маша!' FROM 9 FOR 4) — возвращает 'Маша';

  SUBSTRING('Дорогая Маша!' FROM 9) — возвращает 'Маша!';

  SUBSTRING('Дорогая Маша!' FROM 15) — возвращает NULL.

  Использовать эту функцию в SQL-выражении можно, например, так:

```
SELECT * FROM Клиенты
WHERE SUBSTRING(Регион FROM 1 FOR 5) = 'Север';
```

- UPPER(*строка*) — переводит все символы указанной в параметре строки в верхний регистр.
- LOWER(*строка*) — переводит все символы указанной в параметре строки в нижний регистр.
- TRIM(LEADING | TRAILING | BOTH ['*символ*'] FROM *строка*) — удаляет ведущие (LEADING), заключительные (TRAILING) или те и другие (BOTH) символы из строки. По умолчанию удаляемым символом является пробел (' '), поэтому его можно не указы-

вать. Чаще всего эта функция используется именно для удаления пробелов.

Например:

`TRIM(LEADING ' ' FROM '  город Санкт-Петербург  ')` — возвращает `'город Санкт-Петербург  '`;

`TRIM(TRALING ' ' FROM '  город Санкт-Петербург  ')` — возвращает `'  город Санкт-Петербург'`;

`TRIM(BOTH ' ' FROM '  город Санкт-Петербург  ')` — возвращает `'город Санкт-Петербург'`;

`TRIM(BOTH FROM '  город Санкт-Петербург  ')` — возвращает `'город Санкт-Петербург'`;

`TRIM(BOTH 'г' FROM 'город Санкт-Петербург')` — возвращает `'ород Санкт-Петербур'`.

Среди этих функций наиболее часто используемые — это `SUBSTRING()` и `TRIM()`.

4.2.2. Числовые функции

Числовые функции в качестве параметра могут принимать данные не только числового типа, но возвращают всегда число или `NULL` (неопределенное значение).

- `POSITION(`*`целеваяСтрока`* `IN` *`строка`*`)` — ищет вхождение целевой строки в указанную строку. В случае успешного поиска возвращает номер положения ее первого символа, иначе — 0. Если целевая строка имеет нулевую длину (например, пустая строка `''`), то функция возвращает 1. Если хотя бы один из параметров имеет значение `NULL`, то возвращается `NULL`. Нумерация символов строки ведется слева направо, начиная с 1.

  Например:

  `POSITION('е' IN 'Привет всем')` — возвращает 5;

  `POSITION('всем' IN 'Привет всем')` — возвращает 8;

  `POSITION('' IN 'Привет всем')` — возвращает 1;

  `POSITION('Привет!' IN 'Привет всем')` — возвращает 0.

В таблице Клиенты (см. рис. 3.1) столбец Адрес содержит, кроме названия города, почтовый индекс, название улицы и другие данные. Возможно, вам потребуется выбрать записи о клиентах, проживающих в определенном городе. Так, если требуется выбрать записи, относящиеся к клиентам, проживающим в Санкт-Петербурге, то можно воспользоваться следующим выражением SQL-запроса:

```
SELECT * FROM Клиенты
WHERE POSITION('Санкт-Петербург' IN Адрес) > 0;
```

Заметим, что этот простой запрос на выборку данных можно сформулировать иначе:

```
SELECT * FROM Клиенты
WHERE Адрес LIKE '%Петербург%';
```

- EXTRACT(*параметр*) — извлекает элемент из значения типа дата-время или из интервала.

  Например:

  EXTRACT(MONTH FROM DATE '2005-10-25') — возвращает 10.

- CHARACTER_LENGTH(*строка*) — возвращает количество символов в строке.

  Например:

  CHARACTER_LENGTH('Привет всем') — возвращает 11.

- OCTET_LENGTH(*строка*) — возвращает количество октетов (байтов) в строке. Каждый символ латиницы или кириллицы представляется одним байтом, а символ китайского алфавита — двумя байтами.

- CARDINALITY(*параметр*) — принимает в качестве параметра коллекцию элементов и возвращает количество элементов в коллекции (кардинальное число). Коллекция может быть, например, массивом или мультимножеством, содержащим элементы различных типов.

- ABS(*число*) — возвращает абсолютное значение числа.

  Например:

  ABS(-123) — возвращает 123;

  ABS(2 - 5) — возвращает 3.

- `MOD(`*число1*`, `*число2*`)` — возвращает остаток от целочисленного деления первого числа на второе.

  Например:

  `MOD(5, 3)` — возвращает 2;

  `MOD(2, 3)` — возвращает 0.

- `LN(`*число*`)` — возвращает натуральный логарифм числа.

- `EXP(`*число*`)` — возвращает $e^{число}$ (основание натурального логарифма в степени *число*).

- `POWER(`*число1*`, `*число2*`)` — возвращает $число1^{число2}$ (*число1* в степени *число2*).

- `SQRT(`*число*`)` — возвращает квадратный корень из числа.

- `FLOOR(`*число*`)` — возвращает наибольшее целое число, не превышающее заданное параметром (округление в меньшую сторону).

  Например:

  `FLOOR(5.123)` — возвращает `5.0`.

- `CEIL(`*число*`)` или `CEILING(`*число*`)` — возвращает наименьшее целое число, которое не меньше заданного параметром (округление в большую сторону).

  Например:

  `CEIL(5.123)` — возвращает `6.0`.

- `WIDTH_BUCKET(`*число1*`, `*число2*`, `*число3*`, `*число4*`)` — возвращает целое число в диапазоне между 0 и *число4* + 1. Параметры *число2* и *число3* задают числовой отрезок, разделенный на равновеликие интервалы, количество которых задается параметром *число4*. Функция определяет номер интервала, в который попадает значение *число1*. Если *число1* находится за пределами заданного диапазона, то функция возвращает 0 или *число4* + 1.

  Например:

  `WIDTH_BUCKET(3.14, 0, 9, 5)` — возвращает `2`.

4.2.3. Функции даты-времени

В языке SQL имеются три функции, которые возвращают текущие дату и время.

- `CURRENT_DATE` — возвращает текущую дату (тип `DATE`). Например, `2005-06-18`.
- `CURRENT_TIME(`*число*`)` — возвращает текущее время (тип `TIME`). Целочисленный параметр указывает точность представления секунд. Например, при значении `2` секунды будут представлены с точностью до сотых (две цифры в дробной части): `12:39:45.27`.
- `CURRENT_TIMESTAMP(`*число*`)` — возвращает дату и время (тип `TIMESTAMP`). Например, `2005-06-18 12:39:45.27`. Целочисленный параметр указывает точность представления секунд.

Обратите внимание, что дата и время, возвращаемые этими функциями, имеют не символьный тип. Если требуется представить их в виде символьных строк, то для этого следует использовать функцию преобразования типа `CAST()`.

Функции даты-времени обычно применяются в запросах на вставку, обновление и удаление данных. Например, при записи сведений о продажах в специально предусмотренный для этого столбец вносятся текущие дата и время. После подведения итогов за месяц или квартал, данные о продажах за отчетный период можно удалить.

Данные типа даты-времени подробно были рассмотрены в *разд. 2.2.4*, *2.2.5*.

4.3. Вычисляемые выражения

Вычисляемые выражения строятся из констант (числовых, строковых, логических), функций, имен полей и данных других типов путем соединения их арифметическими, строковыми, логическими и другими операторами. В свою очередь, выражения могут быть объединены посредством операторов в более сложные (составные) выражения. Для управления порядком вычисления выражений используются круглые скобки.

Логические операторы `AND`, `OR` и `NOT` были рассмотрены в *разд. 3.2.5*, а функции — в *разд. 4.1*, *4.2*.

Арифметические операторы приведены далее:

- `+` — сложение;
- `-` — вычитание;
- `*` — умножение;
- `/` — деление.

Строковый оператор только один — оператор конкатенации или склейки строк (`||`). В некоторых реализациях SQL (например, в Microsoft Access) вместо (`||`) используется символ (+). Оператор конкатенации приписывает вторую строку к концу первой. Например, выражение:

```
'Саша' || 'любит' || '  Машу'
```

вернет в качестве результата строку `'Сашалюбит  Машу'`.

При составлении выражений необходимо следить, чтобы операнды операторов имели допустимые типы. Например, выражение:

```
123 + 'Саша'
```

недопустимо, поскольку арифметический оператор сложения применяется к строковому операнду.

Вычисляемые выражения могут находиться после оператора `SELECT`, а также в выражениях условий операторов `WHERE` и `HAVING`.

Рассмотрим несколько примеров.

Пусть таблица `Продажи` содержит столбцы `Тип_товара`, `Количество` и `Цена`, а нам требуется знать выручку для каждого типа товара. Для этого достаточно в список столбцов после оператора `SELECT` включить выражение `Количество*Цена`:

```
SELECT Тип_товара, Количество, Цена, Количество*Цена AS
Итого FROM Продажи;
```

Здесь используется ключевое слово `AS` (как) для задания псевдонима столбца с вычисляемыми данными.

На рис. 4.3 показаны исходная таблица `Продажи` и результатная таблица запроса.

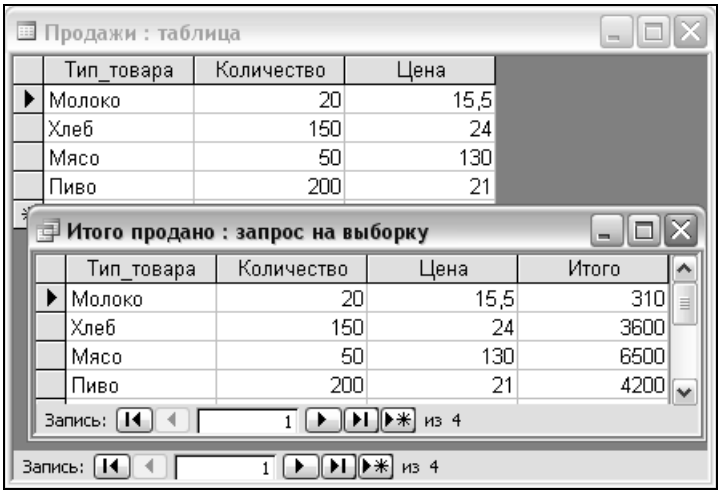


Рис. 4.3. Результат запроса с вычислением выручки по каждому типу товара

Если требуется узнать общую выручку от продажи всех товаров, то достаточно применить следующий запрос:

```
SELECT SUM(Количество*Цена) FROM Продажи;
```

Следующий запрос содержит вычисляемые выражения и в списке столбцов, и в условии оператора `WHERE`. Он выбирает из таблицы `Продажи` те товары, выручка от продажи которых больше 1000:

```
SELECT Тип_товара, Количество*Цена AS Итого
FROM Продажи
WHERE Количество*Цена > 1000;
```

Предположим, что требуется получить таблицу, в которой два столбца:

- `Товар`, содержащий тип товара и цену;
- `Итого`, содержащий выручку.

Поскольку предполагается, что в исходной таблице `Продажи` столбец `Тип_товара` является символьным (тип `CHAR`), а столбец `Цена` — числовой, то при объединении (склейке) данных из этих столбцов необходимо выполнить приведение числового типа

к символьному с помощью функции `CAST()`. Запрос, выполняющий это задание, выглядит так (рис. 4.4):

```
SELECT Тип_товара || ' (Цена: ' || CAST(Цена AS CHAR(5)) ||
')' AS Товар, Количество*Цена AS Итого
FROM Продажи;
```

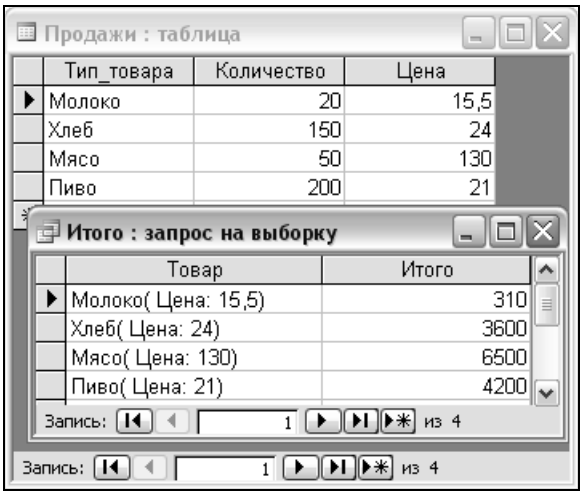


Рис. 4.4. Результат запроса с объединением разнотипных данных в одном столбце

Примечание

В Microsoft Access аналогичный запрос будет иметь следующий вид:

```
SELECT Тип_товара + ' (Цена: ' + CStr(Цена) + ')' AS Товар,
Количество*Цена AS Итого
FROM Продажи;
```

4.4. Условные выражения с оператором *CASE*

В обычных языках программирования имеются операторы условного перехода, которые позволяют управлять вычислительным процессом в зависимости от того, выполняется или нет некоторое

условие. В языке SQL таким оператором является CASE (случай, обстоятельство, экземпляр). В SQL:2003 этот оператор возвращает значение и, следовательно, может использоваться в выражениях. Он имеет две основные формы, которые мы рассмотрим в данном разделе.

4.4.1. Оператор *CASE* со значениями

Оператор CASE со значениями имеет следующий синтаксис:

```
CASE проверяемое_значение
    WHEN значение1 THEN результат1
    WHEN значение2 THEN результат2
    ...
    WHEN значениеN THEN результатN
    ELSE результатX
END
```

В случае, когда *проверяемое_значение* равно *значение1*, оператор CASE возвращает значение *результат1*, указанное после ключевого слова THEN (то). В противном случае *проверяемое_значение* сравнивается с *значение2*, и если они равны, то возвращается значение *результат2*. В противном случае *проверяемое_значение* сравнивается со следующим значением, указанным после ключевого слова WHEN (когда) и т. д. Если *проверяемое_значение* не равно ни одному из таких значений, то возвращается значение *результатX*, указанное после ключевого слова ELSE (иначе).

Ключевое слово ELSE не является обязательным. Если оно отсутствует и ни одно из значений, подлежащих сравнению, не равно проверяемому значению, то оператор CASE возвращает NULL.

Допустим, на основе таблицы Клиенты (см. рис. 3.1) требуется получить таблицу, в которой названия регионов заменены их кодовыми номерами. Если в исходной таблице различных регионов не слишком много, то для решения данной задачи удобно воспользоваться запросом с оператором CASE:

```
SELECT Имя, Адрес,
CASE Регион
```

```
        WHEN 'Москва' THEN '77'
        WHEN 'Тверская область' THEN '69'
        ...
        ELSE Регион
    END
    AS Код региона
    FROM Клиенты;
```

4.4.2. Оператор *CASE* с условиями поиска

Вторая форма оператора `CASE` предполагает его использование при поиске в таблице тех записей, которые удовлетворяют определенному условию:

```
CASE
    WHEN условие1 THEN результат1
    WHEN условие2 THEN результат2
    ...
    WHEN условиеN THEN результатN
    ELSE результатX
END
```

Оператор `CASE` проверяет, истинно ли *условие1* для первой записи в наборе, определенном оператором `WHERE`, или во всей таблице, если `WHERE` отсутствует. Если да, то `CASE` возвращает значение *результат1*. В противном случае для данной записи проверяется *условие2*. Если оно истинно, то возвращается значение *результат2* и т. д. Если ни одно из условий не выполняется, то возвращается значение *результатX*, указанное после ключевого слова `ELSE`.

Ключевое слово `ELSE` не является обязательным. Если оно отсутствует и ни одно из условий не выполняется, оператор `CASE` возвращает `NULL`. После того как оператор, содержащий `CASE`, выполнится для первой записи, происходит переход к следующей записи. Так продолжается до тех пор, пока не будет обработан весь набор записей.

Предположим, в таблице `Книги` (`Название`, `Цена`) столбец `Цена` имеет значение `NULL`, если соответствующей книги нет в наличии.

Следующий запрос возвращает таблицу, в которой вместо NULL отображается текст "Нет в наличии":

```
SELECT Название,
CASE
    WHEN Цена IS NULL THEN 'Нет в наличии'
    ELSE CAST(Цена AS CHAR(8))
END
AS Цена
FROM Книги;
```

Все значения одного и того же столбца должны иметь одинаковые типы. Поэтому в данном запросе используется функция преобразования типов CAST для приведения числовых значений столбца Цена к символьному типу.

Обратите внимание, что вместо первой формы оператора CASE всегда можно использовать вторую:

```
CASE
    WHEN проверяемое_значение = значение1 THEN результат1
    WHEN проверяемое_значение = значение2 THEN результат2
    ...
    WHEN проверяемое_значение = значениеN THEN результатN
    ELSE результатX
END
```

4.4.3. Функции *NULLIF* и *COALESCE*

В ряде случаев, особенно в запросах на обновление данных (оператор UPDATE), удобно использовать вместо громоздкого оператора CASE более компактные функции NULLIF() (NULL, если) и COALESCE() (объединять).

Функция NULLIF(*значение1*, *значение2*) возвращает NULL, если значение первого параметра соответствует значению второго параметра, в случае несоответствия возвращается значение первого параметра без изменений. То есть если равенство *значение1* = *значение2* выполняется, то функция возвращает NULL, иначе — значение *значение1*.

Данная функция эквивалентна оператору `CASE` в следующих двух формах:

- ```
  CASE значение1
      WHEN значение2 THEN NULL
      ELSE значение1
  END
  ```
- ```
  CASE
      WHEN значение1 = значение2 THEN NULL
      ELSE значение1
  END
  ```

Пример использования функции `NULLIF()` в запросе будет приведен в *разд. 6.3*.

Функция `COALESCE(значение1, значение2, ... , значениеN)` принимает список значений, которые могут быть как определенными, так и неопределенными (`NULL`). Функция возвращает первое определенное значение из списка или `NULL`, если все значения не определены.

Данная функция эквивалентна следующему оператору `CASE`:

```
CASE
    WHEN значение1 IS NOT NULL THEN значение1
    WHEN значение2 IS NOT NULL THEN значение2
    ...
    WHEN значениеN IS NOT NULL THEN значениеN
    ELSE NULL
END
```

Предположим, что в таблице `Книги` (`Название`, `Цена`) столбец `Цена` имеет значение `NULL`, если соответствующей книги нет в наличии. Следующий запрос возвращает таблицу, в которой вместо `NULL` отображается текст `"Нет в наличии"`:

```
SELECT Название, COALESCE(CAST(Цена AS CHAR(8)),
'Нет в наличии') AS Цена
FROM Книги;
```

Глава 5

Сложные запросы

В обычной жизни сложные вопросы считаются таковыми, потому что на них трудно найти ответ. Например, вопрос "В чем смысл жизни?" считается сложным потому, что на него трудно однозначно ответить. Поэтому такими вопросами занимаются философы, поэты и обыватели. Простой вопрос предполагает, хотя бы в общих чертах, что именно может быть ответом на него или каким способом можно получить ответ. Например, вопрос "Сколько будет дважды два?" предполагает, что ответчик вспомнит, или вычислит, или обратится в другие инстанции за справками, но, в конце концов, вернет некий ответ, соответствующий вопросу.

В SQL сложные вопросы (запросы) являются комбинацией простых SQL-запросов. Каждый простой запрос в качестве ответа возвращает набор записей (таблицу), а комбинация простых запросов возвращает результат тех или иных операций над ответами на простые запросы. Чтобы понять запрос, лучше всего разобраться в том, что является ответом на него, или понять, каким образом получается ответ.

В SQL сложные запросы получаются из других запросов следующими способами:

- вложением SQL-выражения запроса в SQL-выражение другого запроса. Первый из них называют подзапросом, а второй — внешним или основным запросом;

- применением к SQL-запросам операторов объединения и соединения наборов записей, возвращаемых запросами. Эти операторы называют теоретико-множественными или реляционными.

5.1. Подзапросы

Подзапрос — это SQL-выражение, начинающееся с оператора `SELECT`, которое содержится в условии оператора `WHERE` или `HAVING` для другого запроса. Таким образом, подзапрос — это запрос на выборку данных, вложенный в другой запрос. Внешний запрос, содержащий подзапрос, если только он сам не является подзапросом, не обязательно должен начинаться с оператора `SELECT`. В свою очередь, подзапрос может содержать другой подзапрос и т. д. При этом сначала выполняется подзапрос, имеющий самый глубокий уровень вложения, затем содержащий его подзапрос и т. д. Часто, но не всегда, внешний запрос обращается к одной таблице, а подзапрос — к другой. На практике именно этот случай наиболее интересен.

5.1.1. Простые подзапросы

Простые подзапросы характеризуются тем, что они формально никак не связаны с содержащими их внешними запросами. Это обстоятельство позволяет сначала выполнить подзапрос, результат которого затем используется для выполнения внешнего запроса. Кроме простых подзапросов, существуют еще и связанные (коррелированные) подзапросы, которые будут рассмотрены в *разд. 5.1.2*.

Рассматривая простые подзапросы, следует выделить три частных случая:

- подзапросы, возвращающие единственное значение;
- подзапросы, возвращающие список значений из одного столбца таблицы;
- подзапросы, возвращающие набор записей.

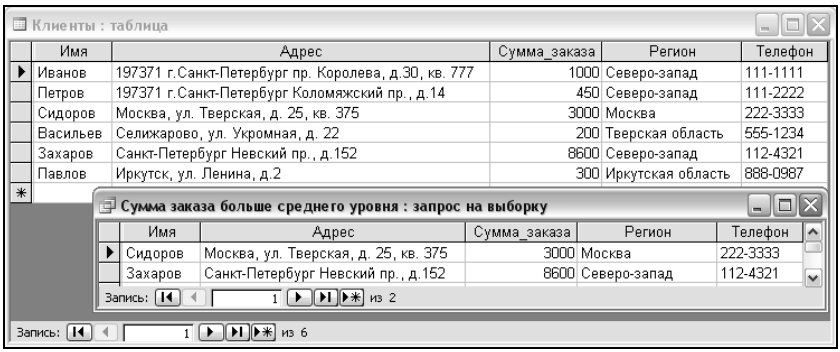
Рассмотрим эти частные случаи более подробно.

Работа с единственным значением

Допустим, из таблицы `Клиенты` требуется выбрать данные о тех клиентах, сумма заказов которых больше среднего значения. Это можно сделать с помощью следующего запроса (рис. 5.1):

```
SELECT * FROM Клиенты
WHERE
Сумма_заказа > (SELECT AVG(Сумма_заказа) FROM Клиенты);
```

В данном запросе сначала выполняется подзапрос (`SELECT AVG(Сумма_заказа) FROM Клиенты`). Он возвращает единственное значение (а не набор записей) — среднее значение столбца `Сумма_заказа`. Если сказать точнее, то данный подзапрос возвращает единственную запись, содержащую единственное поле. Далее выполняется внешний запрос, который выводит все столбцы таблицы `Клиенты` и записи, в которых значение столбца `Сумма_заказа` больше значения, полученного с помощью подзапроса. Таким образом, сначала выполняется подзапрос, а затем внешний запрос, использующий результат подзапроса.



Клиенты : таблица

Имя	Адрес	Сумма_заказа	Регион	Телефон
Иванов	197371 г.Санкт-Петербург пр. Королева, д.30, кв. 777	1000	Северо-запад	111-1111
Петров	197371 г.Санкт-Петербург Коломяжский пр., д.14	450	Северо-запад	111-2222
Сидоров	Москва, ул. Тверская, д. 25, кв. 375	3000	Москва	222-3333
Васильев	Селижарово, ул. Укромная, д. 22	200	Тверская область	555-1234
Захаров	Санкт-Петербург Невский пр., д.152	8600	Северо-запад	112-4321
Павлов	Иркутск, ул. Ленина, д.2	300	Иркутская область	888-0987

Запись: 1 из 6

Сумма заказа больше среднего уровня : запрос на выборку

Имя	Адрес	Сумма_заказа	Регион	Телефон
Сидоров	Москва, ул. Тверская, д. 25, кв. 375	3000	Москва	222-3333
Захаров	Санкт-Петербург Невский пр., д.152	8600	Северо-запад	112-4321

Запись: 1 из 2

Рис. 5.1. Результат запроса с подзапросом среднего значения

Попытка выполнить следующий простой запрос приведет к ошибке:

```
SELECT * FROM Клиенты
WHERE Сумма_заказа > AVG(Сумма_заказа);
```

Внимание

Выражение подзапроса обязательно должно быть заключено в круглые скобки.

Подзапрос, вообще говоря, может возвращать несколько записей. Чтобы в этом случае в условии внешнего оператора `WHERE` можно было использовать операторы сравнения, требующие единственного значения, используются кванторы, такие как `ALL` (все) и `SOME` (некоторый), описанные в *разд. 3.2.1*.

Допустим, в базе данных имеются две таблицы:

- `Сотрудники` (`Имя`, `Зарплата`), содержащая значения зарплаты для сотрудников некоторой фирмы;
- `Ср_зарплата_в_промышленности` (`Тип_промышленности`, `Ср_зарплата`), содержащая значения среднего уровня зарплаты для различных типов промышленности.

Требуется вывести список сотрудников, получающих зарплату, превышающую средний уровень для любого типа промышленности. С этой целью можно воспользоваться следующим запросом (рис. 5.2):

```
SELECT Сотрудники.Имя FROM Сотрудники
WHERE Сотрудники.Зарплата >
ALL (SELECT Ср_зарплата FROM Ср_зарплата_в_промышленности);
```

В данном примере используются полные имена столбцов, содержащие в качестве префиксов имена таблиц. Подзапрос (`SELECT Ср_зарплата FROM Ср_зарплата_в_промышленности`) возвращает список средних зарплат для различных типов промышленности. Выражение `> ALL` (больше всех) означает, что внешний запрос должен вернуть только те значения столбца `Зарплата` из таблицы `Сотрудники`, которые больше каждого значения, возвращенного вложенным подзапросом.

Результатный список сотрудников будет иным, если вместо квантора `ALL` применить `SOME` или `ANY`:

```
SELECT Сотрудники.Имя FROM Сотрудники
WHERE Сотрудники.Зарплата >
SOME (SELECT Ср_зарплата FROM Ср_зарплата_в_промышленности);
```

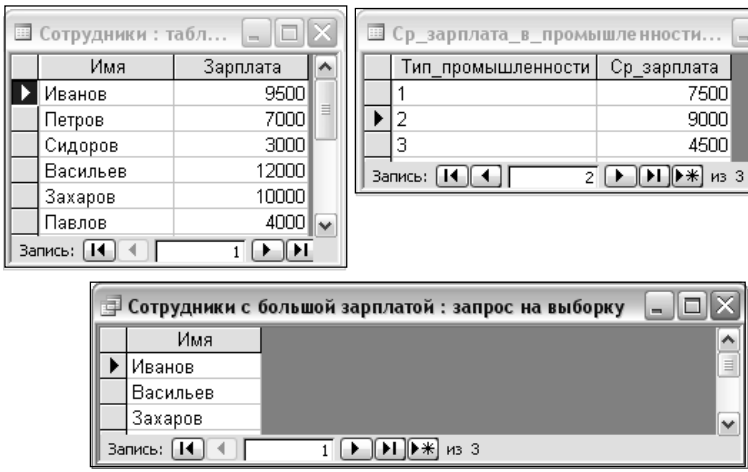


Рис. 5.2. Список сотрудников, зарплата которых выше средней для всех типов промышленности

Этот запрос вернет список сотрудников, у которых зарплата выше средней хотя бы для какого-нибудь одного типа промышленности. В данном примере к списку, показанному на рис. 5.2, добавится `'Петров'`, поскольку его зарплата выше средней для третьего типа промышленности.

Разумеется, вложенные подзапросы могут содержать условия, определяемые оператором `WHERE`.

Теперь рассмотрим запросы с кванторами `ALL` и `SOME` для более общего случая. Пусть имеются две таблицы: `T1`, содержащая как минимум столбец `A`, и `T2`, содержащая, по крайней мере, один столбец `B`. Тогда запрос с квантором `ALL` можно сформулировать следующим образом:

```
SELECT A FROM T1
WHERE A оператор_сравнения
    ALL (SELECT B FROM T2);
```

Здесь *`оператор_сравнения`* обозначает любой оператор сравнения из тех, что перечислены в *разд. 3.2.1*. Данный запрос должен вернуть список всех тех значений столбца `A`, для которых оператор сравнения истинен для всех значений столбца `B`.

Запрос с квантором SOME, очевидно, имеет аналогичную структуру. Он должен вернуть список всех тех значений столбца A, для которых оператор сравнения истинен хотя бы для какого-нибудь одного значения столбца B.

Запрос с квантором ALL станет, возможно, более понятным, если мы перепишем его в терминах математической логики:

```
∀a ∈ T1.A  ∀b ∈ T2.B (a оператор_сравнения b);
```

Данное выражение определяет подмножество *всех* значений столбца A таблицы T1, таких, что для *всех* значений столбца B таблицы T2 выполняется выражение сравнения значений из A и B, записанное в круглых скобках. Чтобы сформировать указанное подмножество значений столбца A, достаточно просто проверить для каждого значения A, выполняется ли выражение сравнения для всех значений столбца B. Если для некоторого значения из A это выражение не выполняется хотя бы для одного значения из B, то оно отбраковывается, т. е. не попадает в формируемое множество. В противном случае оно попадает в требуемое множество.

Условие с квантором SOME можно записать так:

```
∀a ∈ T1.A  ∃b ∈ T2.B (a оператор_сравнения b);
```

Данное выражение определяет подмножество *всех* значений столбца A таблицы T1, таких, что для каждого из них найдется хотя бы одно значение столбца B таблицы T2, при котором выполняется выражение сравнения значений из A и B, записанное в круглых скобках. Чтобы сформировать указанное подмножество значений столбца A, достаточно просто проверить для каждого значения A, выполняется ли выражение сравнения хотя бы для какого-нибудь одного значения из B. Если для некоторого значения из A это выражение не выполняется для всех значений из B, то оно отбраковывается, т. е. не попадает в формируемое множество. В противном случае оно попадает в требуемое множество.

Работа со списком значений из одного столбца

Рассмотрим применение подзапросов, возвращающих не единственное значение, а список значений из одного столбца. Предположим, что некая фирма собирает некоторые технические систе-

мы из компонентов. Информация о системах, компонентах и связях между ними хранится в следующих трех таблицах:

- `Товар` (`ID_системы`, `Название`, `Описание`, `Цена`);
- `Компонент` (`ID_компонента`, `Название`, `Описание`);
- `Состав` (`ID_системы`, `ID_компонента`).

Допустим, нас интересует список систем, в которых имеется некоторый компонент, скажем, микропроцессор. Для этой цели подойдет сложный запрос с предикатом `IN` (в):

```
SELECT ID_системы FROM Состав
WHERE ID_компонента IN
    ( SELECT ID_компонента FROM Компонент
        WHERE Название = 'Микропроцессор' );
```

Сначала выполняется подзапрос, возвращающий список идентификаторов компонентов, имеющих название `'Микропроцессор'`. Далее, внешний запрос сравнивает значение идентификатора компонента из каждой записи таблицы `Состав` с полученным списком. Если сравнение успешно (сравниваемое значение имеется в списке), то идентификатор системы из той же записи добавляется в результатную таблицу.

Теперь сформулируем запрос, возвращающий список систем, в которых нет заданного компонента, например, микропроцессора. В этом запросе будет два подзапроса с использованием ключевых слов `IN` и `NOT IN` (не в):

```
SELECT ID_системы FROM Состав
WHERE ID_системы NOT IN
    ( SELECT ID_системы FROM Состав
        WHERE ID_компонента IN
            ( SELECT ID_компонента FROM Компонент
                WHERE Название = 'Микропроцессор' ));
```

Разумеется, это один из возможных вариантов запросов, возвращающих требуемые данные.

Работа с набором записей

В полнофункциональных базах данных подзапрос можно вставлять не только в операторы `WHERE` и `HAVING`, но и в оператор `FROM`.

Трудно однозначно сказать, зачем это надо делать, но, тем не менее, следующая конструкция работает:

```
SELECT T.столбец1, T.столбец2, ... , T.столбецN
FROM (SELECT ... ) T
WHERE ... ;
```

Здесь таблице, возвращаемой подзапросом в операторе `FROM`, присваивается псевдоним `T`, а внешний запрос выделяет столбцы этой таблицы и, возможно, записи в соответствии с некоторым условием, которое указано в операторе `WHERE`.

Пример для данной формы запроса приведен в конце *разд. 5.2.2*.

5.1.2. Связанные подзапросы

Связанные (коррелированные) подзапросы с практической точки зрения наиболее интересны, поскольку позволяют выразить более сложные вопросы относительно сведений, хранящихся в базе данных. Однако для их применения важно понимать, что именно будет делать СУБД для предоставления ответа. Для начала достаточно уяснить, что при выполнении запросов, содержащих связанные подзапросы, нет такого четкого разделения во времени выполнения между подзапросом и запросом, как в случае простых подзапросов. Напомню, что в случае простых подзапросов сначала выполняется подзапрос, а затем содержащий его запрос. В случае связанных подзапросов порядок выполнения запроса в целом иной, и его желательно понимать, чтобы избежать недоразумений. Основной признак связанного подзапроса заключается в том, что он не может быть выполнен самостоятельно, вне всякой связи с основным запросом. Формально этот признак обнаруживается в выражении сложного запроса следующим образом: подзапрос ссылается на таблицу, которая упоминается в основном запросе. Следовательно, подзапрос должен быть выполнен в некоем контексте с текущим состоянием выполнения основного запроса. Рассмотрим это на следующих примерах.

Прежде чем обратиться к конкретным примерам, рассмотрим сначала некоторый абстрактный и, в то же время, типичный запрос, содержащий связанный подзапрос:

```
SELECT A FROM T1
```

```
WHERE B =
    ( SELECT B FROM T2 WHERE C = T1.C );
```

Данный запрос на выборку данных (поскольку он начинается с оператора `SELECT`) содержит подзапрос, сформулированный в выражении, размещенном в основном запросе после ключевого слова `WHERE`.

Очевидно, данный запрос в целом использует две таблицы: `T1` и `T2`, в которых есть столбцы с одинаковыми именами `B` и `C` и одинаковыми типами. Подзапрос, расположенный в выражении после ключевого слова `WHERE` основного запроса (`SELECT B FROM T2 WHERE C = T1.C`), обращается к этим же таблицам. Поскольку одна из таблиц (`T1`) фигурирует как в подзапросе, так и во внешнем запросе, то подзапрос нельзя выполнить самостоятельно, вне связи с внешним запросом. Поэтому выполнение запроса в целом (т. е. внешнего запроса) происходит следующим образом:

1. Сначала выделяется первая запись из таблицы `T1`, указанной в операторе `FROM` внешнего запроса (вся запись таблицы `T1`, а не только значение столбца `A`). Эта запись называется текущей. Значения столбцов для этой записи доступны и могут быть использованы в подзапросе.
2. Затем выполняется подзапрос, который возвращает список значений столбца `B` таблицы `T2` в тех записях, в которых значение столбца `C` равно значению столбца `C` из таблицы `T1`.
3. Запрос, сформулированный в рассматриваемом примере, предполагает, что его подзапрос возвращает единственное значение (список с единственным элементом), поскольку в операторе `WHERE` используется оператор сравнения (=). Если это не так, то потребуется использование, например, предикатов. Как бы то ни было, считаем, что в этом примере подзапрос возвращает единственное значение. Теперь выполняется оператор `WHERE` основного запроса. Если значение столбца `B` в текущей (выделенной) записи таблицы `T1` равно значению, возвращенному подзапросом, то эта запись выделяется внешним запросом.
4. Оператор `SELECT` внешнего запроса выполняет проверку условия своего оператора `WHERE`. А именно он проверяет, равно ли

текущее значение столбца B таблицы T1 значению, возвращенному подзапросом. Если да, то значение столбца A текущей записи таблицы T1 помещается в результатную таблицу, в противном случае запись игнорируется. Затем происходит переход к следующей записи таблицы T1. Теперь для нее выполняется подзапрос. Аналогичным образом все описанное происходит для каждой записи таблицы T1.

Примечание

При формировании связанных запросов рекомендуется использовать полные имена столбцов, а также псевдонимы таблиц и столбцов для того, чтобы не запутаться. В приведенном примере имена таблиц и столбцов достаточно коротки, чтобы не использовать для них псевдонимы. Однако применение полных имен столбцов сделало бы SQL-выражение запроса более понятным:

```
SELECT T1.A FROM T1
WHERE T1.B =
    ( SELECT T2.B FROM T2 WHERE T2.C = T1.C );
```

Теперь обратимся к более или менее конкретным примерам.

Нередко торгующие фирмы делают своим клиентам скидки, размер которых зависит от суммы покупки (заказа). Предположим, размеры скидок (коэффициенты) заданы в виде таблицы Скидки, показанной на рис. 5.3. В этой таблице содержатся границы диапазонов суммы покупки и соответствующие коэффициенты скидки. Понятно, что чем больше сумма покупки, тем больше коэффициент скидки (размер скидки равен сумме покупки, умноженной на коэффициент скидки). Тогда, чтобы узнать, какова скидка для клиента, скажем, Захарова, сведения о котором находятся в другой таблице, например, Клиенты (см. рис. 3.1), можно воспользоваться следующим запросом:

```
SELECT Скидка FROM Скидки
WHERE Мин_сумма <=
    ( SELECT Сумма_заказа FROM Клиенты
        WHERE Имя = 'Захаров' )
    AND
    Макс_сумма >=
```

```
( SELECT Сумма_заказа FROM Клиенты
    WHERE Имя = 'Захаров' );
```

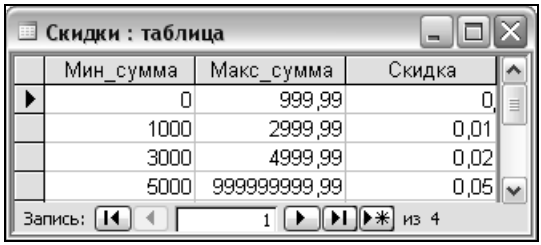


Рис. 5.3. Таблица скидок

Здесь подзапрос состоит из двух аналогичных простых запросов, связанных логическим союзом `AND` (и). Таким образом, подзапрос является составным. Запрос в целом выполняется следующим образом. Сначала в таблице `Скидки` выделяется первая запись. Далее выполняются два идентичных подзапроса, возвращающие значение столбца `Сумма_заказа` таблицы `Клиенты` при условии, что столбец `Имя` имеет значение `'Захаров'`. Предполагается, что подзапрос возвращает единственное значение. Если это не так, то в операторе `SELECT` вместо имени столбца следовало бы написать функцию `SUM(Сумма_заказа)`. Результаты подзапросов сравниваются с текущими границами диапазона скидок. Если оба сравнения успешны (истинны), то внешний запрос возвращает текущее значение коэффициента скидки. В противном случае в таблице `Скидки` происходит переход к следующей записи, и все повторяется, как описано ранее.

Как известно, запрос возвращает одну или несколько записей либо не возвращает ничего. Рассмотрим пример, в котором требуется проверка существования записей. Так, иногда требуется выборка записей из одной таблицы при условии, что в другой таблице существует хотя бы одна соответствующая запись.

Пусть в базе данных имеются:

- таблица `Продажи`, содержащая данные о продажах товаров некоторой фирмы, в том числе столбец `ID_клиента` (идентификатор клиента);

- таблица Контакты, содержащая данные о покупателях (ID_клиента, Имя, Адрес, Телефон), но не содержащая сведений об их покупках.

Требуется получить сведения о клиентах, сделавших хотя бы одну покупку. Далее приведен запрос, выполняющий данное задание:

```
SELECT Имя, Адрес, Телефон FROM Контакты
WHERE EXISTS
    ( SELECT DISTINCT ID_клиента FROM Продажи
        WHERE Продажи.ID_клиента = Контакты.ID_клиента );
```

Здесь предикат EXISTS (существует) принимает значение true, если подзапрос возвращает хотя бы одну запись, и тогда внешний запрос возвращает имя клиента и его данные для контакта. Поскольку в запросе требуется проверка существования записей, возвращаемых подзапросом, а не сами записи, то приведенное SQL-выражение можно несколько упростить:

```
SELECT Имя, Адрес, Телефон FROM Контакты
WHERE EXISTS
    ( SELECT DISTINCT 1 FROM Продажи
        WHERE Продажи.ID_клиента = Контакты.ID_клиента );
```

Здесь, чтобы выполнить подзапрос, содержимое результата которого для нас не важно, вместо имени столбца вставлена просто цифра 1. Разумеется, в данном случае можно было бы использовать и любое другое число. Это своего рода "фокус-покус".

Связанные подзапросы могут относиться к той же таблице, что и внешний запрос. В этих случаях для одной и той же таблицы используются различные псевдонимы, чтобы показать, из каких записей следует выбирать требуемые значения. Кроме того, подзапросы могут содержаться не только в операторе WHERE, но и в операторе HAVING, который обычно (но не всегда) используется вместе с оператором группировки GROUP BY *(см. разд. 3.2.2, 3.2.3)*.

В следующем примере из таблицы Продажи выбирается список тех клиентов, у которых максимальная сумма заказов как минимум в 1,5 раза больше среднего размера заказов остальных клиентов:

```
SELECT T1.ID_клиента FROM Продажи T1
GROUP BY T1.ID_клиента
```

```
HAVING MAX(T1.Сумма_заказа) > ALL
    ( SELECT 1.5*AVG(T2.Сумма_заказа) FROM Продажи T2
       WHERE T1.ID_клиента <> T2.ID_клиента );
```

Данный запрос работает следующим образом:

1. Записи таблицы `Продажи` группируются по значениям столбца `ID_клиента`.
2. Получившиеся группы обрабатываются оператором `HAVING`: для каждой группы вычисляется максимальное значение столбца `Сумма_заказа` (`MAX(T1.Сумма_заказа)`).
3. Подзапрос дважды проверяет среднее значение столбца `Сумма_заказа` для всех тех записей, в которых значение `ID_клиента` отличается от значения этого же столбца в текущей группе внешнего запроса (`WHERE T1.ID_клиента <> T2.ID_клиента`). Поэтому для одной и той же таблицы использовались два различных псевдонима.

5.2. Теоретико-множественные операции

Над наборами записей, содержащихся в таблицах базы данных и/или возвращаемых запросами, можно совершать теоретико-множественные операции, такие как декартово произведение, объединение, пересечение и вычитание. Что такое теоретико-множественные операции, достаточно подробно рассматривалось в *гл. 1*.

5.2.1. Декартово произведение наборов записей

Декартово произведение двух таблиц уже рассматривалось ранее. Напомню, что запрос:

```
SELECT списокСтолбцов FROM T1, T2, ... Tn;
```

возвращает набор записей, полученный в результате декартового произведения наборов записей из таблиц `T1`, `T2`, ... , `Tn`. Таблицы,

указанные в операторе FROM, могут быть как таблицами базы данных, так и виртуальными таблицами, возвращаемыми какими-нибудь запросами.

Иногда требуется получить декартово произведение таблицы самой на себя. В этом случае необходимо применить различные псевдонимы для этой таблицы, например:

```
SELECT списокСтолбцов FROM Mytab T1, Mytab T2;
```

Обратите внимание, что попытка выполнить запрос:

```
SELECT списокСтолбцов FROM Mytab, Mytab;
```

приведет к ошибке.

В списке столбцов следует использовать полные имена столбцов, используя псевдонимы таблиц, или символ (*), если требуется получить все столбцы.

Для декартова произведения в SQL также допустим синтаксис с ключевыми словами CROSS JOIN (перекрестное соединение):

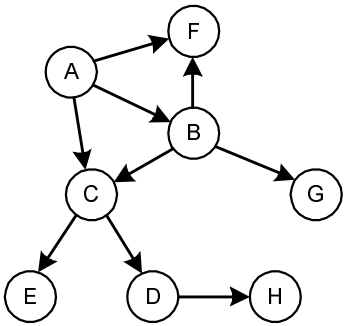
```
SELECT списокСтолбцов FROM Mytab T1 CROSS JOIN Mytab T2;
```

Примечание

Рассмотренные выражения работают в полнофункциональных базах данных. В Microsoft Access для получения декартового произведения возможно использование выражения (SELECT *списокСтолбцов* FROM T1, T2, ... Tn), только если все таблицы в списке имеют различные имена. Если требуется декартово произведение таблицы самой на себя, то в выражении (SELECT *списокСтолбцов* FROM Mytab T1, Mytab T2) Access автоматически добавит ключевое слово AS перед каждым псевдонимом. Попытка использования ключевых слов CROSS JOIN в Access приведет к ошибке.

Запросы на декартово произведение сами по себе очень редко используются. Они приобретают некоторый смысл, если применяются с оператором WHERE.

Допустим, что имеется таблица Рейсы (НАЧАЛЬНЫЙ_ПУНКТ, КОНЕЧНЫЙ_ПУНКТ), содержащая сведения о том, из каких пунктов и в какие можно попасть с помощью того или иного авиарейса.



НАЧАЛЬНЫЙ_ПУНКТ	КОНЕЧНЫЙ_ПУНКТ
A	B
A	C
A	F
B	C
B	G
B	F
C	E
C	D
D	H

Рис. 5.4. Граф и таблица достижимости пунктов

На рис. 5.4 показан граф достижимости пунктов некоторым авиарейсом и соответствующая ему таблица `Рейсы`. Каждой стрелке на графе и каждой записи таблицы соответствует некоторый рейс. Не трудно заметить, что из некоторых пунктов в другие можно попасть только с пересадкой на другой рейс, т. е. через транзитный пункт. Следующий запрос возвращает таблицу (рис. 5.5), содержащую сведения о достижимости пунктов в точности через один транзитный пункт:

```
SELECT T1.НАЧАЛЬНЫЙ_ПУНКТ, T2.КОНЕЧНЫЙ_ПУНКТ
FROM Рейсы T1, Рейсы T2
WHERE T1.КОНЕЧНЫЙ_ПУНКТ = T2.НАЧАЛЬНЫЙ_ПУНКТ;
```

НАЧАЛЬНЫЙ_ПУНКТ	КОНЕЧНЫЙ_ПУНКТ
A	C
A	G
A	F
A	E
A	D
B	E
B	D
C	H

Рис. 5.5. Таблица достижимости пунктов через один транзитный пункт

Рассмотрим данный запрос более подробно. Сначала он выполняет декартово произведение таблицы Рейсы на эту же таблицу. В результате получается таблица с четырьмя столбцами: T1.НАЧАЛЬНЫЙ_ПУНКТ, T1.КОНЕЧНЫЙ_ПУНКТ, T2.НАЧАЛЬНЫЙ_ПУНКТ и T2.КОНЕЧНЫЙ_ПУНКТ. Затем из полученной таблицы выбираются такие записи, в которых T1.КОНЕЧНЫЙ_ПУНКТ = T2.НАЧАЛЬНЫЙ_ПУНКТ. Это и есть пары пунктов, между которыми в графе достижимости находится один промежуточный пункт. Наконец, из четырех столбцов выделяются только два: T1.НАЧАЛЬНЫЙ_ПУНКТ и T2.КОНЕЧНЫЙ_ПУНКТ.

Запрос, содержащий сведения о том, в какие пункты можно попасть так или иначе, будет рассмотрен в *разд. 5.2.2*.

5.2.2. Объединение наборов записей (*UNION*)

Нередко требуется объединить записи двух или более таблиц с похожими структурами в одну таблицу. Иначе говоря, к набору записей, возвращаемому одним запросом, требуется добавить записи, возвращаемые другим запросом. Для этого служит оператор UNION (объединение):

```
Запрос1 UNION Запрос2;
```

При этом в результатной таблице остаются только отличающиеся записи. Чтобы сохранить в ней все записи, после оператора UNION следует написать ключевое слово ALL.

Например, таблицы Клиенты и Контакты имеют однотипные столбцы Имя и Адрес. Тогда, чтобы пополнить список данных о клиентах сведениями из таблицы Контакты, достаточно выполнить следующий запрос:

```
SELECT Имя, Адрес FROM Клиенты
UNION
SELECT Имя, Адрес FROM Контакты;
```

Возможно, что в обеих объединяемых таблицах, Клиенты и Контакты, имеются записи с одинаковыми парами значений в столбцах Имя и Адрес. Однако в результатной таблице данного

запроса повторений не будет, как если бы вы использовали оператор `UNION DISTINCT`.

Оператор `UNION` можно применять только таблицам, удовлетворяющим следующим условиям совместимости:

- количества столбцов объединяемых таблиц должны быть равны;
- данные в соответствующих столбцах объединяемых таблиц должны иметь совместимые типы. Например, символьные (строковые) типы `CHAR` и `VARCHAR` совместимы, а числовой и строковый типы не совместимы.

Обратите внимание, что имена соответствующих столбцов и их размеры могут быть различными. Важно, чтобы количества столбцов были равны, а их типы были совместимы. Чтобы объединить наборы записей с несовместимыми по типу данных столбцами, следует применить функцию преобразования типа данных `CAST()`. Например, следующий запрос возвращает список имен клиентов, к которому добавлен список сумм заказов (не будем обсуждать практическую пользу или смысл такого списка):

```
SELECT Имя FROM Клиенты
UNION
SELECT CAST(Сумма_заказа AS CHAR(10)) FROM Клиенты;
```

Когда требуется объединить записи двух таблиц, имеющих одноименные столбцы с совместимыми типами, можно использовать оператор `UNION CORRESPONDING` (объединение соответствующих):

```
SELECT * FROM Таблица1
UNION CORRESPONDING (списокСтолбцов)
SELECT * FROM Таблица2;
```

После ключевого слова `CORRESPONDING` можно указать столбцы, имеющиеся одновременно в обеих таблицах. Если этот список столбцов не указан, то предполагается список всех имен. При этом возможны недоразумения.

В *разд. 5.2.1* рассматривался запрос, возвращающий таблицу со сведениями о достижимости пунктов в точности через один промежуточный пункт (см. рис. 5.4 и 5.5). Теперь сформулируем запрос по-другому: требуется получить сведения о том, в какие пункты можно попасть, сделав не более одной пересадки (т. е. без

пересадок или с одной пересадкой). Для этого достаточно объединить записи исходной таблицы `Рейсы` с результатом запроса о достижимости через один промежуточный пункт:

```
SELECT НАЧАЛЬНЫЙ_ПУНКТ, КОНЕЧНЫЙ_ПУНКТ FROM Рейсы
UNION
SELECT T1.НАЧАЛЬНЫЙ_ПУНКТ, T2.КОНЕЧНЫЙ_ПУНКТ
    FROM Рейсы T1, Рейсы T2
WHERE T1.КОНЕЧНЫЙ_ПУНКТ = T2.НАЧАЛЬНЫЙ_ПУНКТ;
```

Результат данного запроса показан на рис. 5.6.

НАЧАЛЬНЫЙ_ПУНКТ	КОНЕЧНЫЙ_ПУНКТ
A	B
A	C
A	D
A	E
A	F
A	G
B	C
B	D
B	E
B	F
B	G
C	D
C	E
C	H
D	H

Рис. 5.6. Таблица достижимости пунктов, в которые можно попасть, сделав не более одной пересадки

Аналогичным образом можно сформулировать запрос о достижимости пунктов, в которые можно попасть посредством не более двух, трех и т. д. пересадок. Запрос, содержащий сведения о том, в какие пункты вообще можно попасть, будет рассмотрен далее в *разд. 5.3.5*.

Рассмотренный запрос возвращает сведения о достижимости пунктов из всех возможных начальных пунктов. А если нам нужны сведения о достижимости только из одного пункта, например, `A`? В этом случае достаточно добавить еще одно условие в оператор `WHERE`:

```
SELECT НАЧАЛЬНЫЙ_ПУНКТ, КОНЕЧНЫЙ_ПУНКТ FROM Рейсы
```

```
UNION
SELECT T1.НАЧАЛЬНЫЙ_ПУНКТ, T2.КОНЕЧНЫЙ_ПУНКТ
    FROM Рейсы T1, Рейсы T2
WHERE T1.КОНЕЧНЫЙ_ПУНКТ = T2.НАЧАЛЬНЫЙ_ПУНКТ
    AND T1.НАЧАЛЬНЫЙ_ПУНКТ = 'A';
```

Вместе с тем, допустим и такой эквивалентный запрос:

```
SELECT * FROM (
    SELECT НАЧАЛЬНЫЙ_ПУНКТ, КОНЕЧНЫЙ_ПУНКТ FROM Рейсы
    UNION
    SELECT T1.НАЧАЛЬНЫЙ_ПУНКТ, T2.КОНЕЧНЫЙ_ПУНКТ
        FROM Рейсы T1, Рейсы T2
    WHERE T1.КОНЕЧНЫЙ_ПУНКТ = T2.НАЧАЛЬНЫЙ_ПУНКТ) T
WHERE T.НАЧАЛЬНЫЙ_ПУНКТ = 'A';
```

5.2.3. Пересечение наборов записей (*INTERSECT*)

Пересечение двух наборов записей осуществляется с помощью оператора `INTERSECT` (пересечение), возвращающего таблицу, записи в которой содержатся одновременно в двух наборах:

```
Запрос1 INTERSECT Запрос2;
```

При этом в результатной таблице остаются только отличающиеся записи. Чтобы сохранить в ней повторяющиеся записи, после оператора `INTERSECT` следует написать ключевое слово `ALL`.

Как и в операторе `UNION`, в `INTERSECT` можно использовать ключевое слово `CORRESPONDING`. В этом случае исходные наборы данных не обязательно должны быть совместимыми для объединения, но соответствующие столбцы должны иметь одинаковые тип и длину.

Как известно, различные технические системы могут содержать одинаковые компоненты. Если требуется получить список компонентов, входящих одновременно в две различные системы, то можно воспользоваться таким запросом:

```
SELECT * FROM Система1
INTERSECT
SELECT * FROM Система2;
```

Здесь предполагается, что таблицы Система1 и Система2 имеют одинаковые структуры. Если же структуры этих таблиц в чем-то различаются, но столбцы ID_компонента и Тип_компонента имеют одинаковые тип и длину, то можно применить следующий запрос:

```
SELECT * FROM Система1
INTERSECT CORRESPONDING (ID_компонента, Тип_компонента)
SELECT * FROM Система2;
```

Если в операторе INTERSECT используется ключевое слово CORRESPONDING, то после него в круглых скобках можно указать имена столбцов, значения которых должны проверяться на равенство при выполнении операции пересечения.

5.2.4. Вычитание наборов записей (*EXCEPT*)

Для получения записей, содержащихся в одном наборе и отсутствующих в другом, служит оператор EXCEPT (за исключением):

```
Запрос1 EXEPT Запрос2;
```

С помощью этого оператора из первого набора удаляются записи, входящие во второй набор. Так же, как и в операторах UNION и INTERSECT, в операторе EXCEPT можно использовать ключевое слово CORRESPONDING.

Например, таблицы Клиенты и Контакты имеют однотипные столбцы Имя и Адрес. Чтобы узнать, все ли клиенты содержатся в списке контактов, можно воспользоваться следующим запросом:

```
SELECT * FROM Клиенты
EXCEPT CORRESPONDING (Имя, Адрес)
SELECT * FROM Контакты;
```

Возвращенные этим запросом записи будут содержать сведения о клиентах, которых нет в таблице Контакты. Если же запрос вернет пустую таблицу, то это будет означать, что все клиенты представлены в таблице Контакты.

5.3. Операции соединения

Операции соединения наборов записей возвращают таблицы, записи в которых получаются путем некоторой комбинации

записей соединяемых таблиц. Для этого используется оператор `JOIN` (соединить).

Существуют несколько разновидностей соединения, которым соответствуют определенные ключевые слова, добавляемые к слову `JOIN`. Так, например, декартово произведение *(см. разд. 5.2.1)* является операцией перекрестного соединения. В SQL-выражении для обозначения этой операции используется оператор `CROSS JOIN`. Впрочем, декартово произведение можно получить и без использования этих ключевых слов. В основе любого соединения наборов записей лежит операция их декартового произведения.

Примечание

Довольно часто операции, основанные на операторе `JOIN`, называют объединением таблиц (наборов записей). Однако термин "объединение" лучше подходит для `UNION` — оператора теоретико-множественного объединения записей, при котором записи исходных наборов не комбинируются (не соединяются) друг с другом, а просто к одному набору записей добавляется другой набор. В случае оператора `JOIN` в результатную таблицу попадают записи, полученные из разных наборов путем присоединения одной из них к другой. Поэтому операции, основанные на операторе `JOIN`, будем называть операциями соединения таблиц (наборов записей).

5.3.1. Естественное соединение (*NATURAL JOIN*)

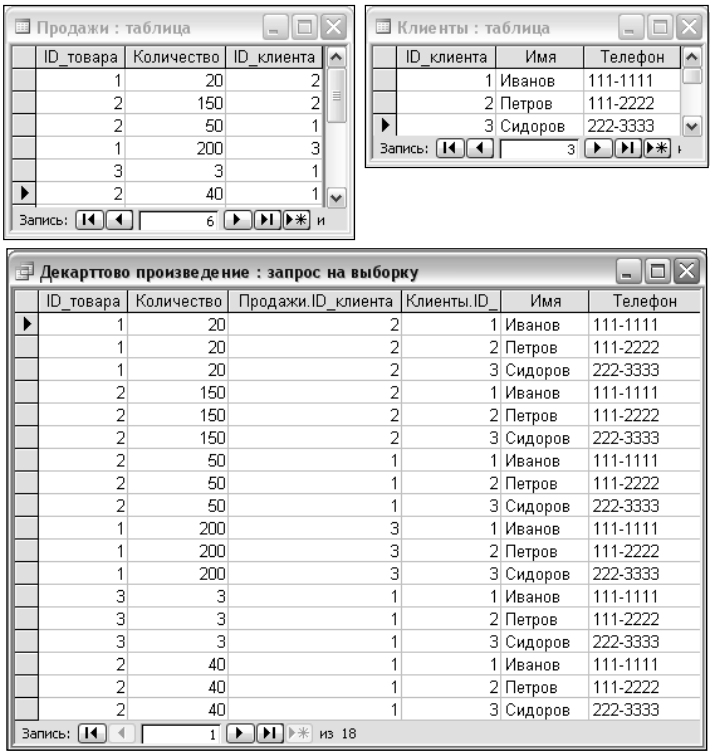
Рассмотрим суть операции естественного соединения на типичном примере. Пусть в базе данных имеются следующие две таблицы:

- `Продажи` (`ID_товара`, `Количество`, `ID_клиента`);
- `Клиенты` (`ID_клиента`, `Имя`, `Телефон`).

Общим столбцом для этих таблиц является `ID_клиента`. Декартово произведение этих таблиц получается с помощью следующего запроса:

```
SELECT * FROM Продажи, Клиенты;
```

На рис. 5.7 показаны примеры таблиц `Продажи` и `Клиенты`, а также результат их декартового произведения.



Продажи : таблица

ID_товара	Количество	ID_клиента
1	20	2
2	150	2
2	50	1
1	200	3
3	3	1
2	40	1

Запись: 6 и

Клиенты : таблица

ID_клиента	Имя	Телефон
1	Иванов	111-1111
2	Петров	111-2222
3	Сидоров	222-3333

Запись: 3

Декарттово произведение : запрос на выборку

ID_товара	Количество	Продажи.ID_клиента	Клиенты.ID_	Имя	Телефон
1	20	2	1	Иванов	111-1111
1	20	2	2	Петров	111-2222
1	20	2	3	Сидоров	222-3333
2	150	2	1	Иванов	111-1111
2	150	2	2	Петров	111-2222
2	150	2	3	Сидоров	222-3333
2	50	1	1	Иванов	111-1111
2	50	1	2	Петров	111-2222
2	50	1	3	Сидоров	222-3333
1	200	3	1	Иванов	111-1111
1	200	3	2	Петров	111-2222
1	200	3	3	Сидоров	222-3333
3	3	1	1	Иванов	111-1111
3	3	1	2	Петров	111-2222
3	3	1	3	Сидоров	222-3333
2	40	1	1	Иванов	111-1111
2	40	1	2	Петров	111-2222
2	40	1	3	Сидоров	222-3333

Запись: 1 из 18

Рис. 5.7. Таблицы `Продажи`, `Клиенты` и их декартово произведение

Очевидно, в полученном декартовом произведении нас могут интересовать не все записи, а только те, в которых идентичные столбцы имеют одинаковые значения (`Продажи.ID_клиента = Клиенты.ID_клиента`). Кроме того, в результатной таблице нам не нужны оба идентичных столбца, достаточно лишь одного из них. Такая таблица и будет естественным соединением таблиц `Продажи` и `Клиенты` (рис. 5.8). Она получается с помощью следующего запроса:

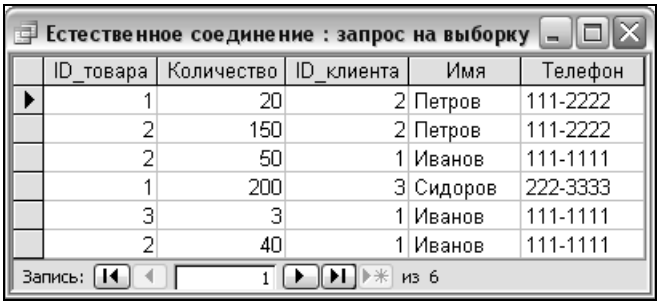
```
SELECT Продажи.*, Клиенты.Имя, Клиенты.Телефон
FROM Продажи, Клиенты
WHERE Продажи.ID_клиента = Клиенты.ID_клиента;
```

Данный запрос можно переписать, используя псевдонимы:

```
SELECT T1.*, T2.Имя, T2.Телефон
FROM Продажи T1, Клиенты T2
WHERE T1.ID_клиента = T2.ID_клиента;
```

Эквивалентный запрос с оператором `NATURAL JOIN` выглядит следующим образом:

```
SELECT T1.*, T2.Имя, T2.Телефон
FROM Продажи T1 NATURAL JOIN Клиенты T2;
```



Естественное соединение : запрос на выборку

ID_товара	Количество	ID_клиента	Имя	Телефон
1	20	2	Петров	111-2222
2	150	2	Петров	111-2222
2	50	1	Иванов	111-1111
1	200	3	Сидоров	222-3333
3	3	1	Иванов	111-1111
2	40	1	Иванов	111-1111

Запись: 1 из 6

Рис. 5.8. Естественное соединение таблиц `Продажи` и `Клиенты`

Примечание

В Microsoft Access оператор `NATURAL JOIN` не поддерживается. Вместо него используется `INNER JOIN` (внутреннее соединение) и ключевое слово `ON` (при), за которым следует условие отбора записей. Впрочем, `INNER JOIN` можно применять и в полнофункциональных базах данных.

При естественном соединении, выполняемом с помощью оператора `NATURAL JOIN`, проверяется равенство всех одноименных столбцов соединяемых таблиц.

5.3.2. Условное соединение (*JOIN ... ON*)

Условное соединение похоже на соединение с условием равенства, которое рассматривалось в начале *разд. 5.3.1*. Отличие состоит в том, что в качестве условия может выступать любое логическое выражение, которое записывается после ключевого слова `ON`

(при), а не WHERE. Если условие выполняется для текущей записи декартового произведения, то она входит в результатную таблицу.

Допустим, в базе данных имеются следующие две таблицы:

- Продажи (ID_товара, Количество, ID_клиента);
- Клиенты (ID_клиента, Имя, Телефон).

Тогда эти таблицы можно соединить, используя, например, следующий запрос:

```
SELECT * FROM Продажи JOIN Клиенты
    ON (Продажи.ID_клиента = Клиенты.ID_клиента)
    AND (Продажи.Количество > 50);
```

Примечание

В Microsoft Access используется оператор INNER JOIN ... ON. В полнофункциональных базах данных также допустимо ключевое слово INNER.

5.3.3. Соединение по именам столбцов (*JOIN ... USING*)

Соединение по именам столбцов похоже на естественное соединение. Отличие состоит в том, что можно указать, какие одноименные столбцы должны проверяться. Напомню, что в естественном соединении проверяются все одноименные столбцы.

Допустим, имеются две таблицы с одинаковыми структурами:

- Болты (Тип, Количество, Материал);
- Гайки (Тип, количество, Материал).

Предположим, что мы готовим крепежные комплекты, в каждом из которых количества однотипных болтов и гаек должны совпадать, а их материалы могут быть различными. Требуется узнать, какие комплекты уже готовы и сколько элементов они содержат. Для этого следует определенным образом соединить таблицы Болты и Гайки. Естественное соединение в этому случае не подойдет, поскольку при нем проверяются все одноименные столбцы, а потому в результатную таблицу не попадут комплекты болтов и гаек из различных материалов. Например, если все болты стальные, а гайки латунные, то в результате естественного соединения будет по-

лучена пустая таблица. Поэтому следует использовать соединение по именам столбцов, при котором столбец `Материал` исключен. Это соединение можно быть представлено так:

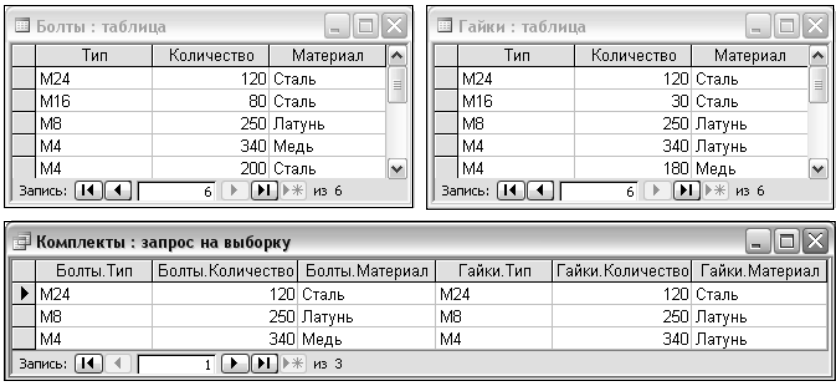
```
SELECT * FROM Болты JOIN Гайки
    USING (Тип, Количество);
```

После ключевого слова `USING` (используя) в круглых скобках указывается список одноименных столбцов соединяемых таблиц, которые необходимо проверять.

На рис. 5.9 показаны таблицы `Болты` и `Гайки`, а также результат рассмотренного запроса.

Очевидно, данный запрос можно сформулировать иначе:

```
SELECT * FROM Болты, Гайки
WHERE (Болты.Тип = Гайки.Тип) AND (Болты.Количество =
Гайки.Количество);
```



Болты : таблица

Тип	Количество	Материал
M24	120	Сталь
M16	80	Сталь
M8	250	Латунь
M4	340	Медь
M4	200	Сталь

Запись: 6 из 6

Гайки : таблица

Тип	Количество	Материал
M24	120	Сталь
M16	30	Сталь
M8	250	Латунь
M4	340	Латунь
M4	180	Медь

Запись: 6 из 6

Комплекты : запрос на выборку

Болты.Тип	Болты.Количество	Болты.Материал	Гайки.Тип	Гайки.Количество	Гайки.Материал
M24	120	Сталь	M24	120	Сталь
M8	250	Латунь	M8	250	Латунь
M4	340	Медь	M4	340	Латунь

Запись: 1 из 3

Рис. 5.9. Таблицы `Болты` и `Гайки` и результат их соединения по столбцам `Тип` и `Количество`

Примечание

В Microsoft Access оператор `JOIN ... USING` не поддерживается. Вместо него можно использовать `INNER JOIN ... ON`. Рассмотренный запрос в Access можно сформулировать так:

```
SELECT *
FROM Болты INNER JOIN Гайки
ON (Болты.Тип = Гайки.Тип) AND (Болты.Количество =
Гайки.Количество);
```

5.3.4. Внешние соединения

Все соединения таблиц, рассмотренные до сих пор, являются внутренними. Во всех примерах вместо ключевого слова `JOIN` можно писать `INNER JOIN` (внутреннее соединение). Из таблицы, получаемой при внутреннем соединении, отбраковываются все записи, для которых нет соответствующих записей одновременно в обеих соединяемых таблицах. При внешнем соединении такие несоответствующие записи сохраняются. В этом и заключается отличие внешнего соединения от внутреннего.

С помощью специальных ключевых слов `LEFT OUTER`, `RIGHT OUTER`, `FULL` и `UNION`, написанных перед `JOIN`, можно выполнить соответственно левое, правое, полное соединение и объединение-соединение. В SQL-выражении запроса таблица, указанная слева от оператора `JOIN`, называется левой, а указанная справа от него — правой.

Левое соединение (*LEFT OUTER JOIN*)

При левом внешнем соединении несоответствующие записи, имеющиеся в левой таблице, сохраняются в результатной таблице, а имеющиеся в правой — удаляются.

Допустим, в базе данных есть две таблицы:

- `Предложение` (`ID_товара`, `Цена`, `Описание`);
- `Склад` (`ID_товара`, `Количество`).

Таблица `Предложение` содержит список идентификаторов, цены и описания товаров, предлагаемых к продаже. Таблица `Склад` содержит сведения о наличии товаров на складе. При этом в таблице `Склад` могут содержаться не все товары, предлагаемые к продаже.

Чтобы получить список всех продаваемых товаров с указанием их количества на складе, достаточно выполнить следующий запрос:

```
SELECT Предложение.ID_товара, Предложение.Описание,
Склад.Количество
FROM Предложение LEFT OUTER JOIN Склад
ON Предложение.ID_товара = Склад.ID_товара;
```

Здесь ключевые слова `LEFT OUTER JOIN` означают операцию левого внешнего соединения.

Примечание

В Microsoft Access ключевое слово `OUTER` не допускается. Поскольку левого внутреннего объединения не существует, то можно просто писать `LEFT JOIN`.

На рис. 5.10 показаны исходные таблицы и результат данного запроса. Пустые поля в действительности имеют неотображаемое значение `NULL`.

Рис. 5.10. Таблицы `Предложение`, `Склад` и результат их левого внешнего соединения

Эквивалентный запрос, не использующий ключевые слова `LEFT OUTER JOIN`, выглядит довольно громоздко:

```
SELECT Предложение.ID_товара, Предложение.Описание,
Склад.Количество
    FROM Предложение, Склад
    WHERE Предложение.ID_товара=Склад.ID_товара
UNION
```

```
SELECT Предложение.ID_товара, Предложение.Описание, NULL
  FROM Предложение
  WHERE Предложение.ID_товара NOT IN
    (SELECT Предложение.ID_товара FROM Предложение, Склад
      WHERE Предложение.ID_товара = Склад.ID_товара);
```

Этот запрос состоит из объединения (`UNION`) двух запросов, второй из которых содержит подзапрос. Чтобы выровнять количества столбцов в объединяемых таблицах, во втором запросе в список столбцов был добавлен `NULL`. Впрочем, вместо `NULL` можно указать 0 (ноль) и тогда в столбце `Количество` для всех отсутствующих товаров будет стоять число 0.

Правое соединение (*RIGHT OUTER JOIN*)

При правом внешнем соединении несоответствующие записи, имеющиеся в правой таблице, сохраняются в результатной таблице, а имеющиеся в левой — удаляются. Ключевыми словами в запросе с таким соединением являются `RIGHT OUTER JOIN`.

Чтобы решить задачу, рассмотренную в предыдущем разделе, используя правое соединение, достаточно просто поменять местами имена соединяемых таблиц:

```
SELECT Предложение.ID_товара, Предложение.Описание,
Склад.Количество
FROM Склад RIGHT OUTER JOIN Предложение
ON Предложение.ID_товара = Склад.ID_товара;
```

Примечание

В Microsoft Access ключевое слово `OUTER` не допускается. Поскольку правого внутреннего объединения не существует, то можно просто писать `RIGHT JOIN`.

Полное соединение (*FULL JOIN*)

Полное соединение выполняет одновременно и левое, и правое внешние соединения. Ключевыми словами в запросе с таким соединением являются `FULL JOIN`.

Допустим, база данных некоторой компании содержит сведения о своих представительствах, отделах и сотрудниках. Эта информация хранится в трех таблицах:

- `Представительства` (`ID_пр`, `Адрес`);
- `Отделы` (`ID_отд`, `ID_пр`, `Название`);
- `Сотрудники` (`ID_сотр`, `ID_отд`, `Имя`).

Таким образом, в таблице `Сотрудники` имеются данные об отделах, в которых они работают, а в таблице `Отделы` — о представительствах, в которые они входят. В общем случае в этой базе данных могут быть:

- представительства без отделов;
- отделы, не входящие ни в одно представительство;
- отделы без сотрудников;
- сотрудники, не входящие ни в один отдел.

Чтобы просмотреть все представительства, отделы и всех сотрудников, не зависимо от того, имеются ли в одних таблицах соответствующие записи в других, используется полное внешнее соединение:

```
SELECT *
FROM Представительства FULL JOIN Отделы
    ON (Представительства.ID_пр = Отделы. ID_пр)
FULL JOIN Сотрудники
    ON (Отделы.ID_отд = Сотрудники.ID_отд);
```

Примечание

В Microsoft Access оператор `FULL JOIN` не поддерживается.

Объединение-соединение (*UNION JOIN*)

Операцию объединение-соединение еще называют объединение-слияние. Ключевыми словами в запросе с таким соединением являются `UNION JOIN`.

При объединении-соединении создается виртуальная таблица, содержащая все столбцы обеих исходных таблиц. При этом столбцы из левой исходной таблицы содержат все записи левой

таблицы, а в тех же записях в столбцах из правой исходной таблицы содержатся значения NULL. Аналогично, столбцы из правой исходной таблицы содержат все записи правой таблицы, а эти же записи в столбцах из левой таблицы содержат NULL. Общее количество записей в виртуальной таблице равно сумме количеств записей, имеющихся в обеих исходных таблицах.

В качестве примера приведем простой запрос:

```
SELECT * FROM T1 UNION JOIN T2;
```

На рис. 5.11 показаны две таблицы и результат их объединения-соединения.

Обычно результат объединения-соединения рассматривается в качестве промежуточного при выполнении более сложного запроса.

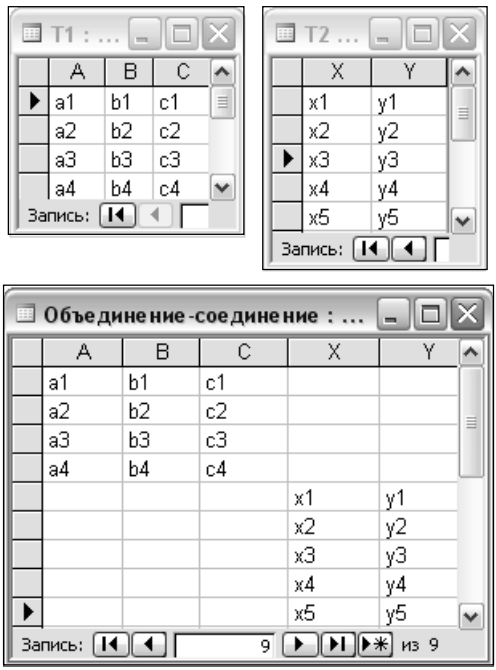


Рис. 5.11. Две исходных таблицы и результат их объединения-соединения

5.3.5. Рекурсивные запросы

В некоторых языках программирования допускается рекурсия. Так, в определении функции возможен вызов этой же функции. Это мощная возможность языка программирования, хотя и редко используемая на практике. Типичным примером, приводимым в литературе для пояснения, что такое рекурсия, является определение функции, вычисляющей факториал заданного числа.

Факториал целого числа *n*, обозначаемый как *n*!, есть целое число, равное:

- 1, если *n* <= 1;
- 2 * 3 * 4 * ... * *n*, если *n* > 1.

Таким образом, *n*! равно произведению всех целых чисел, не больших *n*, или 1, если *n* меньше или равно 1.

Эту простую математическую задачу для заданного числа *n* можно решить программно различными способами, например, с помощью оператора цикла, который есть практически во всех языках. Однако более изящный код получается, если использовать рекурсию. Так, например, на языке JavaScript, на котором пишутся программы для Web-приложений, определение функции вычисления факториала посредством рекурсии выглядит следующим образом:

```
function factorial(n) {
    if (n <= 1) return 1;
    return n*factorial(n-1);
}
```

Рассмотрим в качестве примера вычисление факториала для *n* = 3.

1. Вызов функции `factorial(3)` вернет `3*factorial(2)`, т. е. приведет к вызову функции `factorial(2)`. Таким образом, требуется вторая итерация вычислений.
2. Вызов функции `factorial(2)` вернет `2*factorial(1)`. С учетом результата первой итерации получаем `3*2*factorial(1)`. Требуется вызов функции `factorial(1)`.

3. Вызов функции `factorial(1)` вернет 1. С учетом результата второй итерации получаем `3*2*1`. Поскольку больше вызовов функции `factorial()` не требуется, на этом вычислительный процесс завершается и возвращается результат 6.

Рекурсивным называется запрос, который содержит в себе этот же запрос. Более сложный случай возникает тогда, когда *Запрос1* функционально зависит от *Запроса2*, который, в свою очередь, зависит от *Запроса1*.

В ранних версиях SQL рекурсивные запросы не поддерживались. В SQL:1999 и SQL:2003 появились расширения, поддерживающие рекурсивные запросы. Однако в основной части стандарта SQL:2003 поддержка рекурсии не предусмотрена. Так, Microsoft Access не поддерживает рекурсивные запросы.

В *разд. 5.2.1* рассматривалась задача определения достижимости пунктов на основе таблицы `Рейсы` (см. рис. 5.4). Точнее, в таблице `Рейсы` содержались сведения о том, какие пункты связаны между собой тем или иным авиарейсом. При этом рассматривался запрос, возвращающий таблицу достижимости пунктов через один промежуточный пункт. В *разд. 5.2.2* был рассмотрен запрос, возвращающий таблицу достижимости пунктов с помощью не более одного промежуточного пункта. По аналогии можно сформулировать запросы о достижимости пунктов, в которые можно попасть с помощью не более двух, трех и т. д. пересадок. При этом сложность соответствующего SQL-выражения будет только возрастать. Чтобы решить общую задачу о том, в какие пункты можно вообще попасть, сделав сколько угодно пересадок, требуется рекурсивный запрос:

```
WIDTH RECURSIVE
    Достижимость (НАЧАЛЬНЫЙ_ПУНКТ, КОНЕЧНЫЙ_ПУНКТ)
    AS ( SELECT НАЧАЛЬНЫЙ_ПУНКТ, КОНЕЧНЫЙ_ПУНКТ
        FROM Рейсы
        UNION
        SELECT T1.НАЧАЛЬНЫЙ_ПУНКТ, T2.КОНЕЧНЫЙ_ПУНКТ
        FROM Достижимость T1, Рейсы T2
```

```
        WHERE T1.КОНЕЧНЫЙ_ПУНКТ = T2.НАЧАЛЬНЫЙ_ПУНКТ
        )
SELECT * FROM Достижимость;
```

Данный запрос выполняется итеративно в несколько проходов.

1. В начале первой итерации во временной таблице `Достижимость` нет ни одной записи. Оператор `UNION` (объединить) копирует строки из таблицы `Рейсы` в таблицу `Достижимость`.
2. При второй итерации просматриваются только те записи, в которых значение столбца `КОНЕЧНЫЙ_ПУНКТ` таблицы `Достижимость` равно значению `НАЧАЛЬНЫЙ_ПУНКТ` таблицы `Рейсы`. Для каждой такой записи берутся значения `НАЧАЛЬНЫЙ_ПУНКТ` из таблицы `Достижимость` и значения `КОНЕЧНЫЙ_ПУНКТ` из таблицы `Рейсы`, а затем в качестве новой записи добавляются в таблицу `Достижимость`. Теперь в таблице `Достижимость` находятся все пункты, в которые можно попасть из любого пункта, находящегося в столбце `НАЧАЛЬНЫЙ_ПУНКТ` той же таблицы, делая при этом не более одной пересадки.
3. Во время следующей рекурсивной итерации будут обработаны маршруты с двумя промежуточными пунктами и т. д., пока не будут найдены все пункты, куда в принципе можно попасть.
4. По окончании рекурсии последний оператор `SELECT`, не участвующий в итерациях, возвращает содержимое таблицы `Достижимость`. При необходимости его можно дополнить выражением `WHERE` с условием выборки нужных записей.

Обратите внимание, что в рассмотренном запросе временная таблица `Достижимость` определяется на основе ее самой. Это обстоятельство и делает запрос рекурсивным. Рекурсивной частью определения является оператор `SELECT`, расположенный сразу за оператором `UNION`. Временная таблица `Достижимость` наполняется данными по мере выполнения рекурсии до тех пор, пока все возможные конечные пункты не окажутся в этой таблице. Дублирующих записей в этой таблице не будет, поскольку оператор `UNION` их подавляет.

Рекурсивные запросы позволяют писать экономные по объему кода и ясные по своей структуре SQL-выражения в случаях обра-

ботки таблиц с данными, которые имеют иерархическую организацию. Например, сложные технические системы обычно состоят из компонентов, которые в свою очередь состоят из других компонентов и т. д. Связи между компонентами можно хранить в одной таблице, в одном из столбцов которой находятся идентификаторы компонентов, а в другом — идентификаторы элементов, входящих в состав компонентов. При этом элемент некоторого компонента сам может содержать другие элементы. Таким образом, один и тот же идентификатор элемента может находиться и в первом, и во втором столбцах таблицы, содержащей сведения о связях компонент-элемент. Пример такой таблицы показан на рис. 5.12.

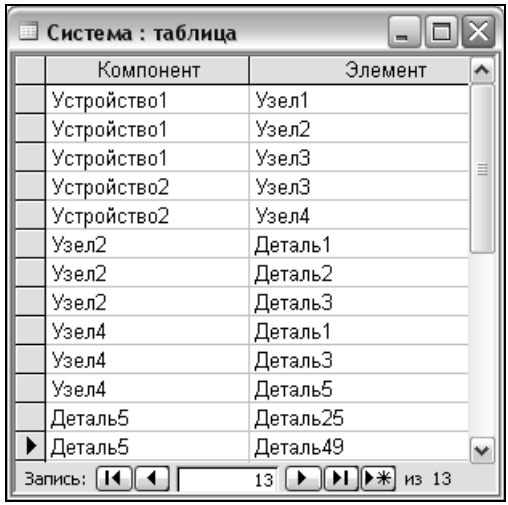


Рис. 5.12. Таблица связей компонент-элемент для некоторой системы

Типичный запрос к такой таблице: дать список всех элементов, из которых состоит компонент, например, `Устройство1`. Не трудно заметить, что эта задача аналогична рассмотренной ранее задаче о достижимости пунктов.

5.4. Задачи

Задача 5.1

В *разд. 4.1* рассматривались итоговые функции, позволяющие производить элементарные вычисления над числовыми данными в таблице. В данной главе вы познакомились со сложными запросами и, следовательно, можете выполнять более сложные вычисления. В качестве упражнения я предлагаю вам задачу, которая на первый взгляд может показаться слишком специальной, представляющей интерес лишь для математиков. С этой задачей чаще всего сталкиваются люди, занимающиеся статистическими расчетами. Однако она полезна для всех изучающих SQL, поскольку связана с комбинацией итоговых функций и вложенных запросов. Такого рода вычисления возникают при анализе данных.

Предлагается вычислить среднеквадратическое отклонение (СКО) некоторой величины. СКО характеризует разброс данных относительно их среднего значения. Напомню, что среднеквадратическое отклонение величины X, обозначаемое как $\sigma(X)$, есть среднее значение отклонения значений величины X от ее среднего значения, обозначаемого здесь как $M(X)$.

Если величина X представлена списком своих значений $x_1, x_2, \ldots, x_N$, где N — количество всех значений X, то СКО рассчитывается по формуле:

$$\sigma(X) = \left(\left(\sum_{i=1}^{N} \left(x_i - M(X) \right) \right)^2 \Big/ N(N-1) \right)^{1/2},$$

где $M(X) = \sum_{i=1}^{N} x_i \Big/ N$ — среднее значение величины X.

Иначе говоря, для вычисления СКО величины X следует выполнить следующие вычисления.

1. Вычислить среднее значение $M(X)$.
2. Вычислить сумму всех квадратов разностей $\left(x_i - M(X) \right)$, т. е. отклонений значений величины X от их среднего значения $M(X)$.

3. Полученный результат поделить на $N(N-1)$.

4. Из полученного результата извлечь квадратный корень.

Допустим, у нас имеется список значений некоторой величины, например, числовой столбец `X` в таблице `Измерения`. Пример такой таблицы показан на рис. 5.13. Требуется вычислить СКО значений этого столбца.

Нетрудно подсчитать, что среднее значение столбца `X` таблицы `Измерения` равно 7. Среднее значений столбца, как известно, можно получить с помощью функции `AVG()`. Сначала посмотрим, каковы отклонения значений величины X от среднего значения. Для этого выполним следующий запрос:

```
SELECT Измерения.X, X–M AS Отклонение_от_ср
FROM Измерения, (SELECT AVG(X) AS M FROM Измерения);
```

Результат данного запроса показан на рис. 5.14.

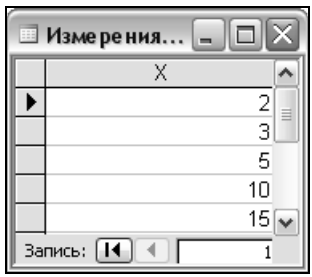


Рис. 5.13. Таблица с числовыми значениями

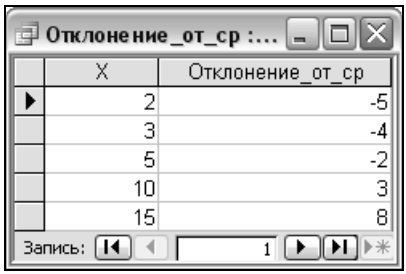


Рис. 5.14. Таблица отклонений значений столбца `X` от среднего значения

Обратите внимание на использование декартового произведения исходной таблицы на результат, возвращаемый подзапросом, который вычисляет среднее значение (`AVG(X)`).

Однако в качестве промежуточного решения поставленной задачи больше подойдет следующий запрос:

```
SELECT SUM((X-M)*(X-M)) AS Сумма_кв_отклонений
FROM Измерения, (SELECT AVG(X) AS M FROM Измерения);
```

Для таблицы, показанной на рис. 5.13, данный запрос вернет 118. Заметим, что в качестве параметра функции `SUM()` можно было бы передать `POWER((X-M), 2)`, т. е. выражение, вызывающее функцию возведения в степень. Однако в Microsoft Access этот вариант не пройдет, поскольку Access использует вычислительные функции из арсенала языка VBA, а не SQL.

Теперь осталось совсем немного сделать, чтобы получить среднеквадратическое отклонение. Для этого вам понадобятся операции определения количества значений, деления и извлечения квадратного корня *(см. разд. 4.2.2)*. Попробуйте самостоятельно сформировать запрос на получение СКО значений столбца `X` таблицы `Измерения`.

Задача 5.2

В *разд. 5.2.1* рассматривалась таблица `Рейсы` (`НАЧАЛЬНЫЙ_ПУНКТ`, `КОНЕЧНЫЙ_ПУНКТ`), содержащая сведения о том, из каких пунктов и в какие можно попасть с помощью того или иного авиарейса. На рис. 5.4 показан граф достижимости пунктов авиарейсами и соответствующая ему таблица. Каждой стрелке на графе и каждой записи таблицы соответствует некоторый рейс. Так, например, из пункта A можно попасть в пункт B посредством некоторого авиарейса, но из B в A авиарейса нет. Допустим, авиакомпания, осуществляющая перевозки по маршрутам, организовала для каждого прямого рейса соответствующий ему обратный рейс. Например, если в исходной таблице имеется запись (A, B), то в модифицированной таблице должна быть запись (B, A).

Сформулируйте запрос к исходной таблице `Рейсы`, возвращающий таблицу, содержащую сведения как о прямых, так и об обратных рейсах.

Задача 5.3

Для таблицы `Рейсы` (`НАЧАЛЬНЫЙ_ПУНКТ`, `КОНЕЧНЫЙ_ПУНКТ`), которая показана на рис. 5.4, сформулируйте запрос, который возвращает таблицу со сведениями о достижимости пунктов, в которые можно попасть посредством не более двух пересадок. Решите эту задачу, не используя рекурсивных запросов.

Задача 5.4

Для таблицы `Рейсы` (`НАЧАЛЬНЫЙ_ПУНКТ`, `КОНЕЧНЫЙ_ПУНКТ`), которая показана на рис. 5.4, сформулируйте запрос, который возвращает список пунктов, в которые можно попасть как из пункта A, так и из пункта B.

Задача 5.5

Пусть имеется таблица `Сотрудники` (`ID_сотрудника`, `Описание`, `ID_начальника`), показанная на рис. 5.15. Эта таблица содержит список идентификаторов (`ID_сотрудника`) всех сотрудников некоторой фирмы.

Сотрудники : таблица

ID_сотрудника	Описание	ID_начальника
1	Генеральный директор	
2	Главный бухгалтер	1
3	Заместитель директора	1
4	Нач. транспортного отдела	3
5	Нач. отдела маркетинга	3
6	Нач. склада	3
7	Бухгалтер	2
8	Водитель	4
9	Автомеханик	4
10	Слесарь	4
11	Кладовщик	6
12	Сторож	6
13	Секретарь	1
14	Уборщица	3

Запись: 14 из 14

Рис. 5.15. Таблица со сведениями о субординации сотрудников

Для каждого сотрудника указан идентификатор его непосредственного начальника (`ID_начальника`). Любое определенное значение столбца `ID_начальника` обязательно встречается и среди значений столбца `ID_сотрудника`, поскольку любой начальник является в то же время и сотрудником фирмы. В общем случае некоторые сотрудники могут не иметь подчиненных. Таким образом, данная таблица содержит информацию об иерархии (субординации) сотрудников фирмы.

Сформулируйте запрос, возвращающий список всех сотрудников, которые непосредственно подчинены тому же начальнику, что и данный сотрудник. Если решить данную задачу, например, относительно водителя, то из таблицы на рис. 5.15 видно, что он непосредственно подчинен начальнику транспортного отдела и, следовательно, необходимо определить всех сотрудников этого отдела.

Сформулируйте запрос, возвращающий список всех сотрудников, которые подчинены (в том числе и опосредованно) тому же начальнику, что и данный сотрудник.

Глава 6

Добавление, удаление и изменение данных в таблицах

Запросы, рассмотренные в предыдущих главах, были направлены на то, чтобы получить данные, содержащиеся в существующих таблицах базы данных. Главным ключевым словом таких запросов на выборку данных является `SELECT`. Запросы на выборку данных всегда возвращают виртуальную таблицу, которая отсутствует в базе данных и создается временно лишь для того, чтобы представить выбранные данные пользователю. При создании и дальнейшем сопровождении базы данных обычно возникает задача добавления новых и удаления ненужных записей, а также изменения содержимого ячеек таблицы. В SQL для этого предусмотрены операторы `INSERT` (вставить), `DELETE` (удалить) и `UPDATE` (изменить). Запросы, начинающиеся с этих ключевых слов, не возвращают данные в виде виртуальной таблицы, а изменяют содержимое уже существующих таблиц базы данных. Запросы на модификацию (добавление, удаление и изменение) данных могут содержать вложенные запросы на выборку данных из той же самой таблицы или из других таблиц *(см. гл. 5)*, однако сами не могут быть вложены в другие запросы. Таким образом, операторы `INSERT`, `DELETE` и `UPDATE` в SQL-выражении могут находиться только в самом начале.

6.1. Добавление новых записей

Когда создается таблица базы данных, она не содержит никаких записей, т. е. является пустой. Чтобы наполнить таблицу данными, необходимо добавить (вставить) в нее хотя бы одну запись. Для этого служит оператор INSERT (вставить), который имеет несколько форм:

- INSERT INTO *имяТаблицы* VALUES (*списокЗначений*) — вставляет пустую запись в указанную таблицу и заполняет эту запись значениями из списка, указанного за ключевым словом VALUES. При этом первое в списке значение вводится в первый столбец таблицы, второе значение — во второй столбец и т. д. Порядок столбцов задается при создании таблицы. Данная форма оператора INSERT не очень надежна, поскольку нетрудно ошибиться в порядке вводимых значений. Более надежной и гибкой является следующая форма;
- INSERT INTO *имяТаблицы* (*списокСтолбцов*) VALUES (*списокЗначений*) — вставляет пустую запись в указанную таблицу и вводит в заданные столбцы значения из указанного списка. При этом в первый столбец из *списокСтолбцов* вводится первое значение из *списокЗначений*, во второй столбец — второе значение и т. д. Порядок имен столбцов в списке может отличаться от их порядка, заданного при создании таблицы. Столбцы, которые не указаны в списке, заполняются значением NULL. Иногда требуется просто добавить пустую запись. В этом случае первая форма оператора INSERT требует после ключевого слова VALUES указать список значений NULL, длина которого равна количеству столбцов. Однако есть лучшее решение:

  ```
  INSERT INTO имяТаблицы (имяЛюбогоСтолбца) VALUES (NULL);
  ```

  Рекомендуется использовать именно данную форму оператора INSERT. Следующий запрос добавляет новую запись в таблицу Клиенты, при этом в столбцы Имя, Телефон и Сумма_заказа вводятся значения 'Петров', '444-4444' и 25300 соответственно:

  ```
  INSERT INTO Клиенты (Имя, Телефон, Сумма_заказа)
  VALUES ('Петров', '444-4444', 25300);
  ```

Начиная с SQL-92 появилась возможность работать со значениями типа запись. Это позволяет за ключевым словом `VALUES` указать несколько наборов значений в круглых скобках (записей), которые необходимо вставить в таблицу. Например:

```
INSERT INTO Клиенты (Имя, Телефон, Сумма_заказа)
VALUES
('Петров', '444-4444', 25300),
('Иванов', '555-5555', 45100),
('Сидоров', '777-7777', 1200),
('Захаров', '123-9870', 7800);
```

- `INSERT INTO` *имяТаблицы* `(`*списокСтолбцов*`) SELECT ...` — вставляет в указанную таблицу записи, возвращаемые запросом на выборку.

На практике нередко требуется загрузить в одну таблицу данные из другой таблицы. Например, ранее созданная таблица `Контакты` содержит имена и другие данные о клиентах, которые необходимо вставить в таблицу `Клиенты`. Для этого в операторе `INSERT` вместо `VALUES` можно использовать оператор `SELECT`, выбирающий записи, необходимые для вставки.

Допустим, таблицы `Клиенты` и `Контакты` имеют одноименные и однотипные столбцы `Имя`, `Адрес` и `Телефон`. Требуется добавить все записи из таблицы `Контакты` в таблицу `Клиенты`. Это можно сделать с помощью следующего запроса:

```
INSERT INTO Клиенты (Имя, Адрес, Телефон)
SELECT Имя, Адрес, Телефон FROM Контакты;
```

Следующий запрос вставляет в таблицу `Клиенты` только те записи из таблицы `Контакты`, которых в ней еще нет. Таким образом, происходит дополнение первой таблицы данными из второй таблицы:

```
INSERT INTO Клиенты (Имя, Адрес, Телефон)
SELECT Имя, Адрес, Телефон FROM Контакты
WHERE Клиенты.Имя <> Контакты.Имя AND
    Клиенты.Адрес <> Контакты.Адрес AND
    Контакты.Имя IS NOT NULL;
```

Здесь в операторе WHERE применено сложное условие на случай, если в таблицах окажутся однофамильцы или в таблице Контакты окажутся неопределенные имена.

С помощью оператора INSERT можно добавить одну или несколько записей только в одну таблицу. Кроме того, необходимо учитывать следующие обстоятельства:

- таблица может иметь столбец типа SERIAL (счетчик), имеющий уникальные значения, которые СУБД назначает автоматически. Это обеспечивает уникальность всех записей таблицы и, следовательно, ее соответствие 1-ой нормальной форме. Поэтому в списке столбцов в операторе INSERT не следует указывать столбцы типа SERIAL;
- если список столбцов, указанный в операторе INSERT, содержит не все столбцы таблицы, то в оставшиеся столбцы (за исключением столбца типа SERIAL) будут введены значения NULL (неопределенное значение), а в столбец типа SERIAL правильное числовое значение введет СУБД;
- при введении значений в столбцы новой записи необходимо следить, чтобы типы значений соответствовали типам столбцов таблицы. При необходимости можно воспользоваться функцией преобразования типов CAST();
- таблица может иметь ограничения *(см. разд. 7.1)*. Если вводимые данные не удовлетворяют условиям этих ограничений, то запрос на добавление данных не будет выполнен. Так, например, если в ограничении на какой-либо столбец указано, что он не может иметь значения NULL, то попытка добавить пустую запись приведет к ошибке. В этом случае необходимо использовать оператор INSERT INTO с ключевым словом VALUES, чтобы сразу ввести значение, удовлетворяющее ограничениям.

6.2. Удаление записей

Для удаления записей из таблицы применяется оператор DELETE (удалить):

```
DELETE FROM имяТаблицы WHERE условие;
```

Данный оператор удаляет из указанной таблицы записи (а не отдельные значения столбцов), которые удовлетворяют указанному условию. Условие — это логическое выражение, различные конструкции которого были рассмотрены в предыдущих главах.

Следующий запрос удаляет записи из таблицы `Клиенты`, в которой значение столбца `Имя` равно `'Иванов'`:

```
DELETE FROM Клиенты WHERE Имя = 'Иванов';
```

Если таблица `Клиенты` содержит несколько записей, в которых есть клиент Иванов, то все они будут удалены.

В операторе `WHERE` может находиться подзапрос на выборку данных (оператор `SELECT`). Подзапросы в операторе `DELETE` работают точно так же, как и в операторе `SELECT`.

Пусть в базе данных имеется таблица `Продажи` (`ID_товара`, `Сумма_заказа`, `ID_клиента`), содержащая сведения о продажах товаров клиентам. Требуется удалить из этой таблицы записи о тех клиентах, которые приобрели товары на сумму, меньшую 1000. Для этого можно использовать такой запрос:

```
DELETE FROM Продажи T1
WHERE 1000 >=
    (SELECT SUM(T2.Сумма_заказа) FROM Продажи T2
        WHERE T1.ID_клиента= T2.ID_клиента);
```

Здесь запрос на удаление записей из таблицы `Продажи` содержит связанный (коррелированный) подзапрос, вычисляющий сумму значений столбца `Сумма_заказа`. Обратите внимание на использование двух различных псевдонимов для одной и той же таблицы. При выполнении данного запроса на удаление происходит, как и положено в случае связанных подзапросов, последовательный просмотр записей в таблице `Продажи`. Для каждой записи проверяется условие оператора `WHERE`, а именно выполняется подзапрос, вычисляющий сумму значений столбца `Сумма_заказа` для всех записей, в которых идентификатор клиента равен значению этого идентификатора в текущей записи (текущей называется запись, просматриваемая в данный момент). Если вычисленное значение не превышает 1000, то условие оператора `WHERE` выполняется и текущая запись удаляется, в противном случае запись

не удаляется. Далее происходит переход к следующей записи, рассматриваемой в качестве текущей, и описанные действия повторяются снова.

Теперь рассмотрим пример, в котором требуется удалить записи в одной таблице в соответствии с условием, обрабатывающим записи в другой таблице. Допустим, в базе данных имеются две таблицы:

- Продажи (ID_товара, Сумма_заказа, ID_клиента);
- Клиенты (ID_клиента, Имя, Адрес).

Требуется удалить из таблицы Клиенты те записи о клиентах, которые приобрели товары на сумму, меньшую 1000. Суммы заказов клиентов содержатся в таблице Продажи, а записи необходимо удалять из таблицы Клиенты, в которой нет данных о приобретенных товарах. Тем не менее, для решения этой задачи можно использовать запрос, аналогичный предыдущему:

```
DELETE FROM Клиенты T1
WHERE 1000 >=
    (SELECT SUM(T2.Сумма_заказа) FROM Продажи T2
        WHERE T1.ID_клиента= T2.ID_клиента);
```

Операция удаления записей из таблицы является опасной в том смысле, что связана с риском необратимых потерь данных в случае семантических (но не синтаксических) ошибок при формулировке SQL-выражения. Чтобы избежать неприятностей, перед удалением записей рекомендуется сначала выполнить соответствующий запрос на выборку, чтобы просмотреть, какие записи будут удалены. Так, например, перед выполнением рассмотренного ранее запроса на удаление не помешает выполнить соответствующий запрос на выборку:

```
SELECT * FROM Клиенты T1
WHERE 1000 >=
    (SELECT SUM(T2.Сумма_заказа) FROM Продажи T2
        WHERE T1.ID_клиента= T2.ID_клиента);
```

Для удаления всех записей из таблицы достаточно использовать оператор DELETE без ключевого слова WHERE. При этом сама таб-

лица со всеми определенными в ней столбцами остается и готова для вставки новых записей. Например:

```
DELETE FROM Клиенты;
```

Внимание

Таблица может иметь ограничения *(см. разд. 7.1)*, которые могут препятствовать удалению некоторых или всех записей.

6.3. Изменение данных

Для изменения значений столбцов таблицы применяется оператор `UPDATE` (изменить, обновить). Чтобы изменить значения в одном столбце таблицы в тех записях, которые удовлетворяют некоторому условию, следует выполнить такой запрос:

```
UPDATE имяТаблицы SET имяСтолбца = значение WHERE условие;
```

За ключевым словом `SET` (установить) следует выражение равенства, в левой части которого указывается имя столбца, а в правой — выражение, значение которого следует сделать значением данного столбца. Эти установки будут выполнены в тех записях, которые удовлетворяют условию в операторе `WHERE`.

Чтобы одним оператором `UPDATE` установить новые значения сразу для нескольких столбцов, вслед за ключевым словом `SET` записываются соответствующие выражения равенства, разделенные запятыми:

```
UPDATE имяТаблицы
SET имяСтолбца1 = значение1, имяСтолбца2 = значение2, ... ,
имяСтолбцаN = значениеN
WHERE условие;
```

Например, следующий запрос изменяет значения столбцов `Телефон` и `Сумма_заказа` в таблице `Клиенты` для тех записей, в которых столбец `Имя` имеет значение `'Иванов'`:

```
UPDATE Клиенты
SET Телефон = '333-1234', Сумма_заказа = 2570
WHERE Имя = 'Иванов';
```

Использование оператора WHERE в операторе UPDATE не обязательно. Если он отсутствует, то указанные в SET изменения будут произведены для всех записей таблицы.

Операция изменения записей, как и их удаление, связана с риском необратимых потерь данных в случае семантических ошибок при формулировке SQL-выражения. Например, стоит только забыть написать оператор WHERE, и будут обновлены значения во всех записях таблицы. Чтобы избежать подобных неприятностей, перед обновлением записей рекомендуется выполнить соответствующий запрос на выборку, чтобы просмотреть, какие записи будут изменены. Например, перед выполнением приведенного ранее запроса на обновление данных не помешает выполнить соответствующий запрос на выборку данных:

```
SELECT * FROM Клиенты
WHERE Имя = 'Иванов';
```

Внимание

Таблица может иметь ограничения *(см. разд. 7.1)*. Если устанавливаемые значения столбцов не удовлетворяют условиям этих ограничений, то запрос на обновление данных выполнен не будет.

Условие в операторе WHERE может содержать подзапросы, в том числе и связанные. Пусть в базе данных имеется таблица Продажи (ID_товара, Сумма_заказа, ID_клиента), содержащая сведения о продажах товаров клиентам. Предположим, что требуется сделать 5% скидку тем клиентам, которые приобрели товары на сумму, большую 1000. Для этого следует изменить значения столбца Сумма_заказа, просто умножить их на 0,95. Однако эти изменения должны быть выполнены, только если сумма значений этого поля для данного клиента превышает 1000. Таким образом, запрос на изменение данных должен содержать связанный подзапрос:

```
UPDATE Продажи T1
SET Сумма_заказа = Сумма_заказа*0.95
WHERE 1000 <
    (SELECT SUM(T2.Сумма_заказа) FROM Продажи T2
        WHERE T1.ID_клиента = T2.ID_клиента);
```

Здесь запрос на изменение данных из таблицы `Продажи` содержит связанный (коррелированный) подзапрос, вычисляющий сумму значений столбца `Сумма_заказа`. Обратите внимание на использование двух различных псевдонимов для одной и той же таблицы. При выполнении данного запроса происходит последовательный просмотр записей в таблице `Продажи`. Для каждой записи проверяется условие оператора `WHERE`, а именно выполняется подзапрос, вычисляющий сумму значений столбца `Сумма_заказа` для всех записей, в которых идентификатор клиента равен текущему значению этого идентификатора. Если вычисленное значение превышает 1000, то условие оператора `WHERE` выполняется и происходит изменение данных в соответствии с выражением `SET`, в противном случае изменения не вносятся. Далее происходит переход к следующей записи, рассматриваемой в качестве текущей, и описанные действия повторяются снова.

Нередко требуется обновить значения столбцов в зависимости от их текущих значений. В SQL:2003 для этого можно использовать оператор `CASE`, возвращающий значения *(см. разд. 4.4)*.

Допустим, имеется таблица `Клиенты` (`Имя`, `Адрес`, `Регион`, `Телефон`). Требуется изменить значения столбца `Регион` следующим образом: если значение столбца равно `'Северо-Запад'`, то его следует заменить на `'Санкт-Петербург'`; если значение равно `'Тверская область'`, то его следует заменить на `'Тверь'`; во всех остальных случаях нужно оставить прежние значения. Эту задачу можно решить с помощью такого запроса:

```
UPDATE Клиенты
SET Регион = CASE
        WHEN Регион = 'Северо-Запад' THEN 'Санкт-Петербург'
        WHEN Регион = 'Тверская область' THEN 'Тверь'
        ELSE Регион
    END;
```

Без оператора `CASE` данную задачу пришлось бы решать с помощью двух последовательных запросов:

```
UPDATE Клиенты
SET Регион = 'Санкт-Петербург' WHERE Регион = 'Северо-Запад';
```

```
UPDATE Клиенты
SET Регион = 'Тверь' WHERE Регион = 'Тверская область';
```

Допустим, что в таблице `Клиенты` было решено изменить значение `'Иркутск'` столбца `Регион`. Однако новое значение пока не определено и поэтому вместо `'Иркутск'` решили внести значение `NULL`. Так обычно поступают, чтобы обозначить тот факт, что значение не известно или еще не введено. В данном случае удобно применить запрос с использованием функции `NULLIF()` *(см. разд. 4.4.3)*:

```
UPDATE Клиенты
SET Регион = NULLIF(Регион, 'Иркутск');
```

При изменении значений в столбцах таблицы необходимо следить, чтобы типы значений соответствовали типам столбцов. При необходимости можно воспользоваться функцией преобразования типов `CAST()`.

Примечание

Некоторые базы данных (например, PostgreSQL) имеют расширение стандарта SQL, позволяющее обновлять одну таблицу данными из другой. Так, например, чтобы обновить таблицу `Клиенты` данными из таблицы `Продажи`, можно использовать такое выражение:

```
UPDATE Клиенты SET Сумма_заказа = Продажи.Сумма_заказа FROM
Продажи
WHERE ID_клиента = 5;
```

6.4. Проверка ссылочной целостности

Между таблицами в базе данных могут быть установлены связи. Эти связи задаются как ограничения ссылочной целостности данных. Что такое ссылочная целостность данных, было рассказано в *разд. 1.4.3*, а про установку связи между таблицами будет описано в *разд. 7.1*. При попытке добавить, изменить или удалить записи в связанных таблицах могут возникнуть так называемые аномалии модификации данных, обусловленные нарушением ссылочной целостности.

Целостность данных может быть нарушена при попытке добавить запись в дочернюю таблицу, для которой нет соответствующей записи в родительской таблице. Например, в базе данных имеется родительская таблица `Клиенты` (`ИмяКлиента`, `Имя`, `Адрес`, `Телефон`) и дочерняя таблица `Продажи` (`ID_заказа`, `Товар`, `Цена`, `Количество`, `ИмяКлиента`). В таблице `Клиенты` столбец `ИмяКлиента` является первичным ключом (имеет уникальные и определенные значения), а в таблице `Продажи` столбец `ИмяКлиента` не обязан иметь уникальные значения, поскольку один и тот же клиент может сделать несколько различных покупок. Чтобы эти две таблицы находились в связи, столбец `ИмяКлиента` в таблице `Продажи` должен быть внешним ключом, ссылающимся на первичный ключ `ИмяКлиента` в таблице `Клиенты`. О внешних ключах более подробно будет рассказано в *разд. 7.1.3*.

Если вы добавите в таблицу `Продажи` новую запись, содержащую значение столбца `ИмяКлиента`, которого еще нет в родительской таблице `Клиенты`, то возникнет аномалия модификации данных. Аналогичная ситуация произойдет, если вы попытаетесь удалить запись из таблицы `Клиенты`: в дочерней таблице `Продажи` некоторые записи могут ссылаться на клиентов, сведения о которых отсутствуют.

Поэтому вначале следует добавить запись в таблицу `Клиенты` с соответствующим значением столбца `ИмяКлиента`, а затем — в `Продажи` или, в случае с удалением, вначале нужно удалить соответствующие записи из таблицы `Продажи`, а затем — из `Клиенты`.

В ряде случаев перед добавлением или удалением записей можно проверить, не приведет ли это к нарушению ссылочной целостности. Такую проверку можно выполнить с помощью предиката `MATCH`.

Предикат `MATCH` имеет следующий синтаксис:

```
ЗначениеТипаЗаписи ROW MATCH [UNIQUE] [SIMPLE | PARTIAL |
FULL] (подзапрос);
```

Здесь в квадратных скобках указаны необязательные ключевые слова, а вертикальной чертой разделены их альтернативные варианты: `UNIQUE` (уникальный), `SIMPLE` (простой), `PARTIAL` (час-

тичный), FULL (полный). Эти ключевые слова определяют правила обработки значений типа записи с полями, имеющими неопределенные значения.

Допустим, требуется определить, есть ли в таблице Продажи запись ('Иванов', 'Компьютер'). Для этого можно выполнить такой запрос к базе данных:

```
SELECT * FROM Продажи
WHERE ('Иванов', 'Компьютер')
    MATCH (SELECT ИмяКлиента, Товар FROM Продажи);
```

Если в таблице Продажи есть запись с указанным именем клиента и товаром, то предикат MATCH вернет значение true.

Глава 7

Создание и модификация таблиц

Создание и модификация таблиц базы данных обычно производятся специальными диалоговыми инструментами сред разработки, которые поставляются вместе с СУБД. Хотя при их использовании, как правило, не требуется написания программ, эти средства включают в себя языки программирования, с помощью которых можно создавать сложные приложения, ориентированные на работу с базами данных. Примерами являются dBase, Paradox, Delphi, Access и др. Вместе с тем язык SQL также обладает средствами создания таблиц базы данных, т. е. тех таблиц, которые сохраняются в составе базы данных в долговременной памяти компьютера, например, на жестком диске. Напомню, что таблицы, возвращаемые запросами на выборку данных, являются виртуальными и доступны только тому, кто инициировал эти запросы.

В данной главе будут рассмотрены создание и модификация таблиц базы данных, временных таблиц, а также представлений.

7.1. Создание таблиц

Создание таблицы производится с помощью оператора `CREATE TABLE` (создать таблицу) с указанием необходимых параметров. При этом создается постоянная таблица. Чтобы удалить ее из базы данных, требуется выполнить специальный SQL-оператор.

Для создания временной таблицы используется оператор CREATE TEMPORARY TABLE (создать временную таблицу). Временная таблица, в отличие от постоянной, существует только в течение сеанса работы с базой данных, в котором она была создана. Однако временная таблица может быть доступна другим пользователям, как и постоянная таблица. Обычно временные таблицы создаются для представления в них текущих итоговых (отчетных) данных, доступных нескольким пользователям базы данных. Далее приведен синтаксис оператора CREATE:

```
CREATE [TEMPORARY] TABLE имяТаблицы (
    { Столбец1 тип [(размер)] [ограничение_столбца] [, ...] }
    {[ , CONSTRAINT ограничение_таблицы] [, ...]}
);
```

Здесь квадратные и фигурные скобки, в отличие от круглых, не являются элементами синтаксиса. В квадратных скобках заключены необязательные элементы, а в фигурных — элементы, которые могут повторяться.

Для создания таблицы необходимо указать ее имя и определить столбцы. Определение столбца включает его имя и тип. Если указывается длина столбца, то она заключается в круглые скобки после типа. Кроме того, можно указать ограничения для столбца. Все перечисленные элементы определения столбца указываются друг за другом через пробел. Если создаваемая таблица содержит несколько столбцов, то их определения разделяются запятыми. Ограничение может быть определено и для всей таблицы, а не только для ее столбцов. В этом случае используется ключевое слово CONSTRAINT (ограничение), после которого указывается само ограничение.

Только что созданная таблица пуста, т. е. не содержит ни одной записи. Для наполнения ее данными следует воспользоваться оператором INSERT INTO *(см. разд. 6.1)*.

Далее приведен запрос на создание простой таблицы без ограничений:

```
CREATE TABLE Студент (
    ID_студента    INTEGER,
```

```
    ФИО             CHAR(20),
    Специальность   CHAR(15),
    Примечание      VARCHAR
);
```

Данный запрос создает таблицу `Студент`, содержащую четыре столбца. Первый столбец целочисленный, а три других — символьные.

Ограничения устанавливаются на данные, которые вводятся в таблицу. Например, на числовой столбец можно наложить ограничение, заключающееся в том, что вводимое число должно находиться в некотором диапазоне. Другой пример ограничения: значение столбца не должно быть неопределенным.

Нередко ограничения устанавливаются и отслеживаются в приложениях, работающих с базами данных. Тем не менее, ограничения могут быть установлены и могут поддерживаться многими современными СУБД. Если вы создаете таблицу посредством SQL, то также имеете возможность задать эти ограничения, а выполнять их будет СУБД. При этом если одна и та же база данных используется несколькими приложениями, то вам придется установить ограничения только один раз, а не столько, сколько имеется приложений.

7.1.1. Ограничения для столбцов

В табл. 7.1 приведены основные ограничения для столбцов. Однако существуют и более сложные ограничения, которые в этой книге не рассматриваются.

Таблица 7.1. *Ограничения для столбцов*

Определение	Описание
`NOT NULL`	Столбец не может содержать значение `NULL`, т. е. значения этого столбца должны быть определенными
`UNIQUE`	Значение, вводимое в столбец, должно отличаться от всех остальных значений в этом столбце, т. е. быть уникальным

Таблица 7.1 *(окончание)*

Определение	Описание
`PRIMARY KEY`	Столбец является первичным ключом. В каждой таблице только один столбец может быть первичным ключом. Это означает, что он не может содержать значение `NULL`, а вводимое в него значение должно отличаться от всех остальных значений в этом столбце. Таким образом, `PRIMARY KEY` является комбинацией `NULL` и `UNIQUE`
`DEFAULT` *значение*	Устанавливает значение по умолчанию. Так, при добавлении новой записи столбец с таким ограничением автоматически получит указанное значение
`CHECK (`*условие*`)`	Позволяет производить проверку условия при вводе данных. Значение будет сохранено, если условие выполняется, в противном случае — нет

Примечание

Microsoft Access не поддерживает ключевые слова `DEFAULT` и `CHECK` в определениях ограничений.

Рассмотрим в качестве примера создание таблицы `Студент` (`ID_студента`, `ФИО`, `Специальность`, `Примечание`). Идентификатор студента (целочисленный столбец `ID_студента`) должен однозначно идентифицировать запись о студенте, т. е. иметь определенные и уникальные значения. Таким образом, данный столбец должен быть первичным ключом. От столбца `ФИО` (фамилия, имя и отчество) потребуем, чтобы в нем не было неопределенных значений. Целочисленный столбец `Специальность` должен содержать номера специальностей, которые не превышают 12. Тогда запрос на создание такой таблицы будет иметь вид:

```
CREATE TABLE Студент (
    ID_студента     INTEGER PRIMARY KEY,
    ФИО             CHAR(20) NOT NULL,
```

```
    Специальность   INTEGER CHECK(Специальность < 12),
    Примечание      VARCHAR
);
```

Предположим, что данный запрос выполнен. Тогда следующий запрос на добавление новой записи в таблицу `Студент` вызовет сообщение об ошибке и не будет выполнен:

```
INSERT INTO Студент;
```

Это произойдет потому, что данный запрос добавляет пустую запись, все столбцы которой содержат значение `NULL`, и, следовательно, не выполняются ограничения для первых двух столбцов. А следующий запрос, добавляющий в таблицу первую запись и устанавливающий определенные значения для столбцов, не вызовет проблем со стороны СУБД:

```
INSERT INTO Студент (ID_студента, ФИО, Специальность)
VALUES (1, 'Иванов Иван Иванович', 9);
```

Попытаемся вслед за данным запросом добавить еще одну запись:

```
INSERT INTO Студент (ID_студента, ФИО, Специальность)
VALUES (1, 'Петров Петр Петрович', 5);
```

Данный запрос не будет выполнен из-за нарушения ограничения, наложенного на первый столбец: его значения должны быть не только определенными (отличными от `NULL`), но и уникальными. Запись можно добавить, если столбцу `ID_студента` присвоить, например, значение 2.

Ограничения для столбцов можно выразить и иначе — через ограничения доменов. Напомню, что домен определяется как множество значений *(см. разд. 1.2.2)*. Домен в реляционной теории связывается с атрибутом отношения и, как таковой, определяет некоторое ограничение на этот атрибут (атрибут может принимать значения только из этого домена). В SQL можно создать домен, сначала не связанный ни с какими атрибутами (столбцами), определив для него допустимые значения: тип данных и дополнительные ограничения. Затем этот домен можно задать в качестве ограничений для любого столбца в любой таблице подобно тому, как задаются типы столбцов. Другими словами, домен можно связать с одним или несколькими столбцами различных

таблиц. В ряде случаев этот прием очень удобен, особенно если у вас имеется несколько "однотипных" столбцов в различных таблицах базы данных.

Чтобы создать домен, используется такой синтаксис:

```
CREATE DOMAIN имяДомена типДанных Ограничения;
```

Например, таблицу `Студент` (`ID_студента`, `ФИО`, `Специальность`, `Примечание`) можно определить традиционным образом:

```
CREATE TABLE Студент (
    ID_студента    INTEGER PRIMARY KEY,
    ФИО            CHAR(20) NOT NULL,
    Специальность  INTEGER CHECK (Специальность < 12),
    Примечание     VARCHAR
);
```

Однако можно сначала определить домен с именем `specDomain`, задав для него тип данных и ограничение:

```
CREATE DOMAIN specDomain INTEGER
    CHECK (Специальность < 12);
```

Обратите внимание, что в определении домена, пока не связанного ни с каким столбцом какой-либо таблицы, задается тип значений домена и ограничение на эти значения.

С учетом того, что домен `specDomain` создан, определение таблицы `Студент` можно задать следующим образом:

```
CREATE TABLE Студент (
    ID_студента    INTEGER PRIMARY KEY,
    ФИО            CHAR(20) NOT NULL,
    Специальность  specDomain,
    Примечание     VARCHAR
);
```

Рассмотрим еще один пример. Предположим, в нескольких таблицах вашей базы данных имеется столбец с именем `Код_продукта` и типом данных `CHAR(6)`, значения которого должны начинаться с символа `'A'`, `'C'` или `'X'`. Для таких столбцов, раз их несколько, можно предварительно создать общий домен:

```
CREATE DOMAIN ProdDomain CHAR (6)
```

```
    CHECK (SUBSTRING (VALUE, 1, 1) IN ('A', 'C', 'X'));
```

После определения домена можно создать таблицу, использующую этот домен:

```
CREATE TABLE Товары (
    Код_продукта  ProdDomain,
    Описание      VARCHAR
);
```

7.1.2. Ограничения для таблиц

Ограничения на вводимые данные можно назначить не только для отдельных столбцов, но и для таблицы в целом. Это удобно в тех случаях, когда необходимо для нескольких столбцов назначить одинаковые ограничения. Кроме того, если первичный ключ составной (т. е. состоит из нескольких столбцов), то указать его можно только как ограничение для таблицы, а не для столбца. Все определение ограничения для таблицы указывается после определений столбцов и состоит из ключевого слова `CONSTRAINT`, за которым следует выражение, определяющее непосредственно само ограничение. В табл. 7.2 приведены основные ограничения для таблицы.

Таблица 7.2. *Ограничения для таблицы*

Определение	Описание
`UNIQUE (`*списокСтолбцов*`)`	Значения в столбцах, указанных в списке, должны быть уникальными
`PRIMARY KEY (`*списокСтолбцов*`)`	В каждой таблице должен быть только один первичный ключ, который определен либо как ограничение для столбца, либо как ограничение для таблицы. Если первичный ключ составной, то он должен быть определен как ограничение для таблицы. Справа от ключевых слов `PRIMARY KEY` в круглых скобках указывается список столбцов, определяющих составной первичный ключ

***Таблица 7.2** (окончание)*

Определение	Описание
`CHECK (`*условие*`)`	Позволяет производить проверку условия при вводе данных. Значение будет сохранено, если условие выполняется, в противном случае — нет. В отличие от ограничения для столбца, здесь можно использовать условия, оперирующие значениями различных столбцов таблицы
`FOREIGN KEY ...` `REFERENCES ...`	Ограничение типа "внешний ключ" *(см. разд. 7.1.3)*

В следующем примере создается таблица `Студент` (`Фамилия`, `Имя`, `Отчество`, `Специальность`, `Примечание`). Предполагается, что комбинация значений первых трех столбцов должна однозначно идентифицировать запись в таблице, т. е. являться первичным ключом.

```
CREATE TABLE Студент (
    Фамилия         CHAR(20),
    Имя             CHAR(15),
    Отчество        CHAR(20),
    Специальность   INTEGER,
    Примечание      VARCHAR,
    CONSTRAINT PRIMARY KEY (Фамилия, Имя, Отчество)
);
```

Следующие два запроса на добавление записей будут выполнены, поскольку комбинации значений столбцов, определяющих первичный ключ, не содержат `NULL` и отличаются друг от друга:

```
INSERT INTO Студент (Фамилия, Имя, Отчество, Специальность)
VALUES ('Петров', 'Петр', 'Петрович', 5);
INSERT INTO Студент (Фамилия, Имя, Отчество, Специальность)
VALUES ('Петров', 'Петр', 'Иванович', 5);
```

7.1.3. Внешние ключи

Одна из важнейших разновидностей ограничений связана с определением внешних ключей. Внешний ключ — это столбец или группа столбцов, соответствующих первичному ключу другой таблицы. Чтобы понять синтаксис выражения, определяющего внешний ключ, рассмотрим пример.

Пусть в базе данных имеются две таблицы:

- `Заказы` (`ID_заказа`, `ID_клиента`) — содержит сведения о том, какие заказы сделал тот или иной клиент;
- `Клиенты` (`ID_клиента`, `Имя`, `Адрес`, `Телефон`) — сведения о клиентах (справочник).

В таблице `Клиенты` столбец `ID_клиента` является первичным ключом, т. е. его значения отличны от `NULL` и уникальны. В таблице `Заказы` столбец `ID_клиента` не обязан иметь уникальные значения, поскольку один и тот же клиент может сделать несколько заказов. Вместе с тем любому значению столбца `Заказы.ID_клиента` соответствует единственное значение столбца `Клиенты.ID_клиента`. При описанных условиях столбец `ID_клиента` таблицы `Заказы` является внешним ключом, ссылающимся на первичный ключ `ID_клиента` таблицы `Клиенты`.

Внешний ключ определяется как ограничение для таблицы в выражении с ключевыми словами `CONSTRAIN FOREIGN KEY` (ограничение "внешний ключ"):

```
CONSTRAINT FOREIGN KEY внешнийКлюч REFERENCES
внешняяТаблица (первичныйКлюч)
```

Здесь *внешнийКлюч* — имя столбца или список столбцов, разделенных запятыми, которые определяют внешний ключ, за ключевым словом `REFERENCES` (ссылки) указывается внешняя таблица и ее первичный ключ, на который ссылается внешний ключ.

Для рассмотренного ранее примера таблицу `Заказы` можно определить следующим образом:

```
CREATE TABLE Заказы (
    ID_заказа INTEGER,
```

```
    ID_клиента INTEGER,
    CONSTRAINT FOREIGN KEY ID_клиента REFERENCES Клиенты
(ID_клиента)
);
```

Использование внешних ключей обеспечивает сохранение ссылочной целостности базы данных при изменении и удалении записей. Если бы таблицы `Заказы` и `Клиенты` не были связаны, то при удалении записи из таблицы `Клиенты` в таблице `Заказы` могли остаться ссылки на клиента, о котором уже нет сведений. Этот факт обычно расценивается как аномалия удаления. В случае определения в таблице `Заказы` внешнего ключа `ID_клиента` из таблицы `Клиенты` не удастся удалить клиента, если он сделал хотя бы один заказ. Если требуется удалить из базы данных все, что касается определенного клиента, то сначала удаляются записи в таблице `Заказы`, а потом — в таблице `Клиенты`.

Аналогичная ситуация может произойти и при обновлении данных. Например, в таблице `Клиенты` вы изменили идентификатор клиента, имеющего заказы, и, таким образом, внешнему ключу теперь не на что ссылаться. Это аномалия изменения. В данном случае необходимо сначала добавить новую запись в таблицу `Клиенты`, указав в ней необходимое значение `ID_клиента`, затем изменить в таблице `Заказы` все старые значения `ID_клиента` на то, которое вы только что ввели в новой записи таблицы `Клиенты`, а затем удалить из таблицы `Клиенты` запись со старым идентификатором клиента.

Чтобы в таблицах, связанных внешним ключом, не делать модификацию данных в несколько этапов, в выражении `CONSTRAIN FOREIGN KEY` можно использовать дополнительные ключевые слова:

- `ON DELETE CASCADE | SET NULL` (при удалении каскадировать | установить `NULL`);
- `ON UPDATE CASCADE | SET NULL` (при обновлении каскадировать | установить `NULL`).

Здесь вертикальная черта не является элементом синтаксиса, а лишь разделяет возможные варианты ключевых слов.

Так, при использовании `ON DELETE CASCADE` в случае удаления записи со значением первичного ключа, которое имеется во внешнем ключе другой таблицы, соответствующие записи удаляются автоматически из двух таблиц. Например, при удалении из таблицы `Клиенты` записи о клиенте, имеющем заказы, в таблице `Заказы` также будут удалены все записи, ссылающиеся на данного клиента. Чтобы данная стратегия выполнялась, таблица `Заказы` должна быть определена следующим образом:

```
CREATE TABLE Заказы (
    ID_заказа   INTEGER,
    ID_клиента  INTEGER,
    CONSTRAINT FOREIGN KEY ID_клиента REFERENCES Клиенты
(ID_клиента)
    ON DELETE CASCADE
);
```

Однако на практике обычно предпочитают сначала убедиться, что клиент не имеет заказов, и лишь затем вычеркнуть его из справочника.

Вариант `SET NULL` обычно используется при обновлении данных. Например:

```
CREATE TABLE Заказы (
    ID_заказа INTEGER,
    ID_клиента INTEGER,
    CONSTRAINT FOREIGN KEY ID_клиента REFERENCES Клиенты
(ID_клиента)
    ON UPDATE SET NULL
);
```

В данном случае при изменении (в том числе и при удалении) в таблице `Клиенты` записи, на которую ссылается внешний ключ таблицы `Заказы`, значения внешнего ключа устанавливаются в `NULL`. Однако этот вариант не сработает, если на столбец `Заказы.ID_клиента` наложено ограничение `NOT NULL`. Обычно так и бывает, поскольку при оформлении заказа клиент должен быть обязательно указан. Поэтому, на всякий случай, лучше использовать ключевые слова `ON UPDATE CASCADE`.

Примечание

Некоторые СУБД (например, PostgreSQL) допускают комбинирование дополнительных ключевых слов. Например:

```
ON UPDATE SET NULL ON DELETE CASCADE
```

7.2. Удаление таблиц

Удалить таблицу из базы данных можно следующим образом:

```
DROP TABLE имяТаблицы;
```

Разумеется, при удалении таблицы теряются и все содержащиеся в ней данные. Во время работы с базой данных нередко создаются таблицы для временного хранения данных, полученных на каком-то промежуточном этапе. Рано или поздно такие таблицы подлежат удалению. Однако можно забыть это сделать. Кроме того, промежуточные таблицы, создаваемые приложениями, могут остаться в базе данных из-за сбоев. Поэтому для создания временных таблиц лучше использовать оператор `CREATE TEMPORARY TABLE` *(см. разд. 7.1)*, а не `CREATE TABLE`. Напомню, что временная таблица, созданная с помощью `CREATE TEMPORARY TABLE`, автоматически уничтожается по окончании сеанса работы с базой данных.

7.3. Модификация таблиц

Разработка базы данных обычно происходит итеративно. Редко когда удается сразу оптимально определить структуру входящих в нее таблиц, чтобы потом их не переделывать. Кроме того, уже готовая база данных со временем может пополняться новыми таблицами, которые связываются с уже имеющимися. А установка новых связей требует коррекции параметров в старых таблицах. Модификация таблицы — это изменение ее структуры, т. е. добавление, удаление и переименование столбцов, а также изменение их типов и размеров.

Язык SQL обладает специальными средствами модификации таблиц. Если бы их не было или если отказаться их применять,

то изменение структуры таблицы можно выполнить следующим образом:

1. Создать новую рабочую таблицу с необходимой структурой.
2. С помощью оператора `INSERT INTO` вставить в рабочую таблицу данные из исходной таблицы.
3. Удалить исходную таблицу.
4. Переименовать рабочую таблицу, присвоив ей имя исходной.

Все предыдущие операции можно выполнить посредством рассмотренных ранее операторов SQL. Переименование таблицы можно сделать средствами программной среды разработки базы данных. Если же ограничиться только операторами SQL, то вместо переименования придется выполнить следующее:

1. Создать новую таблицу с именем исходной и с такой же структурой, как у рабочей таблицы.
2. С помощью оператора `INSERT INTO` вставить в новую таблицу данные из рабочей таблицы.
3. Удалить рабочую таблицу.

Как видите, изменение структуры таблицы обычными операторами SQL довольно трудоемко. Значительно проще изменить структуру таблицы с помощью оператора `ALTER TABLE` (изменить таблицу). С помощью дополнительных ключевых слов можно выполнить следующие операции:

- `ADD COLUMN` — добавить столбец;
- `DROP COLUMN` — удалить столбец;
- `ALTER COLUMN` — изменить тип, размер и ограничение столбца;
- `RENAME COLUMN` — переименовать столбец;
- `RENAME TO` — переименовать таблицу.

Типичной задачей изменения структуры таблицы является добавление столбца. Это можно сделать с помощью SQL-выражения с таким синтаксисом:

```
ALTER TABLE имяТаблицы ADD COLUMN имяСтолбца тип(размер)
```

Например:

```
ALTER TABLE Студенты
ADD COLUMN Адрес CHAR(25);
```

Добавленный столбец оказывается последним в таблице, т. е. занимает крайнюю правую позицию. Иногда это неудобно. Пусть, например, столбец `Имя` в таблице занимает первую позицию, а ранее забытый и позднее добавленный столбец `Фамилия` — десятую позицию. Чтобы выборка данных из этой таблицы выглядела привычным образом, приходится специально указывать необходимое расположение столбцов в операторе `SELECT`, например, `SELECT Имя, Фамилия FROM ...`

Выражение `ADD COLUMN` может использоваться в выражении `ALTER TABLE` несколько раз — для каждого отдельного столбца. При определении параметров добавляемого столбца можно указать ограничение для него. Следующее SQL-выражение добавляет в таблицу `Студенты` столбец `ID_ст` и объявляет его первичным ключом:

```
ALTER TABLE Студенты
ADD COLUMN ID_ст INTEGER PRIMARY KEY;
```

Не следует забывать, что первичный ключ в таблице может быть только один. Если в таблице уже есть первичный ключ, то добавление столбца как первичного ключа не будет выполнено. Если вам требуется переназначить первичный ключ, то сначала необходимо скорректировать соответствующим образом параметры уже имеющегося столбца, объявленного как первичный ключ, а затем добавить новый столбец с параметром `PRIMARY KEY`. Подробнее об ограничениях было рассказано в *разд. 7.1*. Коррекцию столбца можно выполнить так: либо сначала удалить его, а затем добавить новый столбец с требуемыми параметрами либо изменить параметры существующего столбца с помощью оператора `ALTER COLUMN`.

Чтобы удалить столбец из таблицы, достаточно выполнить SQL-выражение со следующим синтаксисом:

```
ALTER TABLE имяТаблицы DROP COLUMN имяСтолбца;
```

При удалении столбца удаляются все содержащиеся в нем данные. Если удаляемый столбец является первичным ключом, на который ссылается внешний ключ из другой таблицы, то удаление не будет выполнено. В этом случае вам придется сначала скорректировать данные в другой таблице.

Например, следующее выражение удаляет из таблицы `Студенты` столбец `Примечание`:

```
ALTER TABLE Студенты
DROP COLUMN Примечание;
```

Для изменения параметров существующего столбца, таких как тип, размер и ограничение, применяются ключевые слова `ALTER COLUMN`:

```
ALTER TABLE имяТаблицы ALTER COLUMN имяСтолбца тип(размер)
[ограничение];
```

Здесь квадратные скобки указывают, что заключенное в них выражение не является обязательным.

В следующем примере в таблице `Студент` уже существующий столбец `ФИО` приобретает новые параметры. А именно будучи символьным, он получает увеличение длины до 50 символов и становится первичным ключом:

```
ALTER TABLE Студенты
ALTER COLUMN ФИО CHAR(50) PRIMARY KEY;
```

Разумеется, если в таблице уже имеется первичный ключ (например, столбец `ID_студента`), то сделать первичным ключом еще один столбец не удастся. Кроме того, не следует забывать, что при преобразовании типов могут быть потеряны данные. Так, если вы преобразуете столбец символьного типа, содержащий фамилии или адреса, в числовой тип, то все данные будут потеряны. При уменьшении размера символьного столбца его значения могут оказаться обрезанными справа. Таким образом, следует очень внимательно изменять параметры столбцов, содержащих некоторые данные.

Переименовывать столбцы приходится редко, поскольку в выборках и представлениях всегда можно присвоить им нужные

псевдонимы. Тем не менее, переименовать столбцы и таблицу можно с помощью оператора `ALTER TABLE` с ключевыми словами `RENAME COLUMN ... TO` (переименовать столбец ... в).

Для переименования столбца используется следующий синтаксис:

```
ALTER TABLE имяТаблицы RENAME COLUMN имяСтолбца
TO новоеИмяСтолбца;
```

При переименовании столбца его прежний тип, размер и ограничение сохраняются.

Для переименования таблицы используется похожий синтаксис:

```
ALTER TABLE имяТаблицы RENAME TO новоеИмяТаблицы;
```

Изменение структуры таблицы является небезопасной операцией с точки зрения возможности потери данных. Если вы не вполне уверены, что получите требуемый результат, то лучше использовать резервное копирование данных, поступая следующим образом:

1. Создайте временную таблицу с помощью оператора `CREATE TEMPORARY TABLE`.
2. Скопируйте во временную таблицу данные из таблицы, структуру которой вы собираетесь модифицировать. Это можно сделать с помощью оператора `INSERT INTO`.
3. Далее можно поступить двумя способами:
   - изменить структуру исходной таблицы так, как вам требуется. Этот способ лучше применять при коррекции структуры, не связанной с изменениями существующих ограничений на столбцы и/или на таблицу в целом;
   - удалить исходную таблицу и создать новую под тем же именем и с требуемой структурой. Я предпочитаю использовать этот способ, если требуется изменить ограничения.
4. Скопируйте данные из временной таблицы в таблицу с новой структурой, используя оператор `INSERT INTO`. Больше временная таблица не нужна, она будет автоматически удалена по окончании сеанса работы с базой данных.

Рассмотрим пример. Пусть имеется таблица `Студенты` (`ID_студента`, `ФИО`, `Специальность`), в которой первичным ключом является `ФИО`.

Требуется изменить первичный ключ, а именно сделать первичным ключом столбец `ID_студента`. Кроме того, необходимо в столбце `ФИО` запретить неопределенные значения, а также добавить столбец `Примечание` неограниченной длины. Чтобы решить эту задачу, можно выполнить следующую последовательность запросов:

```
CREATE TEMPORARY TABLE stud_tmp (
    ID_студента INTEGER,
    ФИО CHAR(20),
    Специальность CHAR(15)
);
INSERT INTO stud_tmp
    SELECT * FROM Студенты;
DROP TABLE Студенты;
CREATE TABLE Студенты (
    ID_студента INTEGER PRIMARY KEY,
    ФИО CHAR(20) NOT NULL,
    Специальность CHAR(15),
    Примечание VARCHAR
);
INSERT INTO Студенты
    SELECT * FROM stud_tmp;
```

7.4. Представления

7.4.1. Что такое представление

База данных состоит из таблиц — объектов, содержащих данные и существующих в долговременной памяти компьютера, например, на жестком диске. Это могут быть отдельные файлы или части одного файла (как, например в Microsoft Access). С помощью запросов на выборку (SQL-выражений, начинающихся с ключевого слова `SELECT`) создаются виртуальные таблицы, являющиеся временными и доступными только тому, кто выполнил данный запрос.

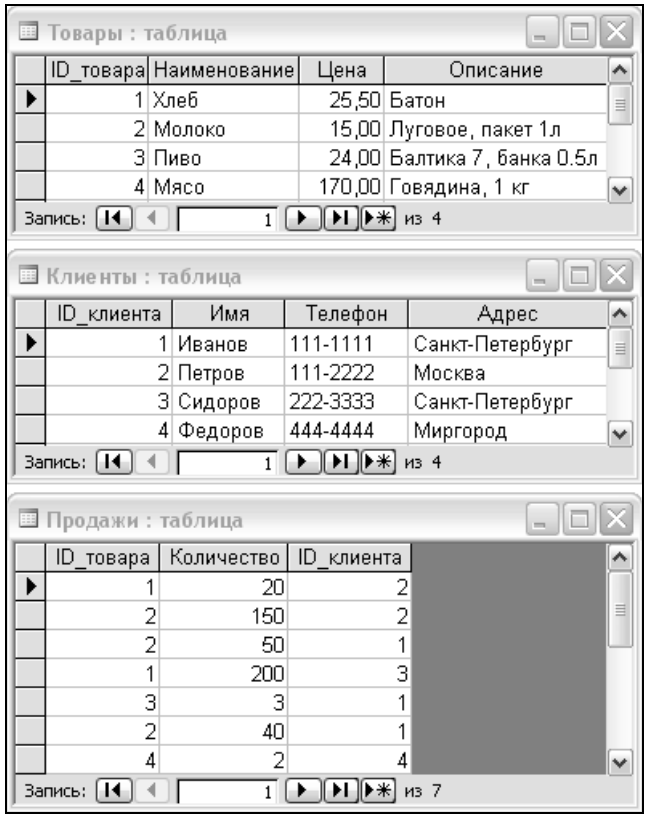
Представления — это тоже виртуальные таблицы, но они могут быть доступны многим пользователям и существуют в базе данных до тех пор, пока не будут принудительно удалены. Они во всем похожи на обычные таблицы базы данных, за исключением того, что не являются физическими объектами хранения данных. Данные в представлениях, подобно в ответах на запрос `SELECT`, просто выбираются из таблиц базы данных, т. е. представляются в том или ином виде. Представления создаются с помощью оператора `CREATE VIEW` (создать вид, представление), но в действительности за ними стоят скрытые SQL-запросы. Эти запросы или параметры представления хранятся в разделе метаданных базы. Работать с представлением можно как с обычной таблицей. Однако любой запрос к представлению в действительности инициирует скрытый запрос, который комбинируется с пользовательским.

Зачем нужны представления? Обычно они применяются, чтобы скрыть от пользователя некоторые столбцы, скомбинировать из нескольких таблиц одну, которая часто нужна пользователю, а запрос для ее получения довольно сложен. Так, например, база данных может содержать ряд служебных таблиц, с помощью которых она приобретает полезные свойства. Смысл этих таблиц может быть неясен пользователю, но для получения требуемых данных они должны использоваться в запросах. Это обстоятельство может существенно затруднить формирование SQL-выражения запроса. Таким образом, представления используются как надстроечные средства для адаптации базы данных к различным категориям пользователей.

Рассмотрим в качестве примера простую базу данных, состоящую из трех таблиц:

- `Товары` (`ID_товара`, `Наименование`, `Цена`, `Описание`);
- `Клиенты` (`ID_клиента`, `Имя`, `Адрес`, `Телефон`);
- `Продажи` (`ID_товара`, `Количество`, `ID_клиента`).

Примеры этих таблиц показаны на рис. 7.1. С точки зрения организации хранения данных такая база неплохо спроектирована.



Товары : таблица

ID_товара	Наименование	Цена	Описание
1	Хлеб	25,50	Батон
2	Молоко	15,00	Луговое, пакет 1л
3	Пиво	24,00	Балтика 7, банка 0.5л
4	Мясо	170,00	Говядина, 1 кг

Запись: 1 из 4

Клиенты : таблица

ID_клиента	Имя	Телефон	Адрес
1	Иванов	111-1111	Санкт-Петербург
2	Петров	111-2222	Москва
3	Сидоров	222-3333	Санкт-Петербург
4	Федоров	444-4444	Миргород

Запись: 1 из 4

Продажи : таблица

ID_товара	Количество	ID_клиента
1	20	2
2	150	2
2	50	1
1	200	3
3	3	1
2	40	1
4	2	4

Запись: 1 из 7

Рис. 7.1. Исходные таблицы базы данных

Однако менеджера по продажам могут не интересовать идентификаторы товаров и клиентов. Скорее всего, ему нужна будет только одна таблица, например, `Обзор_продаж` (`Товар`, `Количество`, `Цена`, `Стоимость`, `Клиент`). Разумеется, эту таблицу можно получить из исходных трех таблиц с помощью следующего запроса:

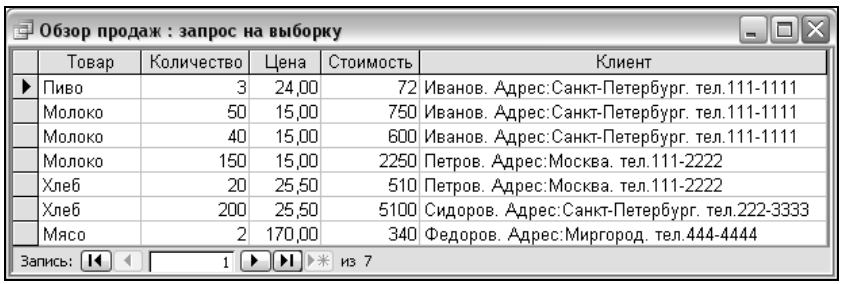
```
SELECT Товары.Наименование AS Товар, Продажи.Количество,
    Товары.Цена, Продажи.Количество*Товары.Цена AS Стоимость,
    Клиенты.Имя || '. Адрес:' || Клиенты.Адрес || '. тел.'
|| Клиенты.Телефон AS Клиент
FROM Продажи, Товары, Клиенты
```

```
WHERE Продажи.ID_Клиента = Клиенты.ID_клиента
    AND
    Продажи.ID_товара = Товары.ID_товара;
```

Примечание

Операция склейки строк в Microsoft Access обозначается символом (+), а не (||).

Рассмотренный запрос выполняет естественное соединение трех таблиц и добавляет столбцы `Стоимость` и `Клиент`, значения которых вычисляются на основе имеющихся столбцов. На рис. 7.2 показана таблица, возвращаемая рассмотренным запросом.



Обзор продаж : запрос на выборку

Товар	Количество	Цена	Стоимость	Клиент
Пиво	3	24,00	72	Иванов. Адрес:Санкт-Петербург. тел.111-1111
Молоко	50	15,00	750	Иванов. Адрес:Санкт-Петербург. тел.111-1111
Молоко	40	15,00	600	Иванов. Адрес:Санкт-Петербург. тел.111-1111
Молоко	150	15,00	2250	Петров. Адрес:Москва. тел.111-2222
Хлеб	20	25,50	510	Петров. Адрес:Москва. тел.111-2222
Хлеб	200	25,50	5100	Сидоров. Адрес:Санкт-Петербург. тел.222-3333
Мясо	2	170,00	340	Федоров. Адрес:Миргород. тел.444-4444

Запись: 1 из 7

Рис. 7.2. Результат запроса с естественным соединением трех таблиц

Если бы таблица, представленная на рис. 7.2, была не результатом запроса типа `SELECT`, а представлением, то можно было бы обращаться к ней с запросами, как к обычной таблице базы данных. Более того, в большинстве случаев можно было бы забыть о том, что база данных состоит из трех таблиц, и работать только с одним представлением.

7.4.2. Создание представлений

Оператор создания представления имеет следующий вид:

```
CREATE VIEW имяПредставления AS запросSELECT;
```

Как и обычная таблица базы данных, представлению присваивается имя, которое не должно совпадать ни с одним именем таб-

лиц. За ключевым словом `AS` следует SQL-выражение запроса на выборку данных.

Рассмотрим в качестве примера простую базу данных, состоящую из трех таблиц:

- `Товары` (`ID_товара`, `Наименование`, `Цена`, `Описание`);
- `Клиенты` (`ID_клиента`, `Имя`, `Адрес`, `Телефон`);
- `Продажи` (`ID_товара`, `Количество`, `ID_клиента`).

Чтобы создать представление, показанное на рис. 7.2, достаточно выполнить следующее SQL-выражение:

```
CREATE VIEW Обзор_продаж AS
   SELECT Товары.Наименование AS Товар, Продажи.Количество,
     Товары.Цена,
     Продажи.Количество*Товары.Цена AS Стоимость,
     Клиенты.Имя || '. Адрес:' || Клиенты.Адрес || '. тел.'
|| Клиенты.Телефон AS Клиент
   FROM Продажи, Товары, Клиенты
   WHERE Продажи.ID_Клиента = Клиенты.ID_клиента
     AND
     Продажи.ID_товара = Товары.ID_товара;
```

В результате будет создана виртуальная таблица `Обзор_продаж`, к которой можно обращаться с запросами, как к обычной таблице базы данных. Так, например, следующий запрос выбирает из представления записи, в которых столбец `Товар` имеет значение `'Молоко'`:

```
SELECT * FROM Обзор_продаж
WHERE Товар = 'Молоко';
```

Обратите внимание, что условие выборки в запросе оперирует со столбцом, который существует только в представлении (его нет ни в одной из исходных таблиц базы данных).

Рассмотренное представление является многотабличным, поскольку создано на основе не одной, а трех таблиц базы данных. На практике часто используются более простые однотабличные представления, в которых скрываются некоторые столбцы и/или

добавляются столбцы, значения которых вычисляются. Например, чтобы создать представление таблицы Клиенты, в котором нет столбца ID_клиента, достаточно выполнить следующее выражение:

```
CREATE VIEW Список_клиентов AS
SELECT Имя, Телефон, Адрес FROM Клиенты;
```

Очевидно, что набор данных, содержащихся в этом представлении, можно получить и с помощью запроса:

```
SELECT Имя, Телефон, Адрес FROM Клиенты;
```

Однако относительно полученного набора данных вы не сможете задать какой-либо запрос, поскольку этот набор данных является виртуальной таблицей, отличной от представления.

Оператор CREATE VIEW допускает еще и такую форму синтаксиса:

```
CREATE VIEW имяПредставления (столбец1, столбец2, ... ,
столбецN) AS запросSELECT;
```

Здесь необязательный список столбцов содержит имена столбцов представления, которые могут отличаться от имен столбцов оператора SELECT. Данная форма позволяет явно установить соответствие между столбцами представления и столбцами виртуальной таблицы, возвращаемой оператором SELECT. Рассмотренный ранее запрос на создание представления Список_клиентов можно сформулировать другим образом:

```
CREATE VIEW Список_клиентов (ФИО, Тел, Адрес) AS
SELECT Имя, Телефон, Адрес FROM Клиенты;
```

В принципе, вы можете представить всю базу данных с помощью одного или нескольких представлений, скрыв исходные таблицы. Действительно, если база данных состоит более чем из 5—7 связанных таблиц, формулирование запросов становится непростым делом: необходимо помнить, в каких таблицах какие столбцы находятся.

Некоторые разработчики приложений считают, что создавать представления следует для как можно большего числа таблиц. В ряде случаев это хороший прием. Однако не следует забывать, что если представление построено на основе многих таблиц

с использованием сложного SQL-выражения, работа с ним может оказаться существенно менее эффективной, чем работа непосредственно с таблицами базы данных. Например, если пользователю нужен всего лишь один столбец, а он вынужден работать с представлением из многих таблиц, то будет выполнено все сложное SQL-выражение, создающее это представление и выделяющее в нем требуемый столбец. Таким образом, представления следует применять, исходя из компромисса между удобством пользователя и производительностью системы.

Представления удаляются из базы данных так же, как и таблицы:

```
DROP VIEW имяПредставления;
```

В отличие от обычных таблиц при удалении представления данные сохраняются без изменений. Напомню еще раз, что представление не является объектом хранения данных.

7.4.3. Изменение данных в представлениях

Некоторые современные СУБД поддерживают операции изменения содержимого представлений — вставку, удаление и изменение записей. Однако при этом могут возникнуть проблемы. Дело в том, что при получении запроса на модификацию данных в представлении СУБД должна выполнить соответствующие операции над таблицами базы данных, на основе которых создано это представление. Напомню, что представление лишь вызывает и представляет, но не хранит данные. При этом могут возникнуть неоднозначные ситуации.

- Если представление многотабличное, то как узнать, какие таблицы следует модифицировать?
- Если в определении представления используются выражения, относящиеся к оператору `SELECT`, то как использовать эти выражения для изменения представления?

Рассмотрим представление `Товары_view` таблицы `Товары` (`ID_товара`, `Наименование`, `Цена`, `Описание`), в котором значения столбца

Цена увеличены на 50. Это представление создается с помощью следующего выражения:

```
CREATE VIEW Товары_view AS
    SELECT Наименование, Цена + 50 AS Новая_цена
    FROM Товары;
```

Попытку обновить в этом представлении значение столбца Новая_цена можно выполнить с помощью выражения:

```
UPDATE Товары_view SET Новая_цена = Новая_цена*2;
```

Однако эта попытка не удастся, поскольку в таблице Товары нет столбца Новая_цена. А раз таблица базы данных не будет обновлена, то не будет обновлено и представление.

Внимание

Если столбец в представлении не соответствует столбцу таблицы базы данных, то обновить этот столбец нельзя.

Допустим, база данных состоит из трех таблиц:

- Товары (ID_товара, Наименование, Цена, Описание);
- Клиенты (ID_клиента, Имя, Адрес, Телефон);
- Продажи (ID_товара, Количество, ID_клиента).

Создадим на основе этих таблиц представление Обзор_продаж:

```
CREATE VIEW Обзор_продаж (Наименование, Количество, Цена,
Клиент) AS
    SELECT Товары.Наименование, Продажи.Количество,
    Товары.Цена, Клиенты.Имя AS Клиент
    FROM Продажи, Товары, Клиенты
    WHERE Продажи.ID_Клиента = Клиенты.ID_клиента
        AND
        Продажи.ID_товара = Товары.ID_товара;
```

Содержимое данного представления можно изменять (если СУБД поддерживает изменение представлений), поскольку имеется однозначное соответствие между столбцами представления и столбцами таблиц базы данных.

7.5. Задачи

Задача 7.1

Пусть база данных содержит следующие три таблицы:

- `Студенты` (`ID_студента`, `Имя`, `Телефон`, `Адрес`, `ID_курса`) — данные о студентах, в том числе и о курсах, которые они изучают;
- `Курсы` (`ID_курса`, `Наименование`) — данные о курсах;
- `Преподаватели` (`ID_преподавателя`, `Имя`, `Телефон`, `Адрес`, `ID_курса`) — данные о преподавателях, в том числе и о курсах, занятия по которым они проводят.

В таблице `Студенты` каждый студент может быть записан на несколько курсов. Аналогично, один и тот же преподаватель может проводить занятия по нескольким курсам. Таким образом, столбцы `ID_курса` в этих таблицах не обязаны содержать уникальные значения. В таблице `Курсы`, напротив, столбец `ID_курса` содержит только уникальные значения.

Сформулируйте SQL-выражения для создания указанных таблиц так, чтобы таблицы `Студенты` и `Преподаватели` были связаны с таблицей `Курсы`.

Задача 7.2

Для базы данных, описанной в задаче 7.1, сформулируйте запрос, возвращающий таблицу, которая содержит:

- наименования курсов и имена преподавателей, которые проводят занятия по данным курсам;
- наименования курсов и имена студентов, которые изучают данные курсы;
- имена студентов и имена преподавателей, которые проводят занятия с данными студентами.

Задача 7.3

Для базы данных, описанной в задаче 7.1, сформулируйте запрос, возвращающий представление, которое содержит наименования курсов, преподавателей и студентов. Каждая запись в представлении должна содержать наименование некоторого курса, имя преподавателя, который проводит занятия по данному курсу, и имя студента, который его изучает.

Задача 7.4

Пусть в базе данных, описанной в задаче 7.1, имеется таблица `Контакты` (`Имя`, `Адрес`, `Телефон`, `Примечания`). Сформулируйте запрос, добавляющий в нее соответствующие данные из таблиц `Студенты` и `Преподаватели`. Решите эту же задачу, записывая при этом в столбец `Примечания` слово `'Студент'` или `'Преподаватель'`, в зависимости от того, из какой таблицы берутся данные для вставки в таблицу `Контакты`.

Глава 8

Транзакции

При работе с базой данных всегда следует иметь в виду, что существует хотя и малая, но отличная от нуля вероятность испортить или потерять данные. Эта вероятность тем больше, чем больше база данных и сложнее запросы к ней. В СУБД имеются специальные настраиваемые средства защиты данных, однако в ряде случаев можно применить дополнительные меры, предоставляемые SQL.

Предположим, что требуется перевести деньги с одного счета на другой. Для этого необходимо выполнить несколько операторов SQL. Если в ходе их выполнения произойдет какой-нибудь аппаратный или программный сбой, из-за которого не выполнится один или несколько операторов, то может произойти так, что деньги будут сняты с одного счета, но не поступят на другой. Чтобы этого не произошло, предпринимаются специальные меры. Их суть состоит в том, что несколько SQL-операторов определяются как единый блок, называемый *транзакцией* (от англ. *transaction* — дело, сделка). В транзакции либо все операторы выполняются успешно, либо ни один из них не выполняется, а база данных возвращается в исходное состояние, в котором она находилась до начала выполнения операторов транзакции. Другими словами, если какой-то оператор в транзакции по какой-нибудь причине не выполнился, то отменяется действие всех уже выполненных операторов в этой транзакции — происходит так называемый *откат*.

Неприятности могут возникнуть и при работе с базой данных в многопользовательском режиме, даже в условиях абсолютной надежности оборудования, СУБД и программ приложения. Если несколько пользователей пытаются одновременно использовать одну и ту же таблицу, то может возникнуть так называемая *коллизия одновременного доступа*. Представьте себе ситуацию с покупкой железнодорожного билета. Первый покупатель запрашивает билет на некоторый поезд в купейном вагоне на определенную дату. Кассир выполняет соответствующий запрос к базе данных и получает сведения о всех свободных местах. Пока покупатель выбирает подходящее ему место в вагоне, другой покупатель (подойдя к другой кассе) может оказаться более проворным и купить билет на тот же поезд, вагон и дату. Но первый покупатель, как и его кассир, еще не знает этого и покупает билет на то же самое место. Иначе говоря, он приобретает билет на уже занятое место. Возникает коллизия одновременного доступа к одной и той же записи в базе данных. В реальной жизни такого не должно происходить (по крайней мере, теоретически), потому что при выполнении запросов к базе данных предпринимаются специальные меры. Запросы к базе данных железнодорожных билетов, состоящие из нескольких SQL-операторов, заключаются в транзакцию, а для различных транзакций определяются так называемые *уровни изоляции*, чтобы исключить нежелательное взаимодействие между ними.

Примечание

Приложение, работающее с базой данных, может использовать и другие средства ограничения одновременного доступа, а также выполнять другие виды транзакций. В данной книге рассматриваются только транзакции SQL.

8.1. Как устроена транзакция

Транзакция состоит из любых операторов SQL, которые могут изменить базу данных. В SQL:2003 не предусмотрен специальный оператор начала транзакции. Однако некоторые реализации

SQL требуют наличия такого оператора. Он может выглядеть поразному, например, `BEGIN WORK`, `BEGIN TRAN` или `BEGIN`. В системах, совместимых с SQL:2003, если никакая транзакция еще не начата, а на выполнение отправляется оператор SQL, то автоматически создается новая транзакция. Например, операторы `CREATE TABLE` (создать таблицу), `UPDATE` (обновить) и `SELECT` (выбрать) выполняются только в транзакции.

Итак, оператора начала транзакции может и не быть. Однако для ее завершения предусмотрены два оператора:

- `COMMIT` — завершить выполнение. Выполнение всех операторов транзакции происходит последовательно в едином блоке;
- `ROLLBACK` — откат. Происходит отмена действия всех операторов транзакции, и база данных возвращается в исходное состояние, в котором она находилась до начала транзакции.

Оператор `COMMIT` применяется тогда, когда все SQL-операторы транзакции предположительно выполнены (восприняты СУБД и не вызвали ошибок) и требуется подтвердить изменения, внесенные в базу данных. Если при выполнении оператора `COMMIT` произойдет системный сбой или ошибка, можно выполнить откат транзакции, а затем попытаться выполнить ее снова.

Примечание

После ключевого слова `COMMIT` допускается необязательное ключевое слово `WORK` (работа): `COMMIT WORK`.

Оператор `ROLLBACK` применяется тогда, когда необходимо отменить изменения, внесенные транзакцией, и восстановить базу данных в прежнем состоянии. Если при выполнении оператора `ROLLBACK` произойдет системный сбой, то после перезагрузки оператор `ROLLBACK` можно выполнить снова, и он должен восстановить базу данных. Таким образом, `ROLLBACK` является отказоустойчивым оператором.

Приложение может состоять из нескольких транзакций. Если не указан явный оператор начала транзакции (например, `BEGIN WORK`), то первая транзакция начинается в самом начале приложе-

ния. Последняя транзакция заканчивается в конце приложения. Для указания конца транзакции используется оператор COMMIT или ROLLBACK. Каждый такой оператор завершает одну транзакцию и начинает следующую. Приложение с несколькими транзакциями имеет следующий вид:

```
Начало приложения
    SQL-операторы транзакции 1;
COMMIT или ROLLBACK;
    SQL-операторы транзакции 2;
COMMIT или ROLLBACK;
    ...
    SQL-операторы транзакции N;
Конец приложения
```

8.2. Определение параметров транзакции

Для транзакции можно установить параметры — режим и уровень изоляции. С этой целью применяется оператор:

```
SET TRANSACTION
    Режим,
    ISOLATION LEVEL уровеньИзоляции;
```

Режим может принимать следующие два значения:

- READ WRITE (чтение-запись) — значение по умолчанию;
- READ ONLY (только чтение).

Уровень изоляции транзакции может принимать четыре значения:

- SERIALIZABLE (последовательное выполнение) — значение по умолчанию;
- REPEATABLE READ (повторяющееся чтение);
- READ COMMITED (подтвержденное чтение);
- READ UNCOMMITED (неподтвержденное чтение).

Транзакция имеет параметры по умолчанию: READ WRITE и SERIALIZABLE. Они обычно подходят большинству пользователей

и основаны на допущениях, что база данных будет обновляться и что лучше как можно больше себя обезопасить. Режим READ WRITE позволяет вносить изменения в базу данных, а уровень изоляции SERIALIZABLE является наиболее безопасным. Если же необходимо задать другие значения, то следует использовать оператор SET TRANSACTION. При этом он записывается либо в начале приложения, определяя первую транзакцию, либо после оператора COMMIT или ROLLBACK, определяя следующую транзакцию. Если же после COMMIT или ROLLBACK не вызвать оператор SET TRANSACTION, то следующая транзакция будет иметь параметры, принятые по умолчанию.

Например:

```
SET TRANSACTION
    READ ONLY,
    ISOLATION LEVEL READ COMMITED;
```

8.3. Уровни изоляции транзакций

Наибольшая безопасность достигается при самом высоком уровне изоляции транзакций, однако за это приходится платить снижением общей производительности работы системы в многопользовательском режиме. Так, если транзакции различных пользователей базы данных полностью изолированы друг от друга, то пока не будет выполнена уже начавшаяся транзакция, другие ждут ее завершения. В связи с этим возникает задача поиска компромисса между необходимым уровнем безопасности и производительностью системы в целом. Рассмотрим вопросы, связанные с выбором различных уровней изоляции, начав с самого низкого.

8.3.1. Неподтвержденное чтение

Самый низкий (слабый) уровень изоляции — READ UNCOMMITED (неподтвержденное чтение или чтение неподтвержденных данных). При данном уровне изменения, внесенные одним пользователем, могут быть прочитаны другим, даже если первый пользо-

ватель еще не подтвердил их с помощью оператора COMMIT. Проблема возникнет в случае, если первый пользователь выполнил откат своей транзакции. Тогда все последующие действия второго пользователя оказываются основанными на неправильных (неактуальных) данных.

Таким образом, уровень READ UNCOMMITED не следует применять, когда требуются точные результаты. Он более или менее подходит в том случае, когда достаточно получить приблизительные результаты, например, при статистической обработке мало изменяющихся данных.

Очевидно, что при уровне изоляции READ UNCOMMITED обе транзакции могут выполняться одновременно.

8.3.2. Подтвержденное чтение

Следующим по силе является уровень изоляции READ COMMITED (подтвержденное чтение или чтение подтвержденных данных). В этом случае изменения, произведенные другой транзакцией, остаются невидимыми в вашей транзакции до тех пор, пока другой пользователь не завершит ее оператором COMMIT. Хотя до завершения другой транзакции внесенные ею изменения вам не видны, обе транзакции могут выполняться совместно.

По сравнению с READ UNCOMMITED данный уровень изоляции дает лучшие результаты, но не освобождает от неприятностей так называемого неповторяющегося чтения. Рассмотрим следующую ситуацию.

Допустим, первый пользователь собирается сначала прочитать данные, а затем использовать их при внесении изменений в другую таблицу. Например, он хочет сначала узнать, сколько имеется некоторого товара на складе, а затем оформить заказ на этот товар. Может случиться так, что этот пользователь прочтет устаревшие данные, которые уже изменены вторым пользователем, но еще не подтверждены им с помощью оператора COMMIT. Например, второй пользователь уже "забрал" весь запас данного товара, а первый пользователь, не зная об этом, оформил заказ на

отсутствующий товар. Таким образом, проблема заключается в том, что результаты чтения данных первым пользователем до и после выполнения вторым пользователем оператора COMMIT могут различаться. Это так называемая проблема неповторяющегося чтения.

8.3.3. Повторяющееся чтение

Уровень изоляции REPEATABLE READ (повторяющееся чтение) исключает проблему неповторяющегося чтения, рассмотренную ранее. Тем не менее, данный уровень не устраняет другую неприятность, называемую фиктивным чтением. Она возникает тогда, когда пользователь читает данные, изменяемые в результате выполнения другой транзакции. При этом изменения данных происходят во время их чтения. Рассмотрим в качестве примера следующую ситуацию.

Предположим, первый пользователь выполняет оператор выборки данных (SELECT) с некоторым условием (WHERE или HAVING). В результате он выбирает какое-то множество записей. Второй пользователь сразу же за этим изменяет данные в одной или нескольких записях, которые выбрал первый пользователь. Более того, второй пользователь завершает свою транзакцию оператором COMMIT. Таким образом, получается, что записи, ранее удовлетворявшие условию выборки, теперь перестали ему соответствовать. Не исключено, что появились новые записи, ранее не удовлетворявшие условию, а теперь соответствующие ему. В это время первый пользователь, еще не завершив свою транзакцию, не знает о произошедших изменениях и отправляет на выполнение еще один SQL-оператор с таким же условием выборки, что и в начале. Однако этот оператор будет работать не с теми же записями, что предыдущий оператор. Данная проблема и является проблемой фиктивного чтения.

8.3.4. Последовательное выполнение

Избежать неприятностей, связанных с ранее рассмотренными уровнями изоляции, позволяет уровень SERIALIZABLE (последовательное вы-

полнение). Транзакции с уровнем изоляции SERIALIZABLE всегда выполняются не параллельно, а последовательно. Если одна из них уже начата, то другая будет ожидать ее окончания. Аппаратные и программные отказы могут привести к невыполнению транзакции, однако при данном уровне изоляции можно быть уверенным, что результаты работы с базой данных будут корректными.

Предоставляя наибольшую надежность, уровень изоляции SERIALIZABLE максимально снижает общую производительность системы.

8.4. Субтранзакции

Начиная с SQL:1999 транзакции могут состоять из нескольких субтранзакций. Субтранзакции отделяются друг от друга так называемыми точками отката, которые задаются с помощью оператора SAVEPOINT (точка сохранения, отката). Он может применяться вместе с оператором ROLLBACK. Если ранее ROOLLBACK применялся для отмены всей транзакции, то теперь его можно использовать для отмены только той части транзакции, которая расположена между ROLLBACK и точкой отката.

Субтранзакции используются не для того, чтобы частично восстановить базу данных в случае возникновения ошибки (в этом случае необходимо выполнить откат всей транзакции). Бывают ситуации, когда выполняется сложная последовательность действий с множеством вариантов выбора. На каком-то этапе по полученным результатам вы можете решить, что следует пойти по другому пути, а для этого придется вернуться на несколько шагов назад. Если бы не субтранзакции, то пришлось бы отменять всю транзакцию целиком.

Точка отката помечает с помощью следующего оператора:

```
SAVEPOINT имяТочкиОтката;
```

Откат транзакции к этой точке выполняется с помощью оператора:

```
ROLLBACK TO SAVEPOINT имяТочкиОтката;
```

8.5. Ограничения в транзакциях

Ограничения, накладываемые на столбцы, таблицы и связи между ними *(см. разд. 7.1)*, оказывают влияние на выполнение транзакций. Например, таблица имеет столбец, на который наложено ограничение `NOT NULL` (значения столбца не могут быть неопределенными), и требуется добавить новые данные. Как правило, в этом случае сначала добавляют новую пустую запись, а затем заполняют ее значениями. Однако данный способ не осуществим, даже если попытаться его выполнить до оператора завершения транзакции `COMMIT`. Дело в том, что ограничение `NOT NULL` не позволит создать строку с неопределенными значениями. В SQL:2003 для решения данной проблемы можно наделить ограничение дополнительными свойствами:

- `DEFERABLE` (допускающий задержку). Ограничения с данным свойством могут быть заданы как `IMMEDIATE` (немедленные) или как `DEFERED` (задержанные);
- `NOT DEFERABLE` (не допускающий задержку). Ограничения с этим свойством выполняются немедленно.

Если ограничение типа `DEFERABLE` задано как `IMMEDIATE`, то оно действует немедленно, как и ограничение типа `NOT DEFERABLE`. Если ограничение типа `DEFERABLE` задано как `DEFERED`, то его действие может быть задержано.

Для добавления пустых записей и выполнения других операций, которые могут нарушить ограничения типа `DEFERABLE`, применяется следующий оператор:

```
SET CONSTRAINS ALL DEFERED;
```

(установить все ограничения как задержанные).

Этот оператор определяет все ограничения типа `DEFERABLE` как `DEFERED`, а на ограничения типа `NOT DEFERABLE` он не действует. После выполнения всех операций, которые могут нарушить ограничения, и достижения таблицей состояния, в котором их нарушить уже нельзя, все ограничения можно возобновить:

```
SET CONSTRAINS ALL IMMEDIATE;
```

(установить все ограничения как немедленные).

Если ограничения, ранее заданные как DEFERED, вы не задаете явно как IMMEDIATE, то при попытке завершить свою транзакцию оператором COMMIT будут активизированы все задержанные ограничения. Если к этому моменту выполняются не все ограничения, транзакция будет отменена с сообщением об ошибке.

Итак, ограничения защищают базу данных от неправильного ввода данных или от их отсутствия. Однако у вас есть возможность внутри транзакции временно отменить мешающие вам ограничения.

Рассмотрим пример, иллюстрирующий использование отсрочки ограничений. Предположим, в базе данных имеются следующие таблицы:

- Отделы (Номер_отдела, Название, Оплата);
- Сотрудники (Номер, Имя, Номер_Отдела, Зарплата).

В таблице Отделы столбец Номер_отдела является первичным ключом, а столбец Оплата содержит суммы зарплат всех сотрудников соответствующего отдела.

В таблице Сотрудники столбец Номер_отдела является внешним ключом, который ссылается на таблицу Отделы.

Каждый раз, когда изменяется значение столбца Зарплата в таблице Сотрудники, необходимо изменить и значение в столбце Оплата таблицы Отделы. Чтобы обеспечить правильность значений, в определение таблицы Отделы можно вставить следующее ограничение:

```
CREATE TABLE Отделы
(
    Номер_отдела  CHAR (3) PRIMARY KEY,
    Название      CHAR (20),
    Оплата        NUMERIC (12, 2),
    CHECK (Оплата = (SELECT SUM(Зарплата)
        FROM Сотрудники
        WHERE Сотрудники.Номер_отдела=Отделы.Номер_отдела))
);
```

Пусть необходимо увеличить на 1000 зарплату сотрудника с номером 15. Для этого необходимо обновить обе таблицы:

```
UPDATE Сотрудники
    SET Зарплата = Зарплата + 1000
    WHERE номер = 15;

UPDATE Отделы
    SET Оплата = Оплата +1000
    WHERE Отделы.Номер_отдела =
        (SELECT Сотрудники.Номер_отдела
        FROM Сотрудники
        WHERE Сотрудники.Номер_сотрудника = 15);
```

При выполнении этих операторов возникнет некоторая трудность. На практике реализации SQL проверяют выполнение только тех ограничений, которые относятся к измененным значениям. После выполнения первого из приведенных операторов `UPDATE` проверяются все ограничения, которые ссылаются на измененные им значения. К таким ограничениям относится и то, которое определено для таблицы `Отделы`. Действительно, это ограничение касается столбца `Зарплата` таблицы `Сотрудники`, а значение этого столбца изменяется оператором `UPDATE`. Данное ограничение после выполнения первого оператора `UPDATE` оказалось нарушенным. Так, если до его выполнения каждое значение столбца `Оплата` было равно сумме зарплат всех сотрудников соответствующего отдела, то после выполнения этого оператора равенство нарушилось.

Данная ситуация исправляется вторым оператором `UPDATE`, но в тот момент, когда первый оператор выполнился, а второй еще нет, ограничение нарушено. С помощью оператора `SET CONSTRAINTS DEFERED` можно временно отключить все или только некоторые ограничения. Действие этих ограничений будет отложено до тех пор, пока не выполнится оператор `SET CONSTRAINTS IMMEDIATE` или не будет вызван один из двух операторов завершения транзакции — `COMMIT` или `ROLLBACK`.

Итак, для обновления таблиц следует применить следующие операторы:

```
SET CONSTRAINTS DEFERED;
UPDATE Сотрудники
    SET Зарплата = Зарплата + 1000
    WHERE номер = 15;

UPDATE Отделы
    SET Оплата = Оплата +1000
    WHERE Отделы.Номер_отдела =
        (SELECT Сотрудники.Номер_отдела
        FROM Сотрудники
        WHERE Сотрудники.Номер_сотрудника = 15);
SET CONSTRAINTS IMMEDIATE;
```

В данном SQL-коде временно отменяются все ограничения. Так, при добавлении новой записи в таблицу `Отделы` не будет проверяться и первичный ключ (ограничение типа `PRIMARY KEY`), что, очевидно, плохо. Поэтому следует указывать конкретно, какие ограничения необходимо задержать. С этой целью при создании ограничений им следует назначать имена:

```
CREATE TABLE Отделы
(
    Номер_отдела  CHAR (3) PRIMARY KEY,
    Название      CHAR (20),
    Оплата        NUMERIC (12, 2),
    CONNSTRAINT PayEq
    CHECK (Оплата = (SELECT SUM(Зарплата)
        FROM Сотрудники
        WHERE Сотрудники.Номер_отдела=Отделы.Номер_отдела))
);
```

Здесь `PayEq` — имя ограничения. На поименованные ограничения можно ссылаться в операторе `SET CONSTRAINTS` *имяОграничения*:

```
SET CONSTRAINTS PayEq DEFERED;
UPDATE Сотрудники
```

```
    SET Зарплата = Зарплата + 1000
    WHERE номер = 15;

UPDATE Отделы
    SET Оплата = Оплата +1000
    WHERE Отделы.Номер_отдела =
        (SELECT Сотрудники.Номер_отдела
        FROM Сотрудники
        WHERE Сотрудники.Номер_сотрудника = 15);
SET CONSTRAINTS PayEq IMMEDIATE;
```

Временно отменять ограничения можно только внутри транзакций. По завершении транзакции все задержанные ограничения сразу же вновь вступают в силу. Правильное использование этой возможности не позволит транзакции создать данные, нарушающие ограничения.

Глава 9

Курсоры и применение SQL в приложениях

Как вы уже могли заметить, SQL-запросы обеспечивают обработку данных одновременно всех строк (записей) таблицы. SQL-запрос на выборку данных возвращает таблицу, которая содержит одну, несколько или ни одной записи. Результат запроса можно просмотреть или скомбинировать с результатом другого запроса с помощью операторов SQL, т. е. сформировать новый комбинированный запрос, который должен вернуть таблицу. При этом допустимы некоторые преобразования данных для представления в результатной таблице, но далеко не все, которые часто требуются на практике. Иначе говоря, нередко возникает потребность обработать данные так, как SQL не позволяет или же делает это не эффективно.

Свободу в обработке табличных данных предоставляют процедурные языки программирования, такие как C, C++, Visual Basic, Pascal, FORTRAN и др. Однако применительно к базам данных они имеют особенности, главная из которых состоит в том, что обработка табличных данных происходит обычно в цикле по отдельным строкам (записям) таблиц. Приложения получают доступ к отдельной записи и обрабатывают ее поля подобно обычным переменным. При необходимости они переходят к той или иной записи и снова обрабатывают ее поля. Иначе говоря, традиционной технологией работы с базой данных в приложениях является обработка полей одиночных записей таблиц базы данных.

А поскольку многие операции по обработке табличных данных более эффективно выполняются средствами SQL, перечисленные процедурные языки допускают вставки SQL-кодов. Иначе говоря, программы на процедурных языках, работающие с базами данных, допускают вставки фрагментов, написанных на языке SQL. Для того чтобы адаптировать основную часть SQL к особенностям обработки табличных данных программами на процедурных языках, в SQL были введены понятие курсора и поддерживающий его механизм. *Курсор* позволяет сделать обработку одной или нескольких записей таблицы с помощью SQL-выражений и, таким образом, упрощает совместное использование SQL с другими языками программирования.

Курсоры применяются в ситуациях, когда требуется выбрать одну или несколько записей из таблицы, проверить их содержимое и по результатам этой проверки выполнить некоторые операции. В данном случае SQL используется совместно с процедурным языком, на котором написано приложение, работающее с базой данных. Выборка записей производится с помощью SQL-запроса, а проверка их содержимого — с помощью кода на процедурном языке. Процедурные языки имеют и собственные средства поиска требуемых записей. Однако если обработку результата запроса можно выполнить средствами SQL, то именно так и рекомендуют делать, а процедурный язык стоит использовать для построчной обработки данных только тогда, когда нельзя обойтись обычным SQL.

Примечание

В *разд. 10.3.4* будет кратко описан способ применения курсоров без обращения к кодам на процедурном языке.

Курсоры оказываются очень полезными, если требуется обработать результаты SQL-запроса, вернувшего слишком большой объем данных. Результат запроса может содержать десятки тысяч записей, что может занять значительную часть памяти компьютера и/или пропускной способности сети. Получение такого объема данных в один прием с помощью обычного оператора `SELECT` может привести к ощутимым временным затратам. Например,

если требуется просто вывести какие-либо данные на монитор, то разумнее выбирать и выводить их по частям, скажем, по 15—25 записей (столько строк, сколько помещается в области отображения). Подготовку различного рода отчетов, предназначенных для вывода табличных данных постранично на печать, также целесообразно выполнять по частям.

В большинстве случаев мы заранее не знаем, сколько записей вернет обычный запрос на выборку данных — ноль, одну или несколько. Можно быть уверенным в получении единственной записи, если в операторе `SELECT` указана итоговая (агрегатная) функция (например, `COUNT()`, `SUM()` и др.) или же выборка осуществляется по значению первичного ключа. В остальных случаях запрос может вернуть сколько угодно записей, в том числе и ни одной. Использование курсоров представляет собой универсальный способ для обработки заранее неизвестного количества записей, возвращаемых запросом.

Курсор имеет область действия (набор записей), команды перемещения указателя курсора на ту или иную запись в области действия и способ передачи значений столбцов отдельной записи в переменные, доступные приложению, написанному на процедурном языке. Чтобы задать и применить курсор, используются несколько SQL-операторов:

- `DECLARE CURSOR` (объявить курсор) — определяет имя курсора и область его действия (выборку записей);
- `OPEN` (открыть) — активизирует объявленный курсор, т. е. собирает все записи, соответствующие запросу в операторе `DECLARE CURSOR`;
- `FETCH` (получить, достать) — осуществляет получение данных, т. е. перемещение в области действия курсора и запись значений столбцов в переменные, доступные в приложении;
- `CLOSE` (закрыть) — закрывает указанный курсор.

Отчасти курсоры могут использоваться вне всякой связи с программами на процедурных языках. Однако это редко используется на практике. Обычно курсоры применяются как механизм связи

SQL-выражений и программ обработки данных на процедурных языках. Тем не менее, мы сначала рассмотрим курсоры сами по себе, т. е. в рамках SQL, а затем их использование в приложениях.

9.1. Объявление курсора

Прежде чем использовать курсор, его необходимо объявить для СУБД. Объявление курсора еще не приводит к каким-либо действиям с данными, а лишь задает имя курсора и определяет запрос, с которым он будет работать. Объявление курсора производится с помощью выражения, начинающегося с ключевого слова DECLARE:

```
DECLARE имяКурсора
    [чувствительность]
    [перемещаемость]
CURSOR
    [фиксируемость]
    [возвращаемость]
FOR выражениеЗапроса
    [ORDER BY порядокСортировки]
    [FOR разрешениеОбновления];
```

Здесь квадратные скобки являются не элементами синтаксиса, а лишь указывают, что заключенный в них элемент не является обязательным. Все SQL-выражение объявления курсора может быть записано в одну или несколько строк. Явным признаком окончания выражения DECLARE CURSOR, как и любого другого SQL-выражения, является точка с запятой.

Имя курсора задается обязательно, поскольку оно используется в других операторах при работе с курсором. Кроме того, можно объявить несколько курсоров, каждый из которых должен иметь уникальное имя.

Далее приведен пример объявления курсора:

```
DECLARE phone348 CURSOR FOR
```

```
SELECT Имя, Адрес, Телефон
    FROM Клиенты
    WHERE Телефон LIKE '348%';
```

9.1.1. Чувствительность

Выражение запроса, указанное после ключевого слова `FOR`, определяет набор записей, которые составляют область действия курсора. Однако программный код обработки данных может внести такие изменения данных, что они перестанут удовлетворять условию запроса, т. е. выпадут из области действия курсора. *Чувствительность* курсора определяет, как в этом случае будет вести себя команда курсора.

Чувствительность курсора может быть задана следующими ключевыми словами:

- `SENSITIVE` (чувствительный);
- `INSENSITIVE` (нечувствительный);
- `ASENSITIVE` (не обладающий чувствительностью).

Чтобы курсор вел себя с учетом произведенных изменений, т. е. был чувствительным к изменениям, необходимо установить значение `SENSITIVE`. Команда курсора может обрабатывать записи, которые изначально соответствовали условию запроса, и тогда следует установить значение `INSENSITIVE`. Наконец, курсор может просто игнорировать измененные или удаленные записи. В этом случае устанавливается значение `ASENSITIVE`. Значение `ASENSITIVE` устанавливается по умолчанию, а его смысл зависит от конкретной реализации SQL. Он может совпадать как с `SENSITIVE`, так и с `INSENSITIVE`, поэтому смысл `ASENSITIVE` в вашей системе следует уточнить по справочной документации.

9.1.2. Перемещаемость

Перемещаемость курсора может принимать два возможных значения:

- `SCROLL` (с прокруткой);
- `NO SCROLL` (без прокрутки).

Значение SCROLL позволяет получить доступ к записям в области действия курсора в любом порядке. Управление перемещением осуществляется с помощью оператора FETCH. Значение NO SCROLL запрещает перемещение по записям.

Примечание

Свободное перемещение по записям при установленном значении SCROLL обеспечивается в SQL-92 и SQL:2003. В SQL-86 и SQL-89 разрешалось только последовательное перемещение курсора, начиная с первой записи из области действия курсора, которая определена условием запроса.

9.1.3. Выражение запроса

Выражение запроса, указанное после ключевого слова FOR, представляет собой любое правильное выражение, начинающееся с ключевого слова SELECT, т. е. запрос на выборку данных. Оператор SELECT задает *область действия курсора* — множество записей, удовлетворяющих условию запроса. При этом он вернет запись, на которую указывает курсор в данный момент времени. Однако выполнение оператора DECLARE CURSOR еще не приводит к выполнению самого запроса. Выборка данных происходит только при выполнении оператора OPEN.

9.1.4. Сортировка

Если требуется, чтобы записи обрабатывались в определенном порядке, можно задать их предварительную сортировку (упорядочивание). Это делается с помощью оператора ORDER BY (сортировать по). После ключевых слов ORDER BY указывается одна или несколько спецификаций сортировки. Если таких спецификаций больше одной, то они разделяются запятыми.

Каждая спецификация сортировки имеет следующий синтаксис:

имяСтолбца [COLLATE BY *имяСопоставления*] [ASC | DESC]

Здесь квадратные скобки являются не элементами синтаксиса, а лишь указывают на то, что заключенный в них элемент не является

обязательным. Вертикальная черта разделяет допустимые значения (в конкретном выражении может использоваться лишь одно из них).

Имя столбца, по значениям которого следует упорядочить записи, должно присутствовать в списке столбцов, указанном после ключевого слова `SELECT`. При этом столбец может быть и вычисляемым, т. е. создаваемым на основе реальных столбцов таблицы (или таблиц). В данном случае следует использовать псевдонимы для вычисляемых столбцов.

По умолчанию принята сортировка в алфавитном порядке или по возрастанию. Вместе с тем, можно выбрать и другой порядок (сопоставление). Каждая реализация SQL предусматривает несколько часто используемых сопоставлений. Это следует выяснить в справочной документации. В этом случае применяется оператор `COLLATE BY` (сопоставить с).

Чтобы указать, в каком порядке следует производить сортировку записей, применяются ключевые слова `ASC` (по возрастанию) или `DESC` (по убыванию). Если ни одно из этих значений не указано, то по умолчанию предполагается `ASC`.

Далее приведен пример объявления курсора с сортировкой:

```
DECLARE name_sum CURSOR FOR
SELECT Имя, Адрес, Сумма_заказа
    FROM Клиенты
ORDER BY Имя, Сумма_заказа DESC;
```

Здесь записи сортируются сначала по именам клиентов в алфавитном порядке, а затем по сумме их заказов в порядке убывания.

В следующем примере применяется сортировка по вычисляемому столбцу:

```
DECLARE name_sum2 CURSOR FOR
SELECT Имя, SUM(Сумма_заказа) AS Сумма
    FROM Клиенты
    GROUP BY Имя
ORDER BY Имя, Сумма DESC;
```

9.1.5. Разрешение обновления

Изменение значений в записях, а также удаление записей в области действия курсора можно разрешить или запретить. В операторе DECLARE CURSOR разрешение устанавливается с помощью оператора FOR.

Чтобы разрешить изменения значений в указанных столбцах и удаление записей, используется следующее выражение:

```
FOR UPDATE OF списокСтолбцов
```

Здесь *списокСтолбцов* — список имен столбцов, разделенных запятыми, в которых разрешено изменение (обновление) значений. Столбцы из этого списка должны присутствовать в выражении запроса оператора DECLARE CURSOR *(см. разд. 9.1.3)*.

Если оператор FOR отсутствует в DECLARE CURSOR, то считается, что значения всех столбцов, указанных в выражении запроса, могут быть изменены. В данном случае с помощью оператора UPDATE можно изменить значения всех столбцов записи в области действия курсора, а с помощью оператора DELETE — удалить запись.

Для запрещения изменения и удаления записей в области действия курсора в операторе DECLARE CURSOR используется выражение:

```
FOR READ ONLY
```

9.2. Открытие и закрытие курсора

Рассмотренный в *разд. 9.1* оператор DECLARE CURSOR определяет, какие записи следует включить в область действия курсора. Он только объявляет курсор для СУБД, не производя выборку данных или каких-либо других действий.

Активизация курсора производится с помощью оператора открытия курсора:

```
OPEN имяКурсора;
```

Пока курсор не открыт, выборка записей из области действия курсора не возможна.

В следующем примере курсор сначала объявляется, а затем открывается:

```
DECLARE name_sum CURSOR FOR
SELECT Имя, Адрес, Сумма_заказа
    FROM Клиенты
ORDER BY Имя, Сумма_заказа DESC;
OPEN name_sum;
```

В SQL-выражениях, применяемых в приложениях на базовом языке, могут использоваться переменные. После открытия курсора значения этих переменных, а также значения функций даты-времени становятся фиксированными. Другими словами, последующее присваивание в программе на базовом языке этим переменным новых значений не повлияет на выбранные курсором записи.

По окончании работы с курсором его рекомендуется закрыть, поскольку открытые курсоры занимают системные ресурсы. Для этого используется оператор:

```
CLOSE имяКурсора;
```

9.3. Работа с отдельными записями

Имя и область действия курсора задаются с помощью оператора `DECLARE CURSOR`. Активизация курсора и выборка записей, удовлетворяющих выражению запроса в `DECLARE CURSOR`, выполняются оператором `OPEN`. При этом открытый курсор может указывать:

- на одну из записей в области действия курсора;
- или на область, находящуюся перед первой записью области действия курсора;
- или на область, находящуюся после последней записи области действия курсора.

На что именно указывает курсор, а также перемещение курсора по записям определяются с помощью оператора `FETCH`:

```
FETCH [[ориентация] FROM] имяКурсора
INTO списокЦелевыхСпецификаций;
```

Квадратные скобки указывают, что заключенный в них элемент не является обязательным. Все выражение оператора FETCH может быть записано в одну или несколько строк. Явным признаком окончания выражения FETCH, как и любого другого SQL-выражения, является точка с запятой.

Ориентация курсора, определяющая, на что он указывает, может иметь следующие значения:

- NEXT (следующая) — куда бы ни указывал курсор в данный момент, он перемещается на следующую запись в порядке, установленном сортировкой. Если курсор указывает на область перед первой записью, то происходит переход на первую запись и завершение его работы. Если курсор находится на последней записи, то он прекращает свою работу и в системную переменную SQLSTATE записывается код отсутствия данных. Ориентация NEXT принята по умолчанию;
- PRIOR (предыдущая) — перемещает курсор на предыдущую запись. Если курсор находится на первой записи, то он прекращает свою работу и в системную переменную SQLSTATE записывается код отсутствия данных;
- FIRST (первая) — перемещает курсор на первую запись;
- LAST (последняя) — перемещает курсор на последнюю запись;
- ABSOLUTE *n* — перемещает курсор на запись с номером *n*, отсчитанным от первой записи, если *n* — положительное число. Если *n* отрицательное, отсчет и перемещение начинаются с последней записи в обратном порядке. Так, FETCH ABSOLUTE -1 перемещает указатель на последнюю запись. FETCH ABSOLUTE -2 перемещает его на предпоследнюю запись и т. д. При попытке переместиться за границы области действия курсора генерируется код отсутствия данных. Например, это произойдет в случае выполнения FETCH ABSOLUTE 0;
- RELATIVE *n* — перемещает курсор на *n* записей относительно его текущей позиции. Если *n* — положительное число, то перемещение происходит вниз по выборке, а если отрицательное — вверх. FETCH RELATIVE 0 не перемещает курсор. FETCH

`RELATIVE 1` действует так же, как и `FETCH NEXT`, а `FETCH RELATIVE -1` — как `FETCH PRIOR`. При попытке переместиться за границы области действия курсора генерируется код отсутствия данных;

- *спецификацияПростогоЗначения* — перемещает указатель на запись, заданную спецификацией простого значения.

Если ориентация в операторе `FETCH` не указана, то курсор устанавливается на первую запись в области действия. Последующие перемещения курсора могут вывести его за область действия. Поэтому при обработке данных с использованием курсора необходимо следить за значением системной переменной `SQLSTATE`.

Оператор `FETCH` обеспечивает не только перемещение курсора по записям области действия, но и присвоение переменным приложения значений столбцов в указанной записи. Значения этих переменных могут быть обработаны приложением на базовом языке.

В операторе `FETCH` после ключевого слова `INTO` указывается одна или несколько целевых спецификаций, т. е. описаний, куда будут записываться прочитанные из столбцов данные. Если целевых спецификаций больше одной, то они разделяются запятыми. Целевыми спецификациями могут быть базовые переменные, если курсор определен во встроенном SQL, или параметры, если он определен в модуле. Количество и типы целевых спецификаций должны соответствовать количеству и типам столбцов, указанных в выражении запроса оператора `DECLARE CURSOR`. Так, например, в случае встроенного SQL, если из таблицы выбираются значения трех столбцов, то в выражении `INTO` должны быть указаны три переменные с правильно подобранными типами. Перед именами переменных в списке целевых спецификаций ставится двоеточие. Это общее правило использования переменных в SQL-выражениях. Например:

```
DECLARE name_sum CURSOR FOR
SELECT Имя, Адрес, Сумма_заказа
    FROM Клиенты
ORDER BY Имя, Сумма_заказа DESC;
```

```
OPEN name_sum;
FETCH LAST FROM name_sum
INTO :name, :address, :summa;
```

Здесь сначала объявляется курсор (задается имя `name_sum` и область действия, определяемая выражением `SELECT`), затем он открывается, а после этого происходит перемещение указателя курсора на последнюю запись в области действия и считывание значений столбцов `Имя`, `Адрес` и `Сумма_заказа` таблицы `Клиенты` в переменные `name`, `address` и `summa` соответственно. Предполагается, что эти переменные определены ранее, причем их типы соответствуют типам соответствующих столбцов. Подробнее о переменных для связи с программой на процедурном языке будет рассказано в *разд. 9.4*.

Запись, на которую в данный момент указывает курсор, может быть изменена или удалена. При этом используются операторы `UPDATE` и `DELETE` с указанием имени курсора.

Для изменения записи, на которую указывает курсор, используется выражение вида:

```
UPDATE имяТаблицы
SET имяСтолбца = значение [, имяСтолбца2 = значение2, ...]
WHERE CURRENT OF имяКурсора;
```

Значение, которое записывается в указанный столбец, может быть выражением или ключевым словом (по умолчанию `DEFAULT`). В случае какой-либо ошибки, возникшей при выполнении оператора `UPDATE`, изменение данных не происходит. Ключевые слова `WHERE CURRENT OF` означают, что изменение требуется произвести в текущей записи указанного курсора.

По-русски приведенное ранее выражение выглядит следующим образом: "обновить таблицу *имяТаблицы*, установив значения столбцов *имяСтолбца* = *значение* [, *имяСтолбца2* = *значение2*, ...] в текущей записи курсора *имяКурсора*".

Для удаления записи, на которую указывает курсор, используется выражение вида:

```
DELETE имяТаблицы WHERE CURRENT OF имяКурсора;
```

Если курсор не указывает на какую-либо запись внутри области действия, то возникает ошибка и удаление не происходит.

По-русски приведенное выражение выглядит следующим образом: "удалить в таблице *имяТаблицы* текущую запись курсора *имяКурсора*".

9.4. SQL в приложениях

Одно из самых значительных достоинств SQL проявляется в получении данных. Но в SQL нет средств для форматирования выводимых данных и генерации красивых отчетов. SQL может работать сразу с целой таблицей в том смысле, что независимо от количества записей в этой таблице необходимые данные можно извлечь из нее с помощью единственного выражения `SELECT`. Однако работа с одиночными записями в SQL не столь проста. Для этого придется использовать курсоры, описанные в предыдущих разделах.

В отличие от SQL, процедурные языки, такие как C, C++, Visual Basic, Pascal, FORTRAN, PHP и др. довольно легко оперируют отдельными записями таблиц базы данных. Однако разработчик приложения на этих языках должен знать и учитывать, каким образом данные хранятся в таблицах базы данных, как расположены столбцы и записи. Процедурные языки достаточно гибки, чтобы с их помощью можно было организовать довольно сложную обработку и отображение на экране данных, а также генерацию сложных отчетов для вывода на печать.

На практике довольно часто имеет смысл комбинировать SQL-запросы с программными кодами на процедурных языках. При этом важно получить объединение как можно большего количества их достоинств и меньшего числа недостатков.

9.4.1. Использование встроенного SQL

Многие языки программирования допускают вставки в свой программный код SQL-выражений. Перед компиляцией или интерпретацией программы с такими встроенными фрагментами на

SQL пропускаются через препроцессор. Каждому встроенному SQL-выражению должна предшествовать специальная команда (директива), сообщающая препроцессору, что за ней следует выражение не на базовом языке, а на SQL. Обычно это директива `EXEC SQL`.

В следующем примере показано, как в программу на базовом языке вставить SQL-операторы объявления и открытия курсора:

```
... // Код на базовом языке
// Объявление курсора
EXEC SQL DECLARE name_sum CURSOR FOR
    SELECT Имя, Адрес, Сумма_заказа
    FROM Клиенты;
... // Код на базовом языке
// Открытие курсора
EXEC SQL OPEN name_sum;
... // Код на базовом языке
```

Для передачи данных между программой на базовом языке и фрагментом на SQL служат переменные (их также называют базовыми переменными). Прежде чем их использовать в SQL-выражении, они должны быть объявлены. Объявление переменных располагается в специальном сегменте объявлений перед программным сегментом. Сегмент объявлений начинается с директивы:

```
EXEC SQL BEGIN DECLARE SECTION;
```

В конце сегмента объявлений указывается директива:

```
EXEC SQL END DECLARE SECTION;
```

В следующем примере на языке C показано, как объявляются переменные и как они могут использоваться в SQL-выражении:

```
// Секция объявления переменных
EXEC SQL BEGIN DECLARE SECTION;
    FLOAT summa;   // Объявление числовой переменной summa
EXEC SQL END DECLARE SECTION;
// Программа на языке C
main () {
```

```
    // Объявление курсора с именем name_sum
    EXEC SQL DECLARE name_sum CURSOR FOR
        SELECT Сумма_заказа
        FROM Клиенты
        WHERE Имя = "Иванов Иван Иванович";
    // Открытие курсора name_sum
    EXEC SQL OPEN name_sum;
    /* Чтение значения столбца Сумма_заказа
       в записи курсора в переменную summa */
    EXEC SQL FETCH FROM name_sum INTO :summa;
    if (summa > 20000) {    // Обработка данных
        // Обновление значения столбца Сумма_заказа
        EXEC SQL UPDATE Клиенты
            SET Сумма_заказа = :summa*0.9
            WHERE CURRENT OF name_sum;
    }
    EXEC SQL CLOSE name_sum;    // Закрытие курсора
}
```

В приведенном коде языке выполняется следующее:

1. В сегменте объявлений объявляется переменная `summa` типа `FLOAT` (числовой тип с плавающей разделительной точкой).
2. В главной функции `main()` объявляется и открывается курсор `name_sum`.
3. Из текущей (первой) записи в области действия курсора читается значение столбца `Сумма_заказа` и записывается в переменную `summa`.
4. Если значение переменной `summa` превышает 20 000, то клиенту делается 10% скидка и новое значение суммы заказа (`summa*0.9`) записывается в столбец `Сумма_заказа` в текущей записи курсора. В противном случае ничего не делается.
5. Курсор закрывается.

В данном примере предполагается, что соединение с базой данных уже установлено.

Примечание

В операторе SELECT для обозначения всех столбцов таблицы можно использовать символ (*). Но в случае использования встроенного SQL в программах рекомендуется не использовать этот символ, а явно указывать требуемые столбцы. Дело в том, что пользователь или вы, как разработчик приложения, может добавить или удалить столбцы уже после создания приложения. В этом случае, если использовался символ (*), приложение может получить не тот набор столбцов, на который оно было первоначально рассчитано.

9.4.2. Использование программ на языке PHP

Как известно, базы данных существуют не только на локальных компьютерах и в локальных сетях, но и в Интернете. Участники всемирной сети получают к ним доступ через различные интернет-приложения. интернет-магазины, гостевые книги и даже баннеры — типичные примеры применения баз данных. При создании Web-приложений в настоящее время широко используется PHP — язык программирования и соответствующая технология. На PHP обычно пишутся программы, выполняемые на стороне Web-сервера. Результаты их работы могут отображаться обычными Web-браузерами. Разумеется, что PHP-программа может что-то сделать (например, записать в файл или базу данных сведения, полученные от пользователя с предварительной обработкой или без нее), ничего не отображая в браузере клиента. Среди таких программ важное место занимают приложения, которые предназначены для работы с базами данных, установленными на сервере и управляемыми СУБД. PHP может работать с различными СУБД, такими как Microsoft SQL Server (MS SQL), MySQL, Oracle, PostgreeSQL, mSQL, Informix, Sybase, Ingres и IBM DB2. Кроме того, PHP поддерживает стандарт ODBC (Open Database Connectivity — открытый интерфейс доступа к базам данных), который поддерживают почти все реляционные СУБД. Для взаимодействия с конкретной СУБД в PHP имеется свой набор спе-

циальных функций. Эти функции из различных наборов похожи, но имеются и различия между ними. Особенность PHP (как, впрочем, и JavaScript, VBScript и др. скриптовых языков) состоит в том, что для работы с БД SQL-выражения не встраиваются в оригинальный код программы на процедурном языке (пусть даже посредством специальных ключевых слов типа EXEC SQL), а передаются СУБД из кода программы посредством вызова специальных функций, специально предназначенных для взаимодействия с БД из прикладных программ.

PHP распространяется бесплатно (файлы модуля PHP находятся на сайте **www.php. net**), и практически все Web-серверы его поддерживают. При создании Web-приложения (например, сайта с использованием серверных ресурсов) вам, скорее всего, потребуется его протестировать, прежде чем опубликовать на сервере. В этом случае вы можете установить на своем локальном компьютере программное обеспечение Web-сервера и настроить его на работу с PHP. В настоящее время существуют несколько программ, выполняющих роль Web-сервера, например, Appache и Microsoft Internet Information Services (IIS, Информационные службы Интернета). Последняя поставляется в дистрибутиве операционной системы Microsoft Windows XP Pro. Вы можете установить на локальном компьютере Web-сервер, PHP и согласовать их взаимодействие. Как это делать, рассказывается в соответствующей справочной документации.

При установке PHP на локальном компьютере следует иметь в виду, что библиотеки для работы с СУБД автоматически не устанавливаются. Это надо сделать вручную, что совсем не сложно. Как установить PHP и его библиотеки, рассказывается в документации (справочном руководстве), которая также доступна через Интернет. Так, при работе в Windows для СУБД MySQL версии не старше 4.0 требуется библиотека php_mysql.dll, для MySQL версии 4.1 и старше — php_mysqli.dll; для MS SQL Server — php_mssql.dll; для PostgreeSQL — php_pgsql.dll. Вам требуется указать в ini-файле для PHP, что нужную библиотеку следует подключить.

При взаимодействии PHP-программы с базой данных обычно выполняются следующие действия:

1. Подключение к серверу базы данных (предполагается, что сервер базы данных запущен).
2. Передача SQL-запроса, который должен быть выполнен СУБД. SQL-запрос представляется на язык SQL. Он оформляется в виде строки и передается специальной PHP-функции в качестве параметра. Эта функция свяжется с СУБД и передаст ей SQL-запрос. СУБД его выполнит (если нет ошибок и других проблем). Если SQL-запрос предполагает возврат данных, то они будут получены с помощью специальной PHP-функции.
3. Получение от СУБД данных, соответствующих переданному ранее SQL-запросу. С помощью специальных функций можно получить данные и далее их обработать (например, просто вывести на экран).
4. Обработка полученных данных. Вы можете отформатировать по своему усмотрению или подвергнуть более сложному анализу и преобразованию полученных данных.
5. Отключение от базы данных, если не предполагается повторно выполнить этапы 2 — 4. Отключаться от базы данных рекомендуется всякий раз, как только вы прекращаете работу с ней на более или менее длительный срок.

Эти и другие действия в PHP-программе выполняются, как уже отмечалось, посредством соответствующих функций. Рассмотрим их более подробно на примере работы с СУБД MS SQL Server. Будем считать, что база данных (возможно, пустая) уже создана. Чтобы выполнить необходимые действия, нужно к ней подключиться.

Подключение к базе данных (БД) осуществляется с помощью специальных функций PHP, которым передаются в качестве параметров следующие данные:

- местоположение (host) — для многих серверов БД это доменное имя или IP-адрес компьютера, на котором расположена

БД; если она находится на том же компьютере, что и модуль PHP, то достаточно указать `localhost`. Для MS SQL Server указывается его имя. При необходимости можно указать через двоеточие порт, который прослушивается сервером базы данных;

- регистрационное имя (`username`) — строка символов, соответствующая имени пользователя, под которым он может получить доступ к БД; сообщается администратором БД;
- пароль (`password`) — строка символов, которая вместе с регистрационным именем служит для обеспечения доступа к БД; сообщается администратором БД;
- имя базы данных — СУБД может поддерживать несколько БД, каждая из которых должна иметь уникальное имя. Вы должны указать имя конкретной БД, с которой хотите работать.

Вот пример PHP-кода подключения к БД под управлением СУБД MS SQL Server:

```
$connect=mssql_connect(имя_сервера_БД, username, password);
$db=mssql_select_db(имя_БД, $connect);
```

Здесь используются две функции:

- первая функция — `mssql_connect()` — выполняет соединение с сервером БД и возвращает идентификатор соединения при успехе и false — при неудаче. При этом в первом параметре данной функции указывается имя сервера БД, устанавливаемое при регистрации сервера. Идентификатор соединения, возвращаемый функцией, используется в программе, в частности, при выборе базы данных, создании запросов и т. д.;
- вторая функция — `mssql _select()` — выбирает конкретную базу данных на сервере, на который указывает идентификатор `$connect`, возвращенный первой функцией. Эта функция возвращает true при успешном завершении операции и false — в противном случае. Если требуемая база данных еще не создана, то это необходимо сделать. Так, в MS SQL Server базу данных (хотя бы пустую) можно создать с помощью утилиты **Enterprise Manager** или с помощью соответствующих выра-

жений SQL, которые могут быть введены с помощью оконного интерфейса либо сформированы и выполнены посредством PHP-программы. Для отключения от базы данных используется функция, которой передается идентификатор соответствующего подключения. Например, `mssql_close($connect)` — для MS SQL Server, `mysql_close($connect)` — для MySQL.

Аналогичный PHP-код для подключения к БД MySQL выглядит следующим образом:

```
$connect=mysql_connect(host, username, password);
$db=mysql_select_db($connect, имя_БД);
```

Подробные сведения о функциях работы с базами данных как рассмотренных в этой книге, так и других, можно найти в справочном руководстве по PHP.

Итак, вы подключились к серверу баз данных и выбрали требуемую конкретную базу данных. Теперь с ней можно выполнять различные манипуляции, которые задаются выражениями на языке SQL.

SQL-запрос к базе данных в качестве строкового параметра передается специальной функции PHP, которая, в свою очередь, передает его СУБД. СУБД выполняет этот запрос, а PHP-функция возвращает указатель на временную таблицу данных, являющуюся результатом выполнения запроса. Для MS SQL Server вызов функции передачи SQL-запроса выглядит так:

```
mssql_query(SQL_запрос, идентификатор_соединения);
```

Ниже приводится пример подключения к серверу MS SQL Server.

```
/* Подключение к серверу БД: */
$connect=mssql_connect("VADIM", "guest")or
              exit("Не удалось соединиться с сервером");
/* Выбор БД: */
$db=mssql_select_db("gbook", $connect)
              exit("Не удалось выбрать БД");
$strsql="SELECT * FROM gb";  // строка с SQL-запросом
$selectall=mssql_query($strsql, $connect); /* выполнение SQL-
запроса */
$mssql_close($connect);      // отключение от БД
```

Здесь производится подключение к MS SQL Server с именем `VADIM` (устанавливается при регистрации сервера) для пользователя с регистрационным именем `guest` без пароля. Затем выбирается база данных `gbook` и выполняется SQL-запрос на выборку из таблицы `gb` всех полей (столбцов) и записей (строк). Наконец, соединение с БД разрывается.

Подключение к серверу и выбор БД может оказаться неудачным по тем или иным причинам. Однако выполнение программы продолжится, что может оказаться бессмысленным и вызвать лавинообразный вывод сообщений об ошибках. Чтобы этого не происходило, используется функция `exit()` (или `die()`), выводящая указанное сообщение и прекращающая дальнейшее выполнение программы.

Ниже приведен аналогичный код программы для подключения к БД MySQL:

```
/* Подключение к серверу БД MySQL: */
$connect=mysql_connect("localhost", "guest") or
              exit("Не удалось соединиться с
       сервером".mysql_error());
/* Выбор БД: */
$db= mysql_select_db("gbook", $connect) or
              exit("Не удалось выбрать БД"".mysql_error());

$strsql="SELECT * FROM gb";  // строка с SQL-запросом
$selectall= mysql_query($strsql, $connect);       // выполнение SQL-запроса
$ mysql_close($connect);     // отключение от БД
```

Здесь функция `mysql_error()` возвращает описание возникшей ошибки.

Примечание

Подключение к базе данных для различных СУБД производится в большинстве случаев похожими функциями. Однако могут быть и существенные отличия. Так например, подключение к серверу PostgreeSQL и выбор базы данных выполняется с помощью одной функции:

```
$connect=pg_connect("host=host user=username password=password
              dbname=имя_БД");
```

Выбор нужных данных из одной или нескольких таблиц выполняется с помощью SQL-запроса, начинающегося с ключевого слова (оператора) `SELECT`. Критерий (условие) выбора данных формулируется в выражении SQL-запроса вслед за ключевым словом `WHERE`. Результатом выполнения такого запроса является временная таблица (возможно, пустая, если нет данных, удовлетворяющих условию запроса). Функция PHP, выполняющая запрос, возвращает указатель на полученную в результате временную таблицу. Этот указатель затем используется в PHP-программе для доступа к данным с помощью специальных функций.

Довольно часто данные из временной таблицы читают в массив, а затем его обрабатывают в программе (например, выводят на экран, изменяют значения, производят вычисления и т. п.). Если требуется изменить значения полей таблицы, добавить новую или удалить имеющуюся запись, то формируют строку с соответствующим SQL-запросом, а затем передают ее в качестве параметра функции, выполняющей запрос (например, `mssql_query()`).

Чтение в массив записей временной таблицы, полученной в результате выполнения SQL-запроса, производится функциями, в имени которых находится слово `fetch` (достать). Например, функция `mssql_fetch_assok()` читает текущую запись таблицы БД MS SQL Server в массив с символьными индексами (ключами). При этом значения индексов совпадают с именами соответствующих полей таблицы. `mssql_fetch_row()` создает аналогичный массив, но с числовыми индексами, значения которых совпадают с номерами полей таблицы. Для MySQL существуют аналогичные функции: `mysql_fetch_assok()`, `mysql_fetch_row()`.

В PHP имеется расширение SQLite, которое позволяет сохранять данные в текстовых файлах и манипулировать ими посредством языка SQL. Если объем ваших данных относительно невелик, структура базы данных проста, а требования по защите данных невысоки, то применение SQLite — хорошее решение. При этом вам не нужны ни СУБД, ни знание команд для работы с файлами. Достаточно знать только SQL. Так например, гостевую книгу или форум для личного сайта вполне можно сделать на основе SQLite.

SQLite сочетает хранение табличных данных в текстовых файлах с возможностью работы с ними посредством SQL. В Windows расширению SQLite соответствует библиотека php_sqlite.dll. Несмотря на то, что база данных SQLite хранится в виде текстового файла, ее создание, редактирование и другие манипуляции следует выполнять не с помощью текстового редактора, а посредством специальных PHP-функций.

Чтобы открыть базу данных (установить соединение с базой данных), необходимо применить функцию:

```
sqlite_open(имя_БД);
```

Данная функция возвращает указатель на открытую базу данных или false, если операция не удалась. Если указанная в качестве параметра база данных не существует, то предпринимается попытка ее создать. Имя базы данных — абсолютный или относительный путь к файлу. Например, `sqlite_open("/db/mydb")` попытается открыть или создать файл базы данных `mydb`, расположенный в папке `db`, которая находится в домашнем каталоге Web-сервера.

После открытия файла базы данных можно использовать SQL-выражения, которые выполняются с помощью функций:

- `sqlite_query(SQL_запрос)` — для выполнения SQL-запросов, возвращающих данные (например, `SELECT …`); при успешном выполнении запроса возвращает указатель на временную таблицу данных, в противном случае — false;
- `sqlite_exec(SQL_запрос)` — для выполнения SQL-запросов, не возвращающих данные (например, `DELETE…`); при успешном выполнении запроса возвращает true, в противном случае — false.

В частности, после создания новой базы данных следует выполнить SQL-запрос на создание таблицы, в которой будут храниться собственно данные.

По завершении работы с базой данных ее следует закрыть с помощью функции

```
sqlite_close(идентификатор_БД).
```

Далее в качестве примера приведен код PHP-программы, который выполняет следующие действия:

1. Открывается база данных `mydatabase`. Если она не существует, то предпринимается попытка создать ее. Идентификатор базы данных присваивается переменной `$db`.
2. Создается таблица `mytable` с тремя столбцами: `first_name`, `last_name` и `zarplata`. Перед вызовом функции выполнения запроса `sqlite_query()` использован оператор `@` подавления вывода сообщений об ошибке. Дело в том, что если таблица `mytable` уже существует, то запрос на ее создание не выполнится, но будет выведено соответствующее сообщение, которое в данном случае нежелательно. При первом выполнении программы таблица должна быть создана, а при последующих — нет.
3. В таблицу `mytable` добавляются две записи.
4. Из таблицы `mytable` выбираются все имеющиеся в ней записи.
5. В цикле с помощью функции `sqlite_fetch_array()` и оператора `echo` данные выборки отображаются в окне браузера. Функция `sqlite_fetch_array()` возвращает значения столбцов текущей записи таблицы как массив и переводит указатель на следующую запись.
6. База данных с идентификатором `$db` закрывается.

При первом выполнении данная программа выведет в окно браузера две строки данных.

Описанный в общих чертах код программы выглядит следующим образом:

```
<?php
/* Открытие БД: */
$db=sqlite_open("mydatabase") or exit("Не удалось открыть
БД");
/* Создание таблицы: */
$strsql="CREATE TABLE mytable1 (
first_name varchar(20),
```

```
       last_name varchar(20),
       zarplata numeric(8,2))";
$result = @sqlite_query($db,$strsql);      /* @ — для подавления вывода
                                                    сообщений
                                                об ошибке */
/* Вставка записи: */
$strsql="INSERT INTO mytable1 VALUES ('Петр', 'Петров',
1500.50)";
$result=sqlite_query($db,$strsql);
/* Вставка еще одной записи: */
$strsql="INSERT INTO mytable1 VALUES ('Иван','Иванов',
3500)";
$result=sqlite_query($db,$strsql);
/* Выборка всех записей из таблицы: */
$strsql="SELECT *FROM mytable1";
$result=sqlite_query($db,$strsql);
/* Отображение записей: */
while ($row = sqlite_fetch_array($result)){
       echo   $row['first_name']." ".
              $row['last_name']." - ".
              $row['zarplata']."<br>";
}
sqlite_close($db);    // закрытие БД
?>
```

Получив с помощью SQL-запроса набор записей, можно воспользоваться функциями для перемещения по записям:

- `sqlite_next()` — переход к следующей записи; возвращает true в случае успеха и false, если предпринята попытка перейти к несуществующей записи;
- `sqlite_prev()` — переход к предыдущей записи; возвращает true в случае успеха и false, если предпринята попытка перейти к несуществующей записи;
- `sqlite_rewind()` — переход к первой записи; возвращает false, если набор не содержит записей, и true — в противном случае;

- `sqlite_current(указатель)` — возвращает текущую запись в указанном наборе как массив; если указатель текущей записи находится за последней записью набора, то функция возвращает false;
- `sqllite_seek(указатель, номер_записи)` — переход к записи с указанным номером (нумерация начинается с 0) в указанном наборе; если указанная запись отсутствует в наборе, то возвращается false, иначе — true.

В данном примере предполагается, что соединение с базой данных уже установлено.

Примечание

В операторе `SELECT` для обозначения всех столбцов таблицы можно использовать символ (`*`). Но в случае использования встроенного SQL в программах рекомендуется не использовать этот символ, а явно указывать требуемые столбцы. Дело в том, что пользователь или вы, как разработчик приложения, можете добавить или удалить столбцы уже после создания приложения. Если использовался символ (`*`), приложение может получить не тот набор столбцов, на который оно было первоначально рассчитано.

Глава 10

Постоянно хранимые модули

В предыдущих главах SQL рассматривался как язык манипулирования данными, лишенный каких бы то ни было собственных средств управления вычислениями (операторы условного перехода, цикла и т. п.), которыми обладают все процедурные языки. Поэтому при необходимости организовать довольно сложную обработку данных SQL-выражения вставляются в коды программ на процедурных языках. При этом SQL-коды делают то, что им удается лучше всего (например, выборка данных), а сложную, многоэтапную обработку данных выполняют коды на процедурных языках. Такова обычная практика применения SQL в приложениях баз данных. Вместе с тем, в дополнение к SQL-92 был выпущен SQL-92/PSM (Persistent Stored Modules — постоянно хранимые модули). Это дополнение легло в основу одного из разделов SQL:1999 и SQL:2003, благодаря чему стало возможно решать многие задачи, не выходя за рамки SQL. А именно в SQL появилась возможность выполнять целые блоки SQL-выражений как одно выражение (составную команду), объявлять и использовать в дальнейшем переменные, применять операторы управления вычислениями, определять процедуры и функции, которые можно вызывать в SQL-запросах. Таким образом, многое стало возможным без привлечения внешних программных средств.

Вы можете написать на языке SQL процедуры и функции, которые могут содержать объявления переменных и составные команды SQL.

Коды этих процедур и функций могут содержать управляющие структуры, такие как операторы условного перехода и циклы, а также обычные SQL-выражения. Функции и процедуры задаются в рамках так называемого модуля, который представляет собой контейнер для их размещения. Модуль создается специальной командой SQL и сохраняется в метаданных базы, т. е. становится ее компонентом, подобно таблицам, индексам и т. д. Таким образом, однажды создав модуль с процедурами и функциями, вы можете затем в частных SQL-запросах использовать их вызовы.

В данной главе мы начнем с того, что так или иначе используется в процедурах, функциях и, в конечном счете, в модулях. К непосредственному рассмотрению модулей мы перейдем лишь в конце главы, а начнем с самого простого — составных команд SQL.

10.1. Составные команды

В SQL:1999 и SQL:2003 можно использовать составные команды, которые включают в себя обычные команды SQL, выполняемые в один прием. Это означает, что все команды, входящие в составную команду, вместе передаются на сервер базы данных, там обрабатываются, в результате чего возвращается один ответ, а не множество ответов на каждую из отдельных команд. Это позволяет снизить нагрузку на сеть.

Все команды, входящие в одну составную команду, обрамляются ключевыми словами `BEGIN` (начало) и `END` (конец). Например, чтобы вставить данные в несколько связанных таблиц, следует использовать такой синтаксис:

```
BEGIN
    INSERT INTO Продажи (Товар, ID_клиента, Сумма_заказа)
        VALUES ("Телевизор", 1024, 5800);
    INSERT INTO Клиенты (ID_клиента, Имя)
        VALUES (1024, "Иванов Иван Иванович");
END;
```

Чтобы вставить данный код в программу, например, на процедурном языке, достаточно перед составной командой написать `EXEC SQL`:

```
void main() {      // Программа на языке C
    EXEC SQL
    BEGIN
      INSERT INTO Продажи (Товар, ID_клиента, Сумма_заказа)
          VALUES ("Телевизор", 1024, 5800);
      INSERT INTO Клиенты (ID_клиента, Имя)
          VALUES (1024, "Иванов Иван Иванович");
    END;
    /* Здесь может быть, например, проверка наличия
       ошибок выполнения составной SQL-команды */
}
```

10.1.1. Атомарность

Выполнение обычной (не составной команды) SQL может быть успешным или нет. Если команда по каким-либо причинам не смогла выполниться, то в базе данных не происходит никаких изменений. Таково общее правило, выполняемое СУБД. В случае составных команд это не так.

Обратимся к приведенному ранее примеру. В нем производится запись сведений о клиенте и покупке, которую он совершил. Эти сведения записываются в две таблицы: `Клиенты` (справочник клиентов) и `Продажи` (сведения о приобретенном товаре и идентификаторе клиента). Предположим, что запись в таблицу `Продажи` прошла успешно, а в таблицу `Клиенты` — нет. Тогда получится, что зарегистрирована продажа товара отсутствующему клиенту. Быть может, в данном конкретном примере это не так уж и страшно, однако бывают ситуации, чреватые серьезными последствиями. Чтобы подобное не происходило, составную команду можно сделать атомарной (неделимой). Такая команда либо выполняется целиком, либо не выполняется совсем. Обычные команды являются изначально атомарными: если обычная команда не выполнилась, то с базой данных ничего не происходит.

Чтобы придать свойство атомарности составной команде, следует за ключевым словом BEGIN дописать слово ATOMIC, как в следующем примере:

```
void main() {
    EXEC SQL
    BEGIN ATOMIC
      INSERT INTO Продажи (Товар, ID_клиента, Сумма_заказа)
          VALUES ("Телевизор", 1024, 5800);
      INSERT INTO Клиенты (ID_клиента, Имя)
          VALUES (1024, "Иванов Иван Иванович");
    END;
    /* Здесь может быть, например, проверка наличия
       ошибок выполнения составной SQL-команды */
}
```

Если при выполнении атомарной составной команды произойдет сбой на какой-либо входящей в нее простой команде, то база данных вернется к исходному состоянию (говорят, что произойдет откат).

10.1.2. Переменные

В составных командах можно объявлять переменные и присваивать им значения. Переменные могут затем использоваться в простых командах внутри составной команды, но они уничтожаются после ее завершения. Иначе говоря, переменные в SQL являются локальными переменными той составной команды, в которой они объявлены.

Переменная объявляется с помощью выражения вида:

```
DECLARE имяПеременной тип;
```

Например:

```
DECLARE myvar NUMERIC;
```

После объявления переменной ей можно присвоить значение. Для этого используется следующий синтаксис:

```
SET имяПеременной = значение;
```

Например:

```
SET vname = "Иванов Иван Иванович";
SET summa = summa * 1.2;
SET S_circle = 3.14 * :radius * :radius;
```

В следующем примере составная команда содержит объявления переменной и курсора, операторы открытия курсора, чтения значения столбца таблицы в переменную и закрытия курсора:

```
BEGIN
    DECLARE vname CHARACTER(25);
    DECLARE mycusor CURSOR FOR
        SELECT Имя FROM Клиенты;
    OPEN mycursor;
    FETCH FROM mycursor INTO vname;
    CLOSE mycursor;
END;
```

Обратите внимание, что имя переменной в операторе `FETCH` указано без ведущего двоеточия. Двоеточие используется в том случае, если переменная была объявлена не внутри составной SQL-команды, а в программе на процедурном языке *(см. разд. 9.4)*.

10.1.3. Обработка состояния

Результат выполнения команды SQL может быть успешным, неудачным или неопределенным. При выполнении команды сервер базы данных обновляет значение специальной переменной `SQLSTATE` (статус или состояние SQL). Эта переменная содержит информацию об успешном или неуспешном выполнении команды. В случае неудачи переменная `SQLSTATE` содержит код ошибки.

Переменная `SQLSTATE` содержит 5 символов, первые два из которых указывают, выполнена команда SQL успешно, неудачно или результат не ясен. Далее приведен список возможных вариантов:

- `'00'` — успешное завершение;
- `'01'` — случилось что-то незапланированное, однако не известно, произошла ошибка или нет (`SQLWARNING` — предупреждение);

- '02' — в результате выполнения команды не получено никаких данных, например, запрос вернул пустую таблицу (NOT FOUND — не найден);
- другое значение — сгенерировано исключение.

Первые два из пяти символов в значении переменной SQLSTATE называют также *классом кода состояния*. Любой класс кода, отличный от '00', '01' или '02', означает, что в программе имеются ошибки. Код ошибки представлен последними тремя символами.

В серьезных программах значение переменной SQLSTATE обычно контролируется после выполнения каждой команды SQL:

- при получении кода '00' можно продолжать выполнение программы;
- при получении кода '01' или '02' требуется решить, как программа должна на это отреагировать. Возможно, что эти сообщения допустимы, и тогда следует продолжить выполнение программы. В противном случае необходимо перейти к выполнению специальной процедуры, которая обрабатывает подобные ситуации;
- получение какого-то другого кода означает наличие ошибок. В данном случае следует перейти к процедуре обработки исключений. Если возможны различные состояния SQLSTATE, то для каждого из них может потребоваться своя процедура обработки.

Для контроля за состоянием выполнения команд SQL необходимо задать (объявить) так называемый *обработчик состояний* (процедуру обработки исключений). Его объявление можно разместить в составной команде. В объявлении указывается состояние, при возникновении которого вызывается обработчик, а также действие, которое выполняется обработчиком. Само действие может быть простой или составной командой. После успешного выполнения обработчика выполняется некоторое последействие, называемое эффектом. Теперь рассмотрим задание обработчика более подробно.

Обработчик объявляется следующим образом:

```
DECLARE имяОбработчика CONDITION
    FOR состояние;
```

Обычно имя обработчика выбирают в соответствии с состоянием, которое он должен обработать (например, `constraint_error` — нарушение ограничения целостности данных).

Далее перечислены состояния, которые можно задать для обработчика:

- `SQLSTATE VALUE 'xxyyy'` — указывается значение переменной `SQLSTATE`, состоящей из пяти символов `'xxyyy'`;
- `SQLEXCEPTION` — класс переменной `SQLSTATE`, отличный от `'00'`, `'01'` или `'02'` (исключение);
- `SQLWARNING` — класс переменной `SQLSTATE '01'` (предупреждение);
- `NOT FOUND` — класс переменной `SQLSTATE '02'` (не найден).

Чтобы указать эффект и действие обработчика, используется следующая конструкция:

```
DECLARE эффект HANDLER
    FOR имяОбработчика
    действие;
```

Возможны следующие три эффекта обработчика:

- `CONTINUE` (продолжение) — выполнение команды, следующей за командой, которая инициировала обработчик. Данный эффект применяется в случае, когда обработчик может устранить любую проблему, которая его вызвала;
- `EXIT` (выход) — выход из составной команды, содержащей обработчик, и переход к выполнению следующей команды. Данный эффект применяется в случае, когда обработчик не может исправить положение дел, однако нет необходимости отменять изменения, произведенные составной командой;
- `UNDO` (отмена, откат) — отмена всех предыдущих команд, входящих в составную команду, и переход к выполнению следующей команды. Данный эффект позволяет вернуться в состояние,

которое было до начала выполнения составной команды. При использовании эффекта UNDO необходимо, чтобы составная команда была атомарной, т. е. после ключевого слова BEGIN должно следовать ключевое слово ATOMIC *(см. разд. 10.1.1)*.

В следующем примере используется обработчик события, связанного с нарушением некоторого ограничения целостности данных, которое может возникнуть при выполнении оператора добавления записи INSERT. Так, при попытке добавить запись с первичным ключом, уже имеющимся в таблице, в переменную SQLSTATE записывается значение '23000'. В результате возникнет состояние, на которое реагирует обработчик constraint_error. Он отменяет все изменения, произведенные командами INSERT. Команда RESIGNAL (действие) передает управление той процедуре, которая вызвала процедуру обработчика. В случае успешного выполнения обеих команд INSERT выполняется команда, следующая за ключевым словом END.

Вот код описанного примера:

```
BEGIN ATOMIC
    DECLARE constraint_error CONDITION
        FOR SQLSTATE VALUE '23000';
    DECLARE UNDO HANDLER
        FOR constraint_error
        RESIGNAL;
    INSERT INTO Продажи (Товар, ID_клиента, Сумма_заказа)
        VALUES (:article, :customer_id, :summa);
    INSERT INTO Клиенты (ID_клиента, Имя)
        VALUES (:customer_id, :name);
END;
```

10.2. Операторы условного перехода

Во всех языках программирования основную управляющую структуру реализует оператор условного перехода. Этот оператор имеет тот или иной синтаксис, в зависимости от конкретного

языка, но между их реализациями много общего. В SQL возможны два оператора условного перехода:

- `IF...THEN...ELSE...END IF`;
- `CASE...END CASE`.

10.2.1. Оператор *IF*

В SQL синтаксис оператора условного перехода `IF` такой:

```
IF условие
THEN действие1
ELSE действие2
END IF
```

Условие в операторе `IF` представляет собой некоторое логическое выражение, которое может быть истинным или ложным.

По-русски этот оператор можно было бы записать следующим образом:

```
ЕСЛИ условие
ТО действие1
ИНАЧЕ действие2
КОНЕЦ
```

Если *условие* истинно, то выполняется *действие1*, в противном случае выполняется *действие2*. Например:

```
IF vname = 'Иванов Иван Иванович'
THEN
    UPDATE Клиенты
        SET Имя = 'Иванов Иван Иванович'
        WHERE ID_клиента = 12345;
ELSE
    DELETE FROM Клиенты
        WHERE ID_клиента = 12345;
END IF;
```

Оператор `ELSE` не является обязательным.

10.2.2. Оператор *CASE ... END CASE*

Если в зависимости от условия возможны два варианта развития событий, то оператор IF подходит наилучшим образом. В случае нескольких вариантов удобнее использовать оператор CASE. При этом выражение условия в операторе CASE может быть и не логическим. Существуют два варианта синтаксиса оператора CASE:

- оператор CASE со значениями;
- оператор CASE с условиями поиска.

Примечание

Оператор CASE применяется также и в условии выборки данных в SQL-выражениях *(см. разд. 4.4)*. Однако там заключительным ключевым словом является просто END, а не END CASE. Кроме того, оператор в выражении поиска возвращает некоторое значение, а рассматриваемый здесь оператор — нет. В остальном синтаксис операторов CASE как части SQL-выражения и как управляющей структуры в модуле совпадает.

Оператор *CASE ... END CASE* со значениями

В данном варианте оператора CASE ... END CASE вычисляется некое выражение (в частности, просто рассматривается значение некоторой переменной), и, в зависимости от имеющегося значения, указываются действия, которые следует выполнить. Например:

```
CASE vname
    WHEN 'Иванов Иван Иванович'
    THEN
        INSERT INTO Клиенты (ID_клиента, Имя)
            VALUES (:custID, vname);
    WHEN 'Петров Петр Петрович'
    THEN
        DELETE FROM Клиенты
            WHERE ID_клиента = 12345;
    ELSE
        INSERT INTO Контакты (ID_клиента, Имя, Адрес)
```

```
            VALUES (:custID, vname, :address);
END CASE;
```

Оператор `ELSE` не является обязательным. Если он указан в операторе `CASE`, то его команды выполняются тогда, когда значение условия (в примере это значение переменной `vname`) не соответствует ни одному из перечисленных в операторах `WHEN`. Если же оператор `ELSE` не указан, а значение условия не соответствует ни одному значению в операторах `WHEN`, то генерируется исключение, которое можно обработать специальным образом *(см. разд. 10.1.3)*.

Оператор *CASE ... END CASE* с условиями поиска

Данный вариант оператора `CASE ... END CASE` похож на первый. Отличие состоит в том, что используются несколько условных выражений, а не одно. Например:

```
CASE
    WHEN vname
        IN ('Иванов Иван Иванович', 'Петров Петр Петрович',
        'Михайлов Михаил Михайлович')
    THEN
        INSERT INTO Клиенты (ID_клиента, Имя)
            VALUES (:custID,vname);
    WHEN vregion
        IN ('Северо-Запад', 'Москва')
    THEN
        INSERT INTO Клиенты (ID_клиента, Имя, Регион)
            VALUES (:custID,vname, vregion);
    ELSE
        INSERT INTO Контакты (ID_клиента, Имя, Адрес, Регион)
            VALUES (:custID, vname, :address, vregion);
END CASE;
```

10.3. Операторы цикла

Циклическое (повторяющееся) выполнение набора команд SQL можно организовать с помощью нескольких специальных операторов под общим названием операторов цикла:

- `LOOP...END LOOP`;
- `WHILE...DO...END WHILE`;
- `REPEAT...UNTIL...END REPEAT`;
- `FOR...DO...END FOR`;
- `ITERATE`.

10.3.1. Оператор *LOOP...END LOOP*

Оператор `LOOP ... END LOOP` (петля) позволяет многократно выполнить некоторую последовательность команд SQL. После выполнения последней команды в этой последовательности управление снова передается первой команде, и все повторяется сначала.

В следующем примере в таблицу `Клиенты` добавляются новые записи. При этом в каждую новую запись столбца `ID_клиента` заносится значение, отличающееся от предыдущего на 1. Поскольку переменная `customerID` объявлена в коде SQL, то в операторе `INSERT` она используется без ведущего двоеточия. Код данного примера следующий:

```
DECLARE customerID NUMERIC;
SET customerID = 12345;
LOOP
SET customerID = customerID + 1;
INSERT INTO Клиенты (ID_клиента)
    VALUES (customerID);
END LOOP;
```

Обратите внимание, что данный цикл является "бесконечным" и будет продолжаться до тех пор, пока остается место на диске. Чтобы прекратить работу цикла, используется оператор `LEAVE` (оставлять, покидать). Справа от оператора `LEAVE` указыва-

ется метка, например, `LEAVE mylabel`. Как только вычислительный процесс доходит до этого оператора, управление передается следующей команде, расположенной после команды, помеченной этой меткой.

В следующем примере создаются 100 новых записей в таблице `Клиенты`, после чего цикл завершается:

```
DECLARE customerID NUMERIC;
SET customerID = 0;
mylabel:
LOOP
    SET customerID = customerID + 1;
    IF customerID > 100
    THEN
        LEAVE mylabel;
    END IF;
    INSERT INTO Клиенты (ID_клиента)
        VALUES (customerID);
END LOOP mylabel;
```

10.3.2. Оператор *WHILE ... DO ... END WHILE*

В операторе `WHILE ... DO ... END WHILE` (пока ... делать ...) после ключевого слова `WHILE` указывается условие. Пока оно истинно, будут выполняться выражения, расположенные между ключевыми словами `DO` и `END WHILE`. Если условие становится ложным, то выражения внутри оператора цикла не выполняются.

Следующий код делает то же, что и рассмотренный в конце *разд. 10.3.1*:

```
DECLARE customerID NUMERIC;
SET customerID = 0;
WHILE customerID < 100 DO
    SET customerID = customerID + 1;
    INSERT INTO Клиенты (ID_клиента)
        VALUES (customerID);
END WHILE;
```

10.3.3. Оператор *REPEAT ... UNTIL ... END REPEATE*

Оператор `REPEAT ... UNTIL ... END REPEATE` (повторять ... до тех пор, пока ...) похож на оператор `WHILE ... DO ... END WHILE`. Отличие состоит лишь в том, что условие продолжения цикла проверяется не в начале, а в конце оператора после ключевого слова `UNTIL`. Таким образом, оператор `REPEAT ... UNTIL ... END REPEATE` всегда обеспечивает выполнение содержащихся в нем выражений хотя бы один раз.

Следующий код делает то же, что и рассмотренные примеры в предыдущих разделах:

```
DECLARE customerID NUMERIC;
SET customerID = 0;
REPEATE
    SET customerID = customerID + 1;
    INSERT INTO Клиенты (ID_клиента)
        VALUES (customerID);
UNTIL customerID < 100;
END REPEATE;
```

10.3.4. Оператор *FOR ... DO ... END FOR*

В SQL оператор цикла `FOR ... DO ... END FOR` (для ... делать ...) не аналогичен оператору цикла со счетчиком количества повторений в процедурных языках. В SQL этот оператор позволяет построчно обрабатывать данные без использования процедурного (базового) языка. Точнее, он выполняет следующие операции:

- объявление и открытие курсора;
- выполнение команд тела цикла, указанных между ключевыми словами `DO` и `END FOR`, для каждой записи курсора;
- закрывает курсор.

В следующем примере происходит обновление каждой записи таблицы `Клиенты` путем ввода в ее столбец `Примечание` строки `'Новый клиент'`:

```
FOR customerID AS mycursor CURSOR FOR
    SELECT ID_клиента FROM Клиенты
DO
    UPDATE Клиенты
        SET Примечание = 'Новый клиент'
        WHERE CURRENT OF mycursor;
END FOR;
```

Обратите внимание, что данный способ применения курсоров отличается от методов, описанных в *гл. 9*.

Другой вариант применения оператора `FOR` приведен далее:

```
BEGIN ATOMIC
    DECLARE prod CHAR VARYING (400)
        DEFAULT '';
    DECLARE x ROW;
    FOR x AS SELECT * FROM Продажи T
        WHERE T.ID_Клиента = 12345
    DO
        IF x.Товар <> ''
        THEN
            SET prod=prod || ',';
        END IF;
        SET prod=prod || x.Товар;
    END FOR;
    UPDATE Клиенты
        SET Примечание = prod
        WHERE ID_клиента = 12345;
END;
```

В данном примере создается список всех товаров, приобретенных клиентом с идентификатором `12345`. Этот список представляется в виде строки, в котором наименования товаров разделены запятой. Затем происходит обновление таблицы `Клиенты`: в столбец

Примечание для клиента с идентификатором 12345 заносится список всех приобретенных им товаров.

10.3.5. Оператор *ITERATE*

Оператор ITERATE (повторить) применяется в теле циклов, заданных операторами цикла, которые были рассмотрены ранее. Он позволяет изменить последовательность выполнения команд в теле цикла. Если условие в операторе цикла истинно или не задано, оператор ITERATE начинает новую итерацию цикла (управление передается в начало цикла). Если же условие ложно или неопределенно, выполняются выражения, следующие за оператором ITERATE, и цикл прекращается. Оператор ITERATE применяется вместе с меткой для указания точки перехода (подобно оператору LEAVE в *разд. 10.3.1*).

Например:

```
DECLARE vresult CHAR (10);
DECLARE customerID NUMERIC;
SET vresult = '';
SET customerID = 0;
mylabel:
WHILE customerID < 100 DO
    SET customerID = customerID + 1;
    INSERT INTO Клиенты (ID_клиента)
        VALUES (customerID);
    ITERATE mylabel;
    SET vresult = 'Готово';
END WHILE;
```

В данном примере цикл выполняется до тех пор, пока переменная customerID не станет равной 99 (оператор ITERATE передает управление в начало оператора WHILE, точнее точке с меткой mylabel). При этом в цикле выполняется только оператор добавления записи INSERT. В следующей итерации переменная customerID станет равной 100 и тогда оператор ITERATE прекратит выполнение цикла, переменной vresult будет присвоено зна-

значение `'Готово'`, а затем вычислительный процесс перейдет к следующей за циклом команде.

10.4. Хранимые процедуры и функции

Процедуры и функции являются подпрограммами, т. е. некими блоками выражений SQL, которые можно многократно вызывать в SQL-запросах и/или из других процедур и функций тогда, когда это понадобится. Между процедурами и функциями много общего. И те, и другие выполняют определенные действия и могут принимать параметры, т. е. информацию из внешних программных единиц, которая может использоваться кодом этих процедур и функций. Отличие между процедурами и функциями заключается в способе их вызова. Выражение вызова функции, в отличие от выражения вызова процедуры, может участвовать в других выражениях. Так, например, вызов функции `SIN(x)`, вычисляющей синус значения `x` и возвращающей результат этого вычисления, может участвовать в качестве элемента в более сложном выражении, например, `5 + SIN(120)`. Точнее говоря, в выражении может присутствовать вызов функции. Процедура, в отличие от функции, не может участвовать в выражениях.

Хранимые процедуры и функции сохраняются на сервере базы данных вместе с другими метаданными. Когда вы вызываете процедуру или функцию, сервер базы данных выполняет ее код и, если это задано, возвращает результат клиенту. Процедуры и функции должны быть сначала определены и лишь после этого они могут быть использованы в запросе или приложении путем их вызовов.

Определение процедуры в SQL имеет следующий синтаксис:

```
CREATE PROCEDURE имяПроцедуры
    (IN имяПараметра1 типПараметра1,
     IN имяПараметра2 типПараметра2,
     ...
```

```
      IN имяПараметраN типПараметраN,
      OUT ВыходнойПараметр)
BEGIN [ATOMIC]
    Операторы SQL
END;
```

Здесь ключевое слово IN обозначает, что следующее за ним выражение является входным параметром, а ключевое слово OUT — выходным (возвращаемым) параметром. Процедуры могут и не иметь входных и/или выходных параметров, т. е. выражения IN и OUT не являются обязательными. Параметры могут задаваться своими именами. Если процедура должна вернуть в вызывающую программу какое-то значение как значение переменной (параметра), то эта переменная должна быть объявлена во внешней программе. После указания входных и/или выходных параметров следует код тела процедуры, который заключается в ключевые слова BEGIN и END. Если этот код представляет собой составную команду, содержащую несколько команд SQL, то после ключевого слова BEGIN следует добавить ключевое слово ATOMIC *(см. разд. 10.1.1)*.

В следующем примере создается процедура, которая обновляет таблицу Клиенты для некоторых клиентов. Процедура не должна возвращать никакого значения.

```
CREATE PROCEDURE updateCustomer
    (IN vcustomerid INTEGER,
     IN vName       CHAR(20),
     IN vNote       CHAR(120))
BEGIN ATOMIC
    UPDATE Клиенты
    SET Имя = vName, Примечание = vNote
    WHERE ID_клиента = vcustomerid;
END;
```

Чтобы вызвать процедуру, применяется команда CALL. Например, следующее выражение вызывает рассмотренную процедуру

updateCustomer, передавая ей параметры, указывающие, какие значения следует обновить в таблице Клиенты:

```
CALL updateCustomer(1025, 'Иванов Иван Иванович', 'Очень
важный клиент');
```

Хранимая функция создается аналогично процедуре. Особенность заключается в том, что входные параметры непосредственно указываются в круглых скобках справа от имени функции, а выходной параметр (точнее, его тип) — после ключевого слова RETURNS.

Вот пример определения функции products:

```
CREATE FUNCTION products (vcustomerid)
    RETURNS CHAR VARYING (400)
BEGIN
    DECLARE prod CHAR VARYING (400)
        DEFAULT '';
    DECLARE x ROW;
    FOR x AS SELECT * FROM Продажи T
        WHERE T.ID_Клиента = vcustomerid
    DO
        IF x.Товар <> ''
            THEN SET prod = prod || ', ';
        END IF;
        SET prod = prod || x.Товар;
    END FOR;
    RETURN prod;
END;
```

Данная функция создает список всех товаров, приобретенных клиентом с указанным идентификатором (vcustomerid), и возвращает его в виде строки, в которой наименования товаров разделены запятыми.

В отличие от процедуры, функция может участвовать в SQL-выражении. Следующий код содержит вызов функции products. Он обновляет таблицу Клиенты, занося в столбец Примечание для

клиента с идентификатором 12345 список всех приобретенных им товаров:

```
SET vcustomerid = 12345;
UPDATE Клиенты
   SET Примечание = 'Приобретено:' || products(vcustomerid)
   WHERE ID_клиента = vcustomerid;
```

10.5. Хранимые модули

Хранимые модули могут содержать множество определений процедур и/или функций. Вообще говоря, процедуры и функции, входящие в один модуль, могут быть и не связаны между собой. Язык SQL предоставляет возможность задавать различным пользователям базы данных определенные полномочия *(см. гл. 11)*. Расширение SQL, поддерживающее постоянно хранимые модули (SQL/PSM), добавляет к уже существующим правам доступа полномочие на выполнение модуля. Оно распространяется на все процедуры и функции, которые содержатся в этом модуле. Поэтому распределение процедур и функций между модулями целесообразно производить с учетом полномочий различных пользователей.

Для создания постоянно хранимого модуля служит команда SQL:

```
CREATE MODULE имяМодуля
    ...
END MODULE;
```

Вот пример модуля, содержащего описания процедуры и функции:

```
CREATE MODULE mylibrary
    CREATE PROCEDURE updateCustomer
        (IN vcustomerid INTEGER,
         IN vName       CHAR(20),
         IN vNote       CHAR(120))
    BEGIN ATOMIC
```

```
        UPDATE Клиенты
        SET Имя = vName, Примечание = vNote
        WHERE ID_клиента = vcustomerid;
    END;

    CREATE FUNCTION products (vcustomerid)
        RETURNS CHAR VARYING (400)
    BEGIN
        DECLARE prod CHAR VARYING (400)
            DEFAULT '';
        DECLARE x ROW;
        FOR x AS SELECT * FROM Продажи T
            WHERE T.ID_Клиента = vcustomerid
        DO
            IF x.Товар <> ''
                THEN SET prod = prod || ', ';
            END IF;
            SET prod = prod || x.Товар;
        END FOR;
        RETURN prod;
    END;
END MODULE;
```

Глава 11

Управление правами доступа

У базы данных может быть несколько пользователей с различными правами доступа к ее объектам. Так, одним пользователям разрешено только просматривать некоторые таблицы, другим разрешено только просматривать и добавлять новые записи, но запрещено их удалять и т. п. Некоторые пользователи могут устанавливать права доступа к объектам базы данных для других пользователей.

Разграничение прав доступа является важным средством защиты базы данных от неправильного использования содержащейся в ней информации различными категориями пользователей.

11.1. Пользователи

Пользователи баз данных могут иметь разные привилегии (полномочия, права доступа) и по объему этих привилегий разделяются на несколько категорий;

- администратор базы данных;
- владелец объектов базы данных;
- привилегированный пользователь, имеющий право предоставлять привилегии;
- привилегированный пользователь, не имеющий права предоставлять привилегии;
- рядовые пользователи.

11.1.1. Администратор базы данных

Наибольшими привилегиями обладает администратор базы данных. Он имеет все права на любые действия с базой данных, в том числе и право предоставлять и аннулировать привилегии для других пользователей. Разумеется, при этом он несет и бóльшую ответственность, поскольку легко может испортить как данные, так и саму базу, или нарушить уже налаженные правила работы с ней.

При установке СУБД на компьютер требуется ввести регистрационное имя или учетную запись пользователя (*login*) и пароль (*password*). В первый раз следует воспользоваться сведениями из руководства по установке. Зарегистрировавшись под данным именем, вы автоматически становитесь администратором базы данных, т. е. привилегированным пользователем с полным объемом прав. Теперь вы можете изменить пароль, чтобы никто, кроме вас, не мог стать ни администратором, ни даже обычным пользователем базы данных, если вы не сообщите ему пароль. Если же вы передадите пароль другому пользователю, то он может войти в систему в качестве администратора и, изменив пароль, лишить вас возможности быть администратором. Обычно у базы данных есть несколько администраторов на случай, если один из них заболеет или уволится.

Итак, администратор, имея все привилегии, первым делом "создает" пользователей базы данных, определяя для них имена и права. Он может создать и максимально привилегированного пользователя, т. е. еще одного администратора. В любой момент администратор может изменить ранее установленные права любого пользователя.

Если администратору необходимо выполнить какую-то работу, для которой не требуются все привилегии, то ему лучше войти в систему под именем пользователя с минимальными привилегиями, которые позволяют решить задачу. Это обычная мера защиты от тяжелых последствий возможных ошибок.

11.1.2. Владелец объектов базы данных

Любой пользователь, который создал некий объект базы данных (таблицу или представление), становится владельцем (*owner*) этого объекта, т. е. привилегированным пользователем, либо может назначить другого владельца.

Владелец таблицы обладает всеми привилегиями относительно этой таблицы, включая управление доступом к ней. Представление, являясь виртуальной таблицей, всегда создается на основе уже имеющихся таблиц базы данных. Причем владелец представления может и не быть владельцем исходных таблиц. Поэтому владелец представления получает относительно него привилегии, аналогичные имеющимся у него относительно исходных таблиц. Это делается для того, чтобы, создав представление, нельзя было обойти защиту исходной таблицы, владельцем которой является другой пользователь.

11.1.3. Другие пользователи

Администраторы и владельцы объектов базы данных могут предоставлять привилегии другим пользователям. Среди этих привилегий есть и право предоставления привилегий. Пользователи, не являющиеся администраторами или владельцами, а также те, кому привилегированные пользователи специально не предоставили права доступа, называются публикой (*public*). Если привилегированный пользователь предоставляет права доступа типа `PUBLIC`, то их получают все пользователи базы данных.

11.1.4. Создание пользователей

Обычно к СУБД прилагается ряд утилит, в число которых входит и утилита для создания пользователей и предоставления им прав. С ее помощью администратор может создать пользователя и наделить его определенными привилегиями.

В SQL для создания пользователя служит оператор `CREATE USER`:

```
CREATE USER имяПользователя
    [WITH
```

```
[SYSID идентификаторПользователя]
[PASSWORD 'пароль']]
[CREATEDB | NOCREATEDB]
[CREATEUSER | NOCREATEUSER]
[IN GROUP имяГруппы [, ...]]
[VALID UNTIL 'время'];
```

Здесь в квадратных скобках указаны необязательные компоненты, а вертикальной чертой разделены альтернативные варианты значений. Данный оператор позволяет создать имя пользователя, задать ему пароль (PASSWORD), дать особые права на создание базы данных (CREATEDB) и пользователей (CREATEUSER), распределить по группам (IN GROUP) и ограничить время, в течение которого данный пользователь будет актуален (VALID UNTIL).

Чтобы удалить пользователя из системы, можно применить такой оператор:

```
DROP USER имяПользователя;
```

11.2. Предоставление привилегий

Как уже отмечалось ранее, администратор базы данных имеет все привилегии на все объекты базы данных; владелец объекта также имеет на него все права. Произвольный пользователь не имеет прав до тех пор, пока они не будут предоставлены ему специально тем пользователем, у которого уже есть эти права и который обладает также правом предоставлять права другим пользователям. Это рекурсивное определение возможностей назначать привилегии. На практике эта рекурсия "раскручивается" следующим образом. Сначала администратор создает пользователей (точнее, их имена) и предоставляет им некоторые права. Если кто-то из уже созданных пользователей имеет привилегию передавать права, то он может, создав таблицу или представление, передать права на нее другим пользователям. И так далее.

Предоставление привилегий осуществляется с помощью оператора GRANT (предоставлять, разрешать), который имеет такой синтаксис:

```
GRANT списокПрав
    ON объект
    TO списокПользователей
    [WITH GRANT OPTION];
```

Здесь необязательные (в квадратных скобках) ключевые слова WITH GRANT OPTION (с правом передачи) применяются, если пользователям предоставляется право передавать права.

Права в списке прав оператора GRANT разделяются запятыми. Если необходимо предоставить все права, то вместо списка можно указать ключевые слова ALL PRIVILEGES (все привилегии).

В SQL:2003 права могут принимать следующие значения:

- SELECT — право просмотра;
- DELETE — право удаления записей;
- INSERT [(*списокСтолбцов*)] — право добавления новых записей с установкой значений для указанных столбцов; если имена столбцов не указаны, то для всех столбцов;
- UPDATE [(*списокСтолбцов*)] — право изменения значений указанных столбцов; если имена столбцов не указаны, то предполагаются все столбцы;
- REFERENCES [(*списокСтолбцов*)] — право доступа к таблице, на которую ссылается данная таблица;
- USAGE — право на домены, наборы символов, сопоставления и трансляции;
- UNDER — право на структурированные типы данных;
- TRIGGER — право на использование триггеров;
- EXECUTE — право на выполнение внешней программы.

Объект в операторе GRANT может принимать следующие значения:

- [TABLE] *имяТаблицы*;
- DOMAIN *имяДомена*;

- COLLATION *имяСопоставления*;
- CHARACTER SET *имяСимвольногоНабора*;
- TRANSLATION *имяТрансляции*;
- TYPE *схематически_обозначенный_пользовательский_тип*;
- SEQUENCE *имяГенератораПоследовательности*
  спецификаторУказателяШаблона.

Список пользователей в операторе GRANT состоит из регистрационных имен пользователей, разделенных запятыми. Вместо списка можно указать ключевое слово PUBLIC (публика). В этом случае права, указанные в операторе GRANT, получают все пользователи базы данных.

11.2.1. Роли и группы

Если пользователей базы данных немного, то при предоставлении им прав можно использовать простые списки их имен. В противном случае назначение и контроль за правами становятся довольно трудоемкими. Удобным выходом из положения является использование ролей или групп пользователей. В SQL:2003 используется понятие роли (*role*), однако его нет в основном стандарте, а потому может не быть в некоторых реализациях. Перед использованием ролей необходимо убедиться, что их поддерживает ваша СУБД. Многие СУБД применяют понятие группы пользователей, которое аналогично понятию роли.

При использовании роли (группы) права назначаются для роли (группы), а затем для каждого пользователя указывается, какую роль он играет или в какую группу входит. Как роль, так и группа характеризуются именем.

Для создания роли используется такой синтаксис:

```
CREATE ROLE имяРоли;
```

После создания роли она назначается пользователям:

```
GRANT имяРоли TO списокПользователей;
```

Назначение прав для роли производится с помощью следующего выражения:

```
GRANT списокПрав
    ON объект
    TO имяРоли
    [WITH GRANT OPTION];
```

Если реализация SQL поддерживает понятие не роли, а группы, то синтаксис оператора для создания группы может быть следующим:

```
CREATE GROUP имяГруппы WITH списокПользователей;
```

В следующих разделах рассмотрим несколько примеров предоставления различных привилегий.

11.2.2. Право просмотра данных

Для предоставления права просмотра таблицы или представления, например, `Продажи` всем пользователям базы данных можно воспользоваться следующим оператором:

```
GRANT SELECT
    ON Продажи
    TO PUBLIC;
```

Таблица `Продажи` может иметь столбец, например, `Цена`, который нежелательно предоставлять всем пользователям. В этом случае рекомендуется на основе таблицы `Продажи` создать представление, в котором нет столбцов, просмотр которых вы хотели бы запретить:

```
CREATE VIEW Продажи_вид1 AS
    SELECT Товар, Количество, Описание
    FROM Продажи;
GRANT SELECT
    ON Продажи_вид1
    TO PUBLIC;
```

11.2.3. Право изменять данные

Предположим, менеджеру по продажам (регистрационное имя SalesManager) разрешено изменять значения Цена в таблицах Товары и Продажи. Чтобы предоставить ему такое право, достаточно выполнить следующий оператор:

```
GRANT UPDATE (Цена)
    ON Товары, Продажи
    TO SalesManager;
```

Если менеджеру нужно разрешить изменять значения нескольких столбцов, то в операторе GRANT следует перечислить их имена через запятую, а если нужно разрешить изменять все столбцы, то стоит использовать такой оператор:

```
GRANT UPDATE
    ON Товары, Продажи
    TO SalesManager;
```

Для предоставления права добавлять записи вместо ключевого слова UPDATE нужно использовать ключевое слово INSERT. В следующем примере предоставляются права на изменение и добавление записей во все столбцы:

```
GRANT UPDATE, INSERT
    ON Товары, Продажи
    TO SalesManager;
```

11.2.4. Право удалять записи

Право на удаление записей предоставляется очень просто, например:

```
GRANT DELETE
    ON Клиенты, Сотрудники, Товары
    TO SuperManager;
```

11.2.5. Право на использование ссылок

Теперь рассмотрим пример предоставления прав на использование ссылок одних таблиц на другие. Так, две таблицы оказываются связанными, если в одной таблице задан внешний ключ,

ссылающийся на первичный ключ в другой таблице *(см. разд. 7.1.3)*. В этом случае если у пользователя есть права доступа к первой таблице, то он может получить некоторые сведения и из второй, даже если у него нет права доступа к этой таблице. Предположим, пользователю стало известно о появлении в базе данных особой таблицы `Сотрудники_секретно`, содержащей отдельный список сотрудников, например, представленных к премии или, наоборот, к увольнению. Ему также стало известно, что первичным ключом в этой таблице является столбец `ID`. Однако пользователь не имеет никаких прав на эту таблицу, даже права на просмотр. Тогда он может узнать, кто попал в секретный список, следующим образом.

Сначала он создает таблицу `mytab` со столбцом `ID`, который определяет как внешний ключ со ссылкой на таблицу `Сотрудники_секретно`:

```
CREATE TABLE mytab (
    ID INTEGER REFERENCES Сотрудники_секретно
);
```

Затем пользователь пытается добавить новые записи в свою таблицу `mytab`, подбирая значения столбца `ID`. Если выбранное им значение принимается таблицей `mytab`, то сотрудник с данным идентификатором находится в секретном списке, а в противном случае — нет. Узнав таким способом список идентификаторов сотрудников секретного списка, шпион может по другой таблице (например, `Сотрудники`) узнать фамилии тех, кто был занесен в секретный список.

Поэтому согласно стандарту SQL:2003 права на использование ссылок должны устанавливаться явно:

```
GRANT REFERENCES (ID)
    ON Сотрудники_секретно
    TO PersonManager;
```

Это не позволит использовать описанный прием взлома защиты.

11.2.6. Право на домены

При создании таблиц нередко используют определения доменов. Домены позволяют определить однотипные столбцы с одинаковыми ограничениями, находящиеся в различных таблицах *(см. разд. 7.1.1)*.

Например, можно определить домен `Bonus` (премия) с типом данных `DECIMAL (9,2)` и ограничением, согласно которому премия не может быть отрицательной и больше 15 000:

```
CREATE DOMAIN Bonus
    DECIMAL (9, 2)
    CHECK (Bonus >= 0 AND Bonus <= 15000);
```

Тогда этот домен можно применить для определения столбцов в одной или нескольких таблицах. Например:

```
CREATE TABLE Платежная_ведомость (
    ID INTEGER,
    Имя VARCHAR (25),
    Оклад DECIMAL (9,2),
    Премия Bonus
);
```

Однако при использовании доменов могут возникнуть вопросы, связанные с безопасностью. Так, верхняя граница премии может являться секретом для большинства пользователей. Однако любой пользователь может создать таблицу со столбцом, который определен через домен `Bonus`. Постепенно увеличивая значения в этом столбце, пока они принимаются таблицей, можно узнать максимальное значение (попытка ввести большее значение будет отвергнута СУБД).

Чтобы не допустить такой возможности, владелец домена (тот, кто его создал) должен явно предоставить права на его использование, например:

```
GRANT USAGE
    ON DOMAIN Bonus
    TO PersonManager, President;
```

11.2.7. Право предоставлять права

На практике нередко делают так, чтобы права доступа мог предоставлять только администратор и владельцы объектов базы данных. При этом владельцы предоставляют права только на объекты, которыми владеют. Третьи лица, получившие права доступа от администратора и/или владельцев, уже не могут предоставлять права. Во многих случаях это вполне разумное ограничение, позволяющее администратору и владельцам сохранить достаточно полный контроль над системой. Вместе с тем встречаются ситуации, в которых желательно делегировать право предоставлять свои собственные права другим пользователям (например, своим сменщикам, на период отпуска и т. п.). С этой целью в операторе `GRANT` используются ключевые слова `WITH GRANT OPTION`.

В следующем примере право обновлять и добавлять записи в таблице `Сотрудники` предоставляется менеджеру по продажам с именем `SalesManager`. Причем ему предоставляется право передавать полученные права доступа другому пользователю.

```
GRANT UPDATE, INSERT
    ON Сотрудники
    TO SalesManager
    WITH GRANT OPTION;
```

Теперь менеджер по продажам может передать свои права помощнику с именем `AssistantManager`:

```
GRANT UPDATE, INSERT
    ON Сотрудники
    TO AssistantManager;
```

11.3. Отмена привилегий

Со временем обязанности пользователей базы меняются, кто-то из них может перейти на сторону конкурента и, наконец, некоторые ранее предоставленные привилегии могут оказаться просто ошибочными с точки зрения общей защиты базы данных. В этих случаях появляется необходимость отменить привилегии.

Отмена привилегий выполняется с помощью оператора REVOKE (отмена, аннулирование):

```
REVOKE [GRANT OPTION FOR] списокПрав
ON объект
FROM списокПользователей [RESTRICT | CASCADE];
```

Здесь в квадратных скобках указаны необязательные элементы, а вертикальной чертой разделены альтернативные варианты значений. Не трудно заметить, что синтаксис оператора REVOKE аналогичен синтаксису оператора GRANT.

С помощью оператора REVOKE можно отменить указанные в списке права для перечисленных пользователей. При этом можно аннулировать все права или только некоторые. Рассмотрим значения необязательных ключевых слов:

- CASCADE (каскадно) — отменяются указанные права у перечисленных пользователей, а также у тех, кому эти пользователи успели предоставить данные права;
- RESTRICT (ограничить) — отменяются права у пользователей, которые никому другому их не предоставляли. Однако если пользователь уже успел предоставить права, указанные в этом операторе REVOKE, то оператор не будет выполнен и появится сообщение об ошибке;
- GRANT OPTION FOR (право предоставления для) — применяется, чтобы отменить у пользователя право передавать указанные права, но оставить эти права за пользователем. Если оператор REVOKE содержит GRANT OPTION FOR и CASCADE, то отменяются все права, предоставленные пользователем, а также право этого пользователя на предоставление прав. Если в REVOKE есть GRANT OPTION FOR и RESTRICT, то возможны следующие два варианта:
  - если пользователь еще не предоставил другому пользователю права, которые вы у него отменяете, оператор REVOKE выполняется, отменяя также право этого пользователя предоставлять права;

- если пользователь уже предоставил кому-нибудь хотя бы одно из отменяемых прав, то права не отменяются и возвращается код ошибки.

Следует иметь в виду, что пользователь может получить одни и те же права от различных привилегированных пользователей. Если один из них отзовет предоставленные права, у пользователя они все равно могут остаться, поскольку продолжают действовать права, предоставленные кем-то другим. При передаче прав от одного пользователя другому возникают зависимости прав: права одного пользователя зависят от того, продолжает ли ими обладать другой.

Рассмотрим еще одно применение оператора `REVOKE`, связанное с целевой задачей не отмены, а наоборот, предоставления прав. Очень часто предоставление прав многим пользователям на многочисленные объекты связано с большими объемами SQL-кода. Комбинируя операторы `GRANT` и `REVOK`, предоставляя сначала широкие права для многих, а затем ограничивая их для некоторых пользователей, можно сократить общий объем SQL-кода.

Предположим, что пользователи Иванов и Петров имеют права просматривать, добавлять, удалять и обновлять записи в таблице `Клиенты` (`ID_клиента`, `Имя`, `Адрес`, `Телефон`, `Сумма_заказов`, `Примечания`), но не могут обновлять значения столбца `ID_клиента`. Все остальные пользователи могут только просматривать записи. Данное распределение прав доступа к таблице можно выполнить с помощью следующих операторов:

```
GRANT SELECT
    ON Клиенты
    TO PUBLIC;

GRANT INSERT, DELETE
    ON Клиенты
    TO Иванов, Петров;

GRANT UPDATE
```

```
    ON Клиенты (Имя, Адрес, Телефон, Сумма_заказов,
                Примечания)
    TO Иванов, Петров;
```

Эквивалентный результат, но без длинного перечня имен столбцов получается с помощью такого кода:

```
GRANT SELECT
    ON Клиенты
    TO PUBLIC;

GRANT INSERT, DELETE, UPDATE
    ON Клиенты
    TO Иванов, Петров;

REVOKE UPDATE
    ON Клиенты (ID_клиента)
    TO Иванов, Петров;
```

Во втором варианте сначала предоставляется право обновлять все столбцы, а затем отзывается право обновлять столбец `ID_клиента`.

Приложение

Зарезервированные слова SQL

ABS
ALL
ALLOCATE
ALTER
AND
ANY
ARE
ARRAY
AS
ASENSITIVE
ASYMMETRIC
AT
ATOMIC
AUTHORIZATION
AVG
BEGIN
BETWEEN
BIGINT
BINARY
BLOB
BOOLEAN
BOTH
BY
CALL
CALLED
CARDINALITY
CASCADED
CASE
CAST
CEIL
CEILING
CHAR
CHAR_LENGTH
CHARACTER
CHARACTER_LENGTH
CHECK
CLOB
CLOSE
COALESCE
COLLATE
COLUMN
COMMIT
CONDITION
CONNECT
CONSTRAINT
CONVERT
CORR
CORRESPONDING
COUNT
COVAR_POP
COVAR_SAMP
CREATE
CROSS
CUBE
CUME_DIST
CURRENT
CURRENT_COLLATION
CURRENT_DATE
CURRENT_DEFAULT_
TRANSFORM_GROUP
CURRENT_PATH
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TRANSFORM_
GROUP_FOR_TYPE
CURRENT_USER
CURRENTJROLE
CURSOR
CYCLE
DAFAULT
DATE
DAY
DEALLOCATE
DEC
DECIMAL
DECLARE

DEFAULT
DELETE
DENSE_RANC
DEREF
DESCRIBE
DETERMINISTIC
DISCONNECT
DISTINCT
DOUBLE
DROP
DYNAMIC
EACH
ELEMENT
ELSE
END
END-EXEC
ESCAPE
EVERY
EXCEPT
EXEC
EXECUTE
EXISTS
EXP
EXTERNAL
EXTRACT
FALSE
FETCH
FILTER
FLOAT
FLOOR
FOR
FOREIGN
FREE
FROM
FULL
FUNCTION
FUSION
GET
GLOBAL
GRANT
GROUP
GROUPING
HAVING
HOLD
HOUR
IDENTITY
IN
INDICATOR
INNER
INOUT
INSENSITIVE
INSERT
INT
INTEGER
INTERSECT
INTERSECTION
INTERVAL
INTO
IS
JOIN
LANGUAGE
LARGE
LATERAL
LEADING
LEFT
LIKE
LN
LOCAL
LOCALTIME
LOCALTIMESTAMP
LOWER
MATCH
MAX
MEMBER
MERGE
METHOD
MIN
MINUTE
MOD
MODIFIERS
MODULE
MONTH
MULTISET
NATIONAL
NATURAL
NCHAR
NCLOB
NEW
NO
NONE
NORMALIZE
NOT
NULL
NULLIF
NUMERIC
OCTET_LENGHT
OF
OLD
ON
ONLY
OPEN
OR
ORDER
OUT
OUTER
OVER
OVERLAPS
OVERLAY

PARAMETER
PARTITION
PERCENT_RANK
PERCENTILE_DISC
POSITION
POWER
PRECISION
PREPARE
PRIMARY
PROCEDURE
RANGE
RANK
READS
REAL
RECURSIVE
REF
REFERENCES
REFERENCING
REGR_AVGX
REGR_AVGY
REGR_COUNT
REGR_INTERCEPT
REGR_R2
REGR_SLOPE
REGR_SXX
REGR_SXY
REGR_SYY
RELEASE
RESULT
RETURN
RETURNS
REVOKE
RIGHT
ROLLBACK
ROLLUP
ROW
ROW_NUMBER
ROWS
SAVEPOINT
SCOPE
SCROLL
SEARCH
SECOND
SELECT
SENSITIVE
SESSIONJJSER
SET
SIMILAR
SMALLINT
SOME
SPECIFIC
SPECIFICTYPE
SQL
SQLEXEPTION
SQLSTATE
SQLWARNING
SQRT
START
STATIC
STDDEV_POP
STRDDEV_SAMP
SUBMULTISET
SUBSTRING
SUM
SYMMETRIC
SYSTEM
SYSTEM_USER
TABLE
TABLESAMPLE
THEN
TIME
TIMESTAMP
TIMEZONE_HOUR
TIMEZONE_MINUTE
TO
TRAILING
TRANSLATE
TRANSLATION
TREAT
TRIGGER
TRIM
TRUE
UNION
UNIQUE
UNKNOWN
UNNEST
UPDATE
UPPER
USER
USING
VALUE
VALUES
VAR_POP
VAR_SAMP
VARCHAR
VARYING
WHEN
WHENEVER
WHERE
WIDTH_BUCKET
WINDOW
WITH
WITHING
WITHOUT
YEAR

Предметный указатель

A

ABS 128
ABSOLUTE 240
ADD COLUMN 204
ALL 111, 142
ALL PRIVILEGES 283
ALTER COLUMN 203
ALTER TABLE 203
AND 118
ANY 111
ASC 117
ASENSITIVE 235
ATOMIC 260
AVG 122

B

BEGIN 261
BEGIN ... END 258
BEGIN ATOMIC 264
BETWEEN 108
BIGINT 80
BOOLEAN 83

C

CALL 275
CARDINALITY 128
CASE 134, 135
CASE ... END CASE 266
CEIL 129
CHARACTER 78
CHARACTER_LENGTH 128
CHECK 194
CLOB 78
CLOSE 239, 245
COALESCE 137
COMMIT 219
CONDITION 263
CONSTRAINT 192
CONTINUE 263
COUNT 121
CREATE DISTINCT TYPE 90
CREATE DOMAIN 196, 288
CREATE FUNCTION 275
CREATE GROUP 285
CREATE MODULE 276
CREATE PROCEDURE 273
CREATE ROLE 284
CREATE TABLE 191
CREATE TYPE 91
CREATE USER 281
CREATE VIEW 210
CURRENT_DATE 130
CURRENT_TIME 130
CURRENT_TIMESTAMP 130
CURSOR 271
CURSOR FOR 244

D

DATA 83
DECIMAL 81
DECLARE 260
DECLARE SECTION 244
DEFAULT 194
DELETE 192
DESC 117
DISTINCT 113
DOUBLE PRECISION 81
DROP COLUMN 203
DROP TABLE 202
DROP USER 282

E

EXCEPT 158
EXEC SQL 244, 259
EXISTS 112
EXIT 263
EXP 129
EXTRACT 128

F

FETCH 244
FETCH INTO 239
FINAL 92
FIRST 240
FLOAT 81
FLOOR 129
FOR ... DO ... END FOR 270
FOR UPDATE 238
FOREIGN KEY 199
FOREIGN KEY ...
REFERENCES 198
FROM 103
FULL JOIN 166

G

GRANT 283
GRANT USAGE 288
GROUP BY 104, 114

H

HANDLER 263
HAVING 104, 116

I

IF 265
IN и NOT IN 109
INSENSITIVE 235
INSERT 180
INTEGER 80
INTERSECT 157
INTERVAL 86
IS NULL 111
ITERATE 272

J

JOIN ... ON 161
JOIN ... USING 162

L

LAST 240
LEFT OUTER JOIN 164
LIKE и NOT LIKE 109
LN 129
LOOP ... END LOOP 268
LOWER 126

M

MATCH 113, 199
MAX 123
MIN 123
MOD 129

N

NATURAL JOIN 159
NEXT 240
NO SCROLL 235
NOT 118
NOT FINAL 92
NOT FOUND 263
NOT NULL 193
NULL 93
NULLIF 136
NUMERIC 81

O

ON DELETE 200
ON UPDATE 200
OPEN 238, 244
OR 118
ORDER BY 104, 117
OVERLAPS 113

P

POSITION 127
POWER 129
PRIMARY KEY 61, 194
PRIOR 240
PUBLIC 281

R

REAL 81
RELATIVE 240
RENAME COLUMN 203
RENAME TO 203
REPEAT ... UNTIL ... END REPEATE 270
RESIGNAL 264
RETURN 275
RETURNS 275
REVOKE 290
RIGHT OUTER JOIN 166
ROLLBACK 219

S

SCROLL 235
SELECT ... FROM 99
SENSITIVE 235
SET TRANSACTION 220
SMALLINT 80
SOME 111, 142
SQLEXCEPTION 263
SQLSTATE 261
SQLWARNING 263
SQRT 129
SUBSTRING 126
SUM 122

T

TIME WITH TIME ZONE 84
TIME WITHOUT TIME ZONE 83
TIMESTAMP WITH TIME ZONE 85
TIMESTAMP WITHOUT TIME ZONE 84
TOP 101
TRIM 126

U

UNDO 263
UNION 154
UNION JOIN 167
UNIQUE 112, 193
UPDATE 195
UPPER 126

V

VARCHAR 78

W

WHERE 100, 102, 106
WHILE ... DO ... END
  WHILE 269
WIDTH RECURSIVE 170
WIDTH_BUCKET 129
WITH GRANT
  OPTION 283, 289

А

Администратор базы
  данных 279
Аномалия модификации 62
Арность отношения 33
Атомарность 259
Атрибут 34

В

Владелец объектов 279
Выражения:
  вычисляемые 130
  условные 133

Г

Группа
  пользователей 284
Группировка 114, 116

Д

Декартово произведение
  таблиц 151
Декартово произведение
  множеств 30
Декомпозиция:
  корректная 44
  некорректная 44
  отношений 43
Добавление записей 190
Домен 34, 195, 288
Дополнение
  множества 24

З

Зависимость:
  многозначная 49
  правила вывода 55
  транзитивная 67
  тривиальная 48, 58
  функциональная 46
Законы Моргана 24

И

Изменение данных 195

К

Кванторы 27, 142
Ключ 58
  внешний 199
  первичный 194
Курсор 232, 270
  закрытие 239
  область действия 236
  объявление 234
  ориентация 240
  открытие 238
  перемещаемость 235
  разрешение
    обновления 238
  чувствительность 235

М

Множество 17
Модификация таблиц 202
Модули:
  хранимые 276

О

Объединение множеств 23
Ограничения:
  столбцов 193
  таблиц 197
  целостности 30
  целостности отношений 60
Операторы:
  арифметические 131
  для уточнения запроса 103
  логические 118
  условного перехода 264
  цикла 268
Операция:
  внешнее соединение 164
  вычитание 158
  декартово произведение 151
  естественное соединение 159
  объединение 154
  пересечение 157
  проекции 38
  селекции 35
  соединение по именам столбцов 162
  сужения 35
  условное соединение 161
Отношение 29
  включения 22
  принадлежности 23
  структура 34
  унарное, бинарное, тернарное 31

П

Переменные 260
Пересечение множеств 23
Подзапросы 140
Права доступа 279
Предикат 21
Предикаты:
  для вложенных запросов 111
  сравнения 108
Представление 207
Преобразование типов 95
  функция CAST() 95
Привилегированные пользователи 279
Процедуры хранимые 273
Псевдоним 124, 146, 150

Р

Работа с отдельными записями 239
Разность множеств 23
Рекурсивные запросы 169
Роль 284
Рядовые пользователи 279

С

Сегмент объявлений 244
Сложные запросы 139
Создание:
  базы данных:
    в Enterprise Manager 9
    в Microsoft Access 2003 12
  группы 285
  домена 196
  отдельного типа данных 90
  пользователя 281
  представления 210
  роли 284
  структурированного типа данных 91
  таблицы 191
Сортировка 117, 236
Составные команды 258
Статус SQL 261

Т

Типы данных 76
  дата-время 83
  запись 87
  интервал 86
  логический 83
  массив 88
  отдельные 90
  пользовательские 89
  строковые (символьные) 78
  структурированные 91
  числовые 80
Транзакция 217

У

Удаление записей 192
Удаление таблиц 202
Универсум 24
Управление правами доступа 279

Ф

Функции:
  даты-времени 130
  итоговые 121
  обработки значений 125
  строковые 126
  хранимые 273
  числовые 127
Функция CAST() 95
Функция PHP:
  sqlite_close() 253
  sqlite_current() 256
  sqlite_exec() 253
  sqlite_fetch_array() 254
  sqlite_next() 255
  sqlite_open() 253
  sqlite_prev() 255
  sqlite_query() 253
  sqlite_rewind() 255
  sqlite_seek() 256

Ц

Целостность:
  доменная 61
  семантическая 61
  ссылочная 62