- **Введение**
  Добро пожаловать в область разработки баз данных, выполняемой с помощью стандартного языка запросов SQL. В системах управления базами данных (СУБД) имеется много инструментов, работающих на самых разных аппаратных платформах.
- **Основы реляционных баз данных**
  В этой главе… | Организация информации | Что такое база данных | Что такое СУБД | Сравнение моделей баз данных | Что такое реляционная база данных
- **Основы SQL**
  В этой главе… | Что такое SQL | Заблуждения, связанные с SQL | Взгляд на разные стандарты SQL | Знакомство со стандартными командами и зарезервированными словами SQL | Представление чисел, символов, дат, времени и других типов данных | Неопределенные значения и ограничения
- **Компоненты SQL**
  В этой главе… | Создание баз данных | Обработка данных | Защита баз данных | SQL – это язык, специально разработанный, чтобы создавать и поддерживать данные в реляционных базах. И хотя компании, поставляющие системы для управления такими базами, предлагают свои реализации SQL, развитие самого языка определяется и контролируется стандартом ISO/ANSI.
- **Создание и поддержка простой базы данных**
  В этой главе… | Создание, изменение и удаление таблицы из базы данных с помощью инструмента RAD. | Создание, изменение и удаление таблицы из базы данных с помощью SQL. | Перенос базы данных в другую СУБД.
- **Создание многотабличной реляционной базы данных**
  В этой главе… | Что должно быть в базе данных | Определение отношений между элементами базы данных | Связывание таблиц с помощью ключей | Проектирование целостности данных | Нормализация базы данных | В этой главе будет представлен пример создания многотабличной базы данных.
- **Манипуляции данными из базы**
  В этой главе… | Работа с данными | Получение из таблицы нужных данных | Вывод информации, выбранной из одной или множества таблиц | Обновление информации, находящейся в таблицах и представлениях | Добавление новой строки в таблицу
- **Определение значений**
  В этой главе… | Использование переменных для уменьшения избыточного кодирования | Получение часто запрашиваемой информации, находящейся в поле таблицы базы данных | Комбинирование простых значений для создания составных выражений | В этой книге постоянно подчеркивается, насколько важной для поддержания целостности базы данных является структура этой базы.
- **Сложные выражения со значением**
  В этой главе… | Использование условных выражений case | Преобразование элемента данных из одного типа данных в другой | Экономия времени ввода данных с помощью выражений со значением типа запись | В главе 2 SQL был назван подъязыком данных.
- **"Пристрелка" к нужным данным**
  В этой главе… | Указание требуемых таблиц | Отделение нужных строк от всех остальных | Создание эффективных предложений where | Как работать со значениями null | Создание составных выражений с логическими связками | Группирование вывода результата запроса по столбцу
- **Реляционные операторы**
  В этой главе… | Объединение таблиц, имеющих похожую структуру | Объединение таблиц, имеющих разную структуру | Получение нужных данных из множества таблиц | SQL – это язык запросов, используемый в реляционных базах данных.
- **Использование вложенных запросов**
  В этой главе… | Извлечение данных из множества таблиц с помощью одного оператора SQL | Поиск элементов данных путем сравнения значения из одной таблицы с набором значений из другой | Поиск элементов данных путем сравнения значения из одной таблицы с выбранным с помощью оператора select единственным значением из другой
- **Рекурсивные запросы**
  В этой главе… | Управление рекурсией | Как определять рекурсивные запросы | Способы применения рекурсивных запросов | SQL-92 и более ранние версии часто критиковали за отсутствие реализации рекурсивной обработки.
- **Обеспечение безопасности базы данных**
  В этой главе… | Управление доступом к таблицам базы данных | Принятие решения о предоставлении доступа | Предоставление полномочий доступа | Аннулирование полномочий доступа | Предотвращение попыток несанкционированного доступа
- **Защита данных**
  В этой главе… | Как избежать повреждения базы данных | Проблемы, вызванные одновременными операциями | Решение этих проблем с помощью механизмов SQL | Задание требуемого уровня защиты с помощью команды set transaction
- **Использование SQL в приложениях**
  В этой главе… | SQL в приложении | Совместное использование SQL с процедурными языками | Как избежать несовместимости | Код SQL, встроенный в процедурный код | Вызов модулей SQL из процедурного кода | Вызов SQL из RAD-инструмента | В предыдущих главах мы в основном рассматривали SQL-команды в отдельности, т.е. формулировалась задача обработки данных, и под нее создавался SQL-запрос.
- **ODBC и JDBC**
  В этой главе… | Определение ODBC | Описание частей ODBC | Использование ODBC в среде клиент/сервер | Использование ODBC в Internet | Использование ODBC в локальных сетях | Использование JDBC | С каждым годом компьютеры одной организации или нескольких различных организаций все чаще соединяются друг с другом. Поэтому возникает необходимость в налаживании совместного доступа к базам данных по сети.
-

# Введение

Добро пожаловать в область разработки баз данных, выполняемой с помощью стандартного **языка запросов** SQL. В системах управления базами данных (СУБД) имеется много инструментов, работающих на самых разных аппаратных платформах. Различия между этими инструментами могут быть очень существенными, но все они имеют одну общую черту – доступ к данным и их обработка выполняются с помощью SQL. Зная SQL, вы можете создавать реляционные базы данных и получать из них полезную информацию.

## Об этой книге

Системы управления базами данных играют жизненно важную роль во многих организациях. Люди часто думают, что создание и поддержание этих систем – очень сложное занятие, удел "жрецов" баз данных, которым открыта истина, недоступная простым смертным. Эта книга напрочь отметает мистику, связанную с базами данных.Прочитав данную книгу, вы:

- доберетесь до истоков баз данных;
- узнаете, что собой представляет структура СУБД;
- откроете для себя основные функциональные компоненты SQL;
- создадите базу данных;
- защитите базу данных;
- научитесь работать с ее данными;
- узнаете, каким образом получать из базы данных нужную информацию.

Цель книги состоит в том, чтобы научить вас создавать реляционные базы данных и извлекать из них ценную информацию с помощью **SQL** – международного стандартного языка, используемого во всем мире для создания и поддержки реляционных баз данных. В пятом издании книги рассказывается о последней версии этого языка, SQL:2003.

В книге речь идет не о том, как проектировать базу данных, поскольку на эту тему написано много хороших книг. Предполагается, что уже существует подходящий проект базы. Здесь будет показано, как реализовать этот проект с помощью SQL. Если же у вас есть подозрение, что имеющийся проект не очень хорош, то исправьте его во что бы то ни стало, пока еще не начали создавать саму базу данных. Чем раньше вы обнаружите и исправите недостатки разработанного проекта, тем дешевле вам обойдутся те исправления, которые все-таки придется делать.

## Для кого предназначена эта книга

Если вам надо хранить данные в СУБД или получать их оттуда, то практическое знание SQL поможет вам делать свою работу намного лучше. Чтобы использовать SQL, не нужно быть программистом и знать программные языки, такие как COBOL, C или Basic. Синтаксис языка SQL основан на упрощенном синтаксисе английского.

А если вы все-таки программист, то сможете использовать SQL в своих программах. С его помощью у обычных языков программирования появляется мощный аппарат обработки данных. В настоящей книге рассказывается, что именно нужно знать, чтобы реализовать в своих программах богатый набор возможностей, предоставляемых SQL.

# Основы реляционных баз данных

В этой главе…

- Организация информации
- Что такое база данных
- Что такое СУБД
- Сравнение моделей баз данных
- Что такое реляционная база данных
- С какими трудностями можно столкнуться при проектировании баз данных

**SQL** (**Structured Query Language** – язык структурированных запросов) – это стандартный язык, предназначенный для создания баз данных, добавления новых и поддержки имеющихся данных, а также извлечения требуемой информации. В зависимости от используемой теоретической модели, базу данных относят к одному из нескольких типов. Язык SQL был создан для работы с данными из тех баз, которые следуют **реляционной модели**. Недавно в международный стандартный язык SQL были включены элементы**объектной модели**, в результате чего появились гибридные структуры, называемые**объектно-реляционными базами данных**. В этой главе рассказывается о хранении данных. Один из ее разделов посвящен сравнению реляционной и других основных моделей. Кроме того, в ней представлен обзор главных особенностей реляционных баз данных.

Впрочем, перед тем как рассказывать о SQL, нужно дать определение понятию **базы данных**. Развитие компьютеров изменило способы хранения и обработки информации, а также значение этого термина.

## Работа с данными

С помощью компьютеров люди выполняют множество таких задач, которые раньше выполнялись с помощью других инструментов. Документы теперь создают и исправляют не на пишущих машинках, а на компьютерах. Электромеханические калькуляторы также заменены компьютерами – лучшим средством для выполнения математических вычислений. Компьютеры пришли на смену миллионам листов бумаги, миллионам папок и стеллажей для документов и являются главными хранилищами важной информации.

Конечно, в сравнении со старыми инструментами компьютеры выполняют намного больше работы, намного быстрее и, главное, – с большей точностью. Однако за все приходится платить. Пользователи компьютеров больше не имеют прямого физического доступа к своим данным.

Как только компьютеры внезапно перестают работать, у сотрудников учреждений тут же закрадываются сомнения– а действительно ли компьютеризация благо? Раньше папкам с документами угрожало лишь падение на пол, и достаточно было просто нагнуться, собрать выпавшие листы бумаги и снова положить их в папку, чтобы все встало на свои места. Если не считать землетрясений и других катаклизмов, то стеллажи с папками никогда не "удаляются" и никогда не отправляют сообщения об ошибке. А вот авария с жестким диском – это совсем другое дело: потерянные биты и байты "подобрать" нельзя. Отказы оборудования, вызванные механическими, электрическими и человеческими воздействиями, могут безвозвратно отправить ваши данные на тот свет.

## Малое – это прекрасно

В области хранения данных компьютеры действительно предстали во всей своей красе. Это произошло потому, что они могут хранить в двоичном виде любую информацию: текст, числа, звуки, графические изображения, телевизионные программы или анимацию. Причем компьютер может хранить данные с очень высокой плотностью, позволяя поместить большое количество информации. По мере совершенствования технологий все больше и больше данных занимают все меньше и меньше места. Где только ни используются сейчас компьютеры: и в газовых насосах, и в новых автомобилях, и в ошеломляющем количестве игрушек. Осталось недолго ждать появления компьютеризованных туфель, меняющих упругость своих подошв в зависимости от того, идете ли вы, бежите или прыгаете. А звезды баскетбола, может быть, вскоре смогут использовать обувь с возможностью хранения небольшой базы данных с игровой статистикой…

Большая скорость и точность компьютеров только тогда пойдут на пользу, если будут приняты необходимые меры по защите от случайной потери данных. При хранении важных данных возникают четыре главные задачи.

- Соответствующие операции должны выполняться быстро и легко, так как заниматься ими вам, скорее всего, придется часто.
- Носитель, предназначенный для хранения данных, должен быть надежным. Вы не захотите впоследствии обнаружить, что большая часть ваших данных пропала.
- Получение данных должно быть быстрым и легким, независимо от количества сохраняемых элементов этих данных.
- Необходим способ легкого поиска нужной информации среди "тонн" ненужных данных.

Этим четырем критериям и соответствуют базы данных. Если приходится хранить более десятка элементов данных, то, вероятнее всего, появляется желание хранить эти элементы именно в базе данных.

## Что такое база данных

Понятие база данных вошло в широкий обиход довольно-таки поздно и при этом потеряло многие из своих первоначальных значений. Для некоторых база данных является собранием элементов данных (телефонной книгой, перечнем прачечных, пергаментным свитком… да всем, чем угодно). Другие же определяют понятие более строго.

В этой книге **база данных** определяется как самоописательное собрание интегрированных записей. При этом она является компьютерной технологией, укомплектованной языками типа SQL.

***Помни:***

*Запись является представлением некоего физического или умозрительного объекта. Скажем, вы, например, собираетесь сохранять данные о своих клиентах. Каждый из них имеет свою запись. А в каждой записи имеется набор атрибутов, таких как имя, адрес и номер телефона. Имена, адреса и другие значения, соответствующие этим атрибутам, и представляют собой данные*.

База данных состоит как из данных, так и из **метаданных**. Метаданные – это данные, которые являются описанием структуры данных, находящихся внутри базы. Зная, как расположены данные, можно их получить. Так как описание структуры базы данных находится в самой базе, то она является **самоописательной**. База данных является**интегрированной**, ибо содержит в себе не только элементы данных, но и существующие между ними взаимосвязи.

В базе данных метаданные хранятся в области, которая называется **словарь данных**. Он описывает таблицы, столбцы, индексы, ограничения и другие компоненты, из которых состоит база данных.

Так как в системе плоских файлов (описанной далее в этой главе) метаданных нет, то в приложениях, работающих с этими файлами, роль метаданных должна взять на себя часть приложения.

### Размер и сложность базы данных

Базы данных бывают любых размеров, начиная от простого набора из нескольких записей до огромных систем с миллионами записей.

***Помни:***

*Персональная база данных предназначена для использования одним человеком на одном компьютере. У такой базы обычно достаточно простая структура и относительно небольшой размер. База данных отдела или рабочей группы используется сотрудниками одного отдела или членами одной рабочей группы в пределах одной организации. Такая база обычно больше персональной и, конечно же, более сложная. С ней должны иметь возможность работать несколько пользователей, которым одновременно нужен доступ к одним и тем же данным. База данных организации бывает громадных размеров. Она может полностью моделировать информационный обмен в крупной организации*.

### Что такое система управления базами данных

**Система управления базами данных** (СУБД) – это набор программ, используемых для определения, администрирования и обработки баз данных и связанных с ними приложений. База данных, управляемая такой системой, является, в сущности, структурой, которую создают, чтобы хранить в ней нужные данные. А СУБД – это инструмент, используемый для создания этой структуры и работы с данными, которые в ней хранятся.

Сегодня на рынке имеется много программ СУБД. Некоторые из них работают только на мэйнфреймах, другие – только на мини-компьютерах, а есть такие, которые работают только на персональных компьютерах. Однако наблюдается тенденция к переносу СУБД на множество платформ с возможностью работы в сетях со всеми тремя классами машин.

Система СУБД, работающая на платформах нескольких классов, больших и малых, называется **масштабируемой**.

Каким бы ни был класс компьютера с базой данных – независимо от того, соединена ли машина с сетью или нет, – поток информации между базой данных и пользователем в принципе один и тот же. На рис. 1.1 показано, что пользователь соединяется с базой данных с помощью СУБД. Та скрывает физические детали хранения базы данных, так что приложению приходится иметь дело только с логическими характеристиками данных, а не с тем, каким образом эти данные хранятся.

Много лет назад один умник подсчитал, что если разложить любое человеческое существо на компоненты, такие как атомы углерода, водорода, кислорода и азота (плюс незначительное количество других), то их стоимость будет только 97 центов. Впрочем, это совершеннейшая глупость. Люди не состоят из простых изолированных наборов атомов. Наши атомы комбинируются в энзимы, протеины, гормоны и другие вещества, стоимость которых на фармацевтическом рынке обычно составляет миллионы долларов за унцию.

Точная структура таких комбинаций атомов – вот что составляет их ценность. И, аналогично, структура баз данных позволяет интерпретировать данные, кажущиеся бессмысленными. Закономерности и тенденции, имеющиеся в данных, становятся известными благодаря структуре этих данных. Неструктурированные данные, как и неупорядоченные атомы, имеют малую ценность или совсем ее не имеют.

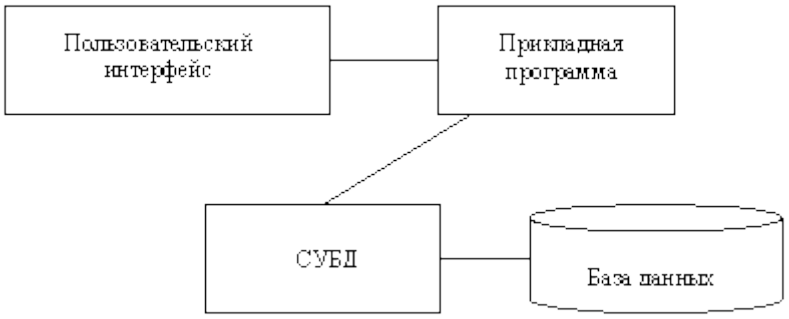


**Рис. 1.1**. Схема информационной системы, работающей на основе СУБД

## Плоские файлы

**Плоские файлы** – самая простая разновидность структурированных данных. Нет, плоский файл – это не папка, придавленная стопкой книг. **Плоские файлы** называются так потому, что имеют минимальную структуру. Если бы они были зданиями, то их стены поднимались бы не от фундамента, а прямо от земли. Плоский файл – это собрание записей данных, записываемых в определенном формате одна за другой, – данные, одни только данные и ничего, кроме данных, т.е. список. На компьютерном языке плоский файл называется простым. В таком файле нет метаданных со структурной информацией, а есть лишь одни данные.

Скажем, вам нужно сохранить в системе плоских файлов имена и адреса клиентов вашей компании. У этой системы может быть примерно такая структура.

| | | | | |
|---|---|---|---|---|
| Harold Percival | 26262 | S. Howards Mill Rd | Westminster | CA92683 |
| Jerry Appel | 32323 | S. River Lane Rd | Santa Ana | CA92705 |
| Adrian Hansen | 232 | Glenwood Court | Anaheim | CA92640 |
| John Baker | 2222 | Lafayette St | Garden Grove | CA92643 |
| Michael Pens | 77730 | S. New Era Rd | Irvine | CA92715 |
| Bob Michimoto | 25252 | S. Kelmstey Dr | Stanton | CA92610 |
| Linda Smith | 444 | S.E. Seventh St | Costa Mesa | CA92635 |
| Robert Funnell | 2424 | Shen Court | Anaheim | CA92640 |
| Bill Checkal | 9595 | Curry Dr | Stanton | CA92610 |
| Jed Style | 3535 | Randall St | Santa Ana | CA92705 |

Как видите, в файле нет ничего, кроме данных. Каждое поле имеет фиксированную длину (например, длина поля имени всегда равна 15 символам), и в этой структуре поля не отделены друг от друга. Тот, кто создал базу данных, для каждого из полей назначил позицию и длину. Любая программа, которая использует этот файл, должна "знать", какие характеристики назначены каждому полю, потому что этой информации в самой базе данных нет.

Такая структура плоских файлов позволяет работать с ними очень быстро. Однако недостатком является то, что программная логика, которая предназначена для манипуляции данными из файлов, должна быть очень подробной. Приложение должно точно "знать", где и как в файле хранятся данные. Итак, что касается малых систем, то в них плоские файлы работают прекрасно. Но чем больше система плоских файлов, тем труднее

с ней работать. Использование базы данных вместо системы плоских файлов позволяет этого избежать. Хотя файлы базы данных имеют больший "фундамент", приложения могут работать на большем количестве аппаратных платформ и операционных систем. Кроме того, базы данных позволяют легче писать прикладные программы, потому что программисту не нужно вникать в детали того, как в файлах физически расположены данные.

Базы данных облегчают работу программистов, потому что при работе с данными в детали "вникает" СУБД. А приложениям, написанным для работы с плоскими файлами, необходимо держать эти детали при себе, т.е. в собственном коде. Если нескольким приложениям приходится одновременно получать доступ к одним и тем же данным из плоских файлов, то в каждом из приложений обязательно должен быть код, предназначенный для работы с этими данными. Но когда используется СУБД, то такой код в приложениях вообще не нужен.

Кроме того, если в приложении имеется код для работы с данными из плоских файлов, причем работает он только на определенной аппаратной платформе, то перенос приложения на новую платформу – это довольно сложное дело. Ведь придется изменить весь код, связанный с аппаратным обеспечением. А вот перенос на другую платформу аналогичного СУБД-приложения проходит намного проще – с меньшим количеством проблем и выпитого аспирина.

## Модели баз данных

Независимо от размеров баз данных все они относятся к одной из трех нижеприведенных моделей.

- **Реляционная**. Если сейчас где-нибудь развертывают новую систему, предназначенную для управления базами данных, то почти всегда такая система является реляционной. Конечно, в организациях, где уже вложено много ресурсов в иерархическую и сетевую технологии, могут наращивать и имеющуюся модель. Однако в группах, где нет необходимости поддерживать совместимость с унаследованными от прошлого системами, для своих баз данных почти всегда выбирают реляционную модель.
- **Иерархическая**. Иерархические базы данных получили такое название потому, что имеют простую иерархическую структуру, позволяющую иметь быстрый доступ к данным. Их недостатками являются избыточность данных, т.е. их дублирование, и негибкость структуры, что усложняет модификацию таких баз данных.
- **Сетевая**. В сетевых базах данных дублирование минимально, но за это приходится платить сложностью структуры.

Первыми базами данных, получившими широкое распространение, были большие базы данных организаций, созданные в соответствии с иерархической или сетевой моделью. Через несколько лет появились системы, созданные в соответствии с реляционной моделью. Язык SQL является по-настоящему современным; он применяется только к реляционной модели и ее производной – объектно-реляционной модели. Так что в этом месте книги остается сказать иерархической и сетевой моделям: "Приятно было познакомиться, а теперь – до свидания".

Новые системы управления базами данных, которые не являются реляционными, соответствуют, скорее всего, более новой, чем реляционная, объектной модели или гибридной объектно-реляционной модели.

## Реляционная модель

Впервые реляционную модель баз данных сформулировал в 1970 году работавший в компании IBM доктор И.Ф. Кодд (**E. F. Codd**), а примерно десятилетие спустя эта модель

начала появляться в готовых продуктах. По иронии судьбы первую реляционную СУБД разработала не IBM. Такая честь выпала на долю маленькой компании-новичка, назвавшей свой продукт Oracle.

Базы данных, созданные на основе предыдущих моделей, были заменены реляционными, потому что не имели тех ценных свойств, которые и отличают реляционные базы от баз других типов. Вероятно, самым важным из этих свойств является то, что в реляционной базе данных можно менять структуру, не внося изменений в приложения. Такого не скажешь о приложениях, созданных на основе старых структур. Предположим, например, что в таблицу базы данных вы добавили один или несколько новых столбцов. В этом случае нет необходимости менять никакое из уже написанных приложений, которые будут продолжать обрабатывать эту таблицу, – только если вы не изменили столбцы, с которыми работают эти приложения.

***Внимание:***

*Конечно, если вы удалили столбец, к которому обращается имеющееся приложение, то какая бы модель базы данных ни применялась, вы столкнетесь с трудностями. Один из лучших способов устроить аварийное завершение работы приложения базы данных – запросить у него такую информацию, которой нет в вашей базе данных*.

### Почему реляционная модель лучше

В приложениях, работающих с СУБД, которые следуют иерархической или сетевой модели, структура базы данных "зашита" в само приложение. Это означает, что приложение зависит от определенной физической реализации базы данных. При добавлении в базу данных нового атрибута вам, чтобы привести свое приложение в соответствие с изменением базы, придется это приложение изменить, причем неважно, будет ли оно использовать новый атрибут.

У реляционных баз данных более гибкая структура. Приложения для таких баз поддерживать легче, чем те, что написаны для иерархических или сетевых баз данных. Эта гибкость структуры дает возможность получать такие комбинации данных, которые, возможно, еще не были нужны при проектировании базы данных.

### Компоненты реляционной базы данных

Гибкость реляционных баз данных объясняется тем, что их данные находятся в таблицах, которые в значительной степени независимы друг от друга. В таблицу данные можно добавлять, удалять их из нее, вносить в них изменения и при этом не затрагивать данные из других таблиц – если только таблица не является **родительской** по отношению к этим другим таблицам. (Об отношениях родительских и дочерних таблиц рассказывается в главе 5, но там речь пойдет не о конфликте поколений.) В этом разделе будет показано из чего состоят таблицы и как они связаны с другими частями реляционной базы данных.

### Родственники и таблицы – что общего?

В праздничные дни многие родственники приходят ко мне в гости и сидят за моим столом. В базах данных также имеются отношения (но не родственные), и каждое из них имеет свою собственную таблицу (английское слово " table " может иметь два значения: "стол" или "таблица", а " relation " – "родство" или "отношение"). Реляционная база данных состоит из одного или нескольких отношений.

***Помни:***

*Отношение – это двумерный массив строк и столбцов, причем у каждого из этих элементов может быть только одно значение, а строки друг друга не дублируют. Каждая ячейка в массиве может содержать только одно значение, и никакие из двух строк не могут быть одинаковы*.

Большинство людей знакомы с двумерными массивами строк и столбцов. Это электронные таблицы, с которыми можно работать в таких приложениях, как Microsoft Excel. Другой пример двумерного массива – это статистика на обратной стороне бейсбольной карточки игрока высшей лиги. В такой карточке имеются столбцы, указывающие год, команду, количество игр, ударов битой, попаданий, выигранных очков и т.п. Строки соответствуют годам, на протяжении которых игрок играл в высшей лиге. Эти данные также можно хранить в отношении (таблице), которое имеет ту же простую структуру. На рис. 1.2 показана таблица реляционной базы данных, содержащая статистику по одному из игроков высшей лиги. В действительности такая таблица может хранить статистику для всей команды или даже целой лиги.

### Исторические перспективы

Персональные базы данных впервые появились на персональных компьютерах в начале 1980-х годов. Самые первые продукты работали с системами плоских файлов, но чуть позже появились продукты, из которых некоторые пытались следовать реляционной модели. По мере развития некоторые популярные системы СУБД для персональных компьютеров достаточно близко подошли к тому, чтобы стать по-настоящему реляционными, т.е. удовлетворяющими определению доктора Кодца. Начиная с конца 1980-х годов в организациях все больше и больше ПК объединяются в рабочие группы или в сети отделов. Чтобы заполнить эту новую рыночную нишу, те СУБД, которые первоначально работали только на больших ЭВМ (мэйнфреймах), "спустились" на персональные компьютеры, а реляционные СУБД для ПК, наоборот, "поднялись" на большие ЭВМ.

| Player | Year | Team | Game | At Bat | Hits | Runs | RBI | 2B | 3B | HR | Walk | Steals | Bat. Avg. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Roberts | 1988 | Pedres | 5 | 9 | 3 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | .333 |
| Roberts | 1989 | Pedres | 117 | 329 | 99 | 81 | 25 | 15 | 8 | 3 | 49 | 21 | .301 |
| Roberts | 1990 | Pedres | 149 | 556 | 172 | 104 | 44 | 36 | 3 | 9 | 55 | 46 | .309 |

**Рис. 1.2**. Таблица со статистикой игрока

Столбцы в массиве являются **постоянными**, т.е. означают в каждой строке одно и то же. Если в одной строке столбец содержит фамилию игрока, то в остальных строках в том же столбце должны быть фамилии. Порядок расположения строк и столбцов в массиве не имеет значения. Что касается СУБД, то для нее не имеет значения, какой столбец будет первым, какой – следующим и какой – последним. Каким бы ни был порядок столбцов, СУБД будет обрабатывать таблицу одинаковым способом. То же верно и для строк. Для СУБД порядок расположения строк не имеет значения.

Каждый столбец в таблице из базы данных представляет один из атрибутов этой таблицы. Это означает, что столбец содержит однородные данные в каждой строке таблицы. Например, в таблице могут находиться имена, адреса и номера телефонов всех клиентов организации. Каждая табличная строка (также называемая **записью**, или **кортежем**) содержит данные по одному клиенту. А каждый столбец содержит один атрибут. Это, например, может быть один из следующих атрибутов: номер клиента, его имя, улица, на которой он живет, его город, штат, почтовый код или номер телефона. Столбцы и строки такой таблицы показаны на рис. 1.3.

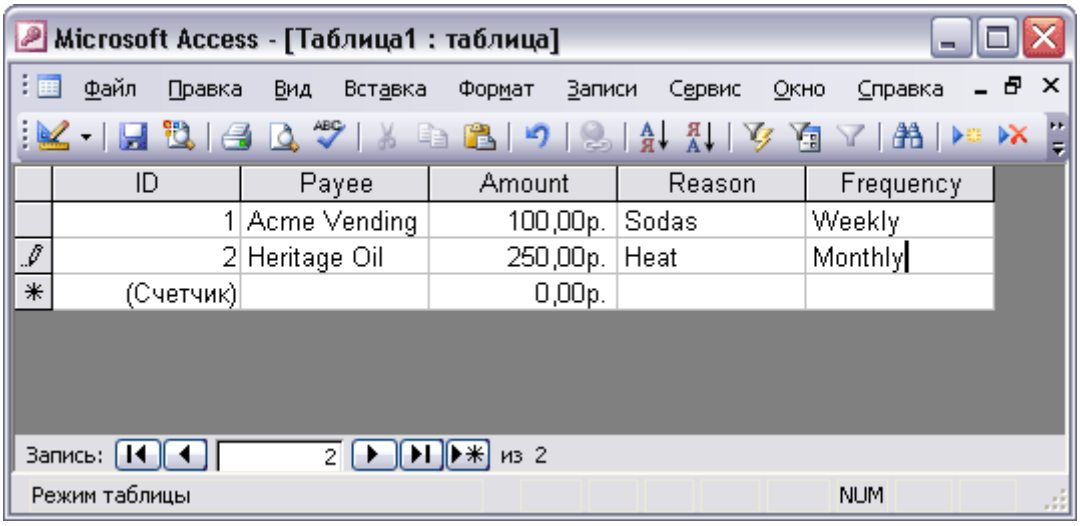


**Рис. 1.3**. Каждая строка базы данных содержит запись; каждый столбец этой базы содержит один атрибут

***Помни:***

*Понятие отношения в модели базы данных – это то же самое, что таблица в базе данных, основанной на определенной модели. Попробуйте повторить это определение быстро десять раз.*

## Оцените представление

В моем представлении часто возникает один из моих любимых пейзажей: вид на Йосемитскую долину весенним вечером на выезде из туннеля Уовона. Золотой свет заливает отвесную поверхность скалы Эль-Капитан, вдали сияет пик Хаф-Доум, а водопад "Фата невесты" (Бридалвейл) создает серебряный каскад сверкающей воды, в то время как стайка легких облаков слегка закрывает небо. У баз данных также имеются виды, пусть даже не такие живописные. Называются они представлениями. Их красота заключена в реальной пользе, которую они приносят при работе с данными.

В таблицах может находиться достаточно много строк и столбцов. Иногда все эти данные могут вас интересовать, а иногда – нет. Вас, возможно, интересуют только некоторые из столбцов, имеющихся в таблице, или только строки, которые удовлетворяют определенному условию. Или могут интересовать некоторые столбцы из одной таблицы и некоторые другие – из другой, связанной с ней таблицы. Чтобы исключить данные, которые вам в данный момент не нужны, можно создать представление. **Представление** (view) – это подмножество базы данных, которое обрабатывается приложением. В представлении могут находиться части одной или множества таблиц.

***Помни:***

*Представления иногда называют виртуальными таблицами. Для приложения или пользователя они ведут себя точно так же, как и таблицы. Однако представления сами по себе не существуют. Они дают возможность просматривать данные, но сами частью данных не являются.*

Скажем, вы работаете, например, с базой данных, в которой имеются таблицы**CUSTOMER** (клиент) и **INVOICE** (счет-фактура). В первой таблице имеются столбцы**CustomerID** (идентификатор клиента), **FirstName** (имя), **LastName** (фамилия), **Street**(улица), **City** (город), **State** (штат), **Zipcode** (почтовый код) и **Phone** (телефон). А столбцами второй таблицы являются **InvoiceNumber** (номер счета-фактуры), **CustomerID**(идентификатор клиента), **Date** (дата), **TotalSale** (всего продано), **TotalRemitted** (всего переведено денег) и **FormOfPayment** (форма платежа).

Главный менеджер по продажам хочет видеть на экране только такие реквизиты клиентов, как имя, фамилия и номер телефона. Если из таблицы CUSTOMER создать представление, в котором содержатся лишь три столбца с названной информацией, то менеджер будет просматривать только нужную ему информацию. А данные из тех столбцов, которые его не интересуют, он видеть не будет. На рис. 1.4 показано получение представления для главного менеджера по продажам.

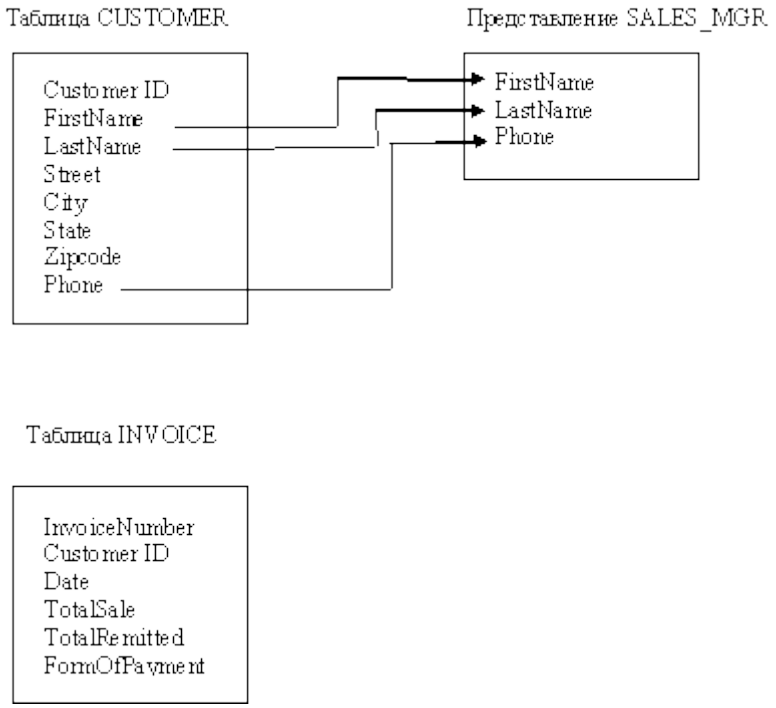


**Рис. 1.4**. Представление для главного менеджера по продажам, получаемое из таблицы CUSTOMER

Региональный менеджер по продажам, возможно, хочет видеть имена, фамилии и номера телефонов всех клиентов, с почтовым кодом в диапазоне между 90000 и 93999 (Южная и Центральная Калифорния). Эту работу выполняет представление, которое накладывает ограничение на получаемые с его помощью строки, а также на выводимые с его помощью столбцы. На рис. 1.5 показано получение представления для регионального менеджера по продажам.

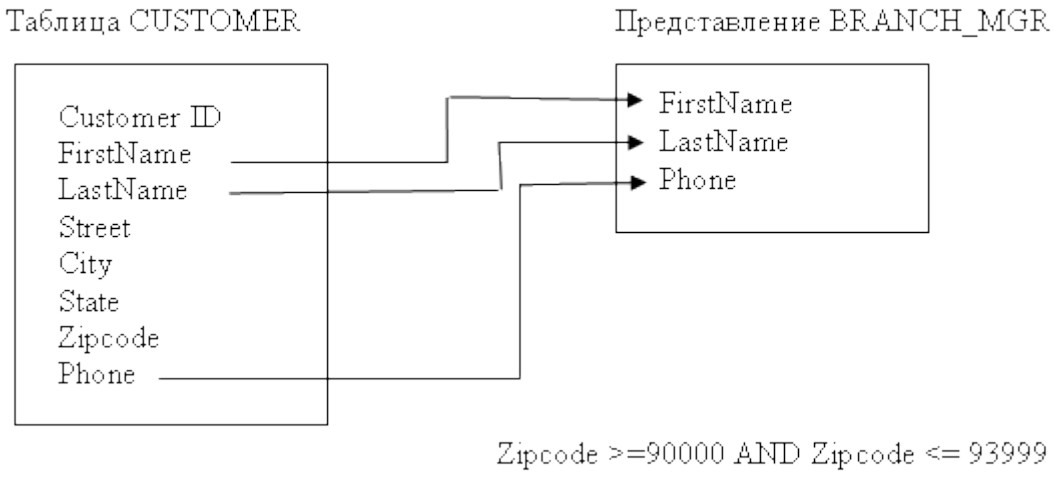


**Рис. 1.5**. В представлении для регионального менеджера по продажам имеются только некоторые строки из таблицы CUSTOMER

Менеджеру по оплате счетов, возможно, требуется просматривать имена и фамилии клиентов из таблицы CUSTOMER и значения столбцов Date, TotalSale, TotalRemitted и FormOfPayment из таблицы INVOICE, причем только из таких записей, где значение TotalRemitted меньше значения TotalSale. Это ограничение на записи имеет место тогда, когда оплата еще не проведена полностью. Для такого просмотра требуется создать представление, которое собирает данные из обеих таблиц. На рис. 1.6 показаны потоки данных, идущие из обеих таблиц, CUSTOMER и INVOICE, в представление для менеджера по оплате счетов, которое называется ACCTS_PAY.

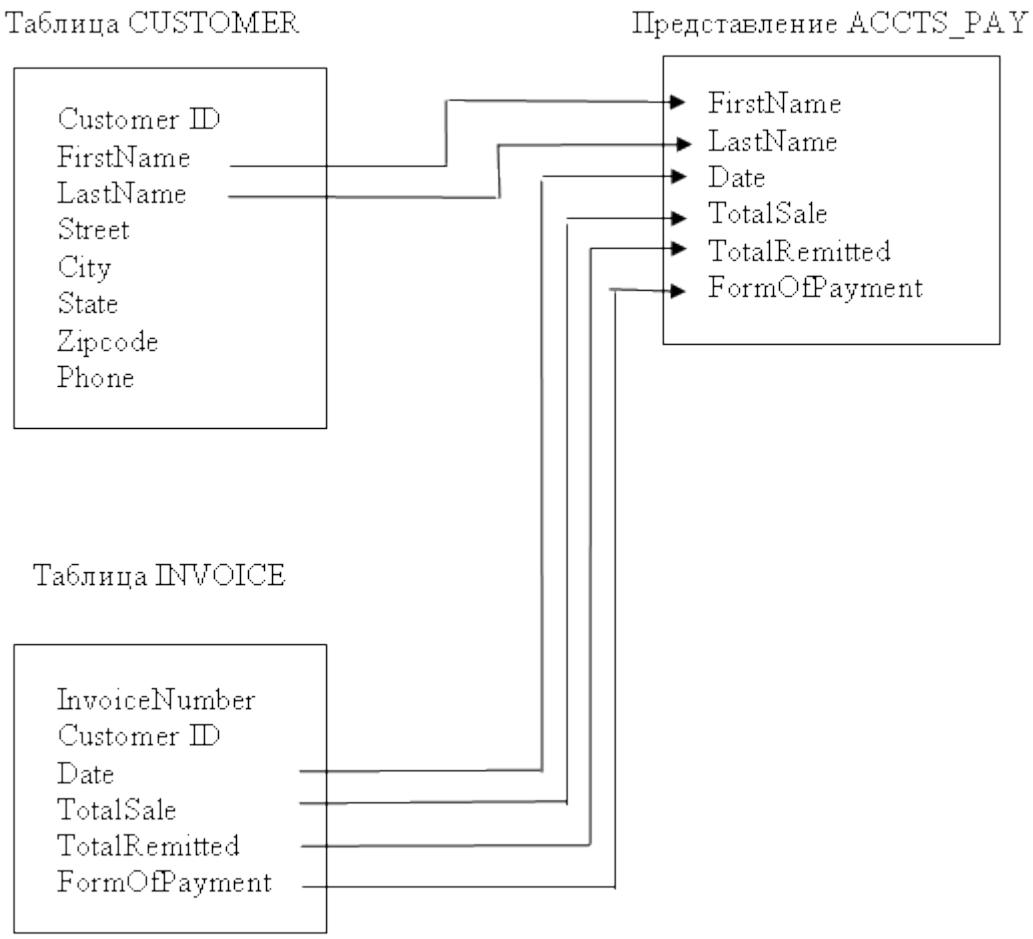


**Рис. 1.6**. Представление для менеджера по оплате счетов собирает данные из двух таблиц

Представления полезны потому, что дают возможность получать и выводить в определенном формате информацию из базы данных, при этом физически не меняя хранящиеся в ней данные. В главе 6 показано, как создать представление с помощью SQL.

## Схемы, домены и ограничения

**База данных** – это не только набор таблиц. В ней имеются и другие структуры; они помогают поддерживать на нескольких уровнях целостность данных. **Схема** базы данных дает таблицам общую организацию. **Домен** табличного столбца указывает, какие значения можно хранить в этом столбце. **Ограничения** на таблицу базы данных можно использовать для того, чтобы никто (включая и вас) не смог сохранять в таблице неправильные данные.

### Схемы

Структура всей базы данных – это ее схема или **концептуальное представление**. Кроме того, эта структура иногда называется **полным логическим представлением** базы данных. Схема представляет собой метаданные и в качестве таковых является частью базы данных. Сами метаданные, которые описывают структуру базы данных, хранятся в таблицах, похожих на те таблицы, в которых хранятся обычные данные. Метаданные – тоже данные, и в этом их прелесть.

## Домены

Атрибут отношения (т.е. столбец таблицы) может допускать некоторое конечное число значений. Множество всех таких значений и является **доменом** атрибута.

Скажем, например, что вы являетесь дилером по продаже автомобилей и торгуете новинкой – спортивным Curarri GT 4000 с двухместным закрытым кузовом. Данные об автомобилях, находящихся на складе, вы держите в базе данных, а именно в таблице, которую называете INVENTORY (наличные товары). Один из столбцов этой таблицы вы назвали Color (цвет), и в нем находятся данные о наружном цвете каждого автомобиля. GT 4000 поступают только четырех цветов: огненно-малиновый (**blazing crimson**), насыщенный черный (**midnight black**), снежно-белый (**snowflake white**) и металлический серый (**metallic grey**). Эти четыре цвета и представляют собой домен атрибута Color.

## Ограничения

Ограничения являются важным, хотя часто и недооцениваемым компонентом базы данных. Ограничения – это правила, определяющие допустимые значения для атрибутов таблицы.

Предотвратить ввод в столбец неправильных данных можно, применяя к этому столбцу жесткие ограничения. Конечно, любое значение, правильно входящее в домен столбца, должно удовлетворять всем ограничениям, которые имеются для этого столбца. Как уже говорилось в предыдущем разделе, доменом столбца является множество всех тех значений, которые могут быть в этом столбце. Ограничение – это именно ограничение на то, что может находиться в столбце. Характеристики табличного столбца и применяемые к нему ограничения определяют домен этого столбца. Применяя ограничения, можно предотвратить ввод в столбец таких данных, которые не входят в его домен.

Что касается примера с автомобилями, то можно применить к таблице ограничение, при котором в столбец Color (цвет) можно будет вводить только одно из перечисленных значений. Если после этого оператор ввода данных попытается ввести в этот столбец такое значение, как, например, темно-зеленый, система это значение не примет. Ввод данных нельзя будет продолжить до тех пор, пока оператор не введет в поле Color правильное значение.

## Объектная модель бросает вызов реляционной

Реляционная модель имела невероятный успех в большом количестве самых разных прикладных областей. Однако "беспроблемной" ее назвать нельзя. Проблемы сильнее проявились из-за роста популярности объектно-ориентированных языков программирования, таких как C++, Java и C#. Такие языки могут решать более сложные задачи, чем обычные языки программирования. Это достигается благодаря таким возможностям, как расширяемые пользователями системы типов, инкапсуляция, наследование, динамическое связывание методов, сложные и составные объекты и объектная целостность. Мы не собираемся в этой книге объяснять, что означают все эти понятия (хотя позднее затронем некоторые из них). Достаточно сказать, что со многими из этих особенностей классическая реляционная модель не совсем хорошо стыкуется. В результате были разработаны и появились на рынке системы управления базами данных, работающие на основе объектной модели. Впрочем, пока что их доля на рынке относительно невелика.

## Объектно-реляционная модель

Проектировщики баз данных, как и все остальные люди, постоянно ищут самый лучший из всех возможных миров. Они размышляют: "Не правда ли, было бы прекрасно, чтобы у нас были преимущества систем объектно-ориентированных баз данных и одновременно сохранялась совместимость с реляционной системой, которую мы узнали и полюбили?"

Такого рода размышления и привели к созданию гибридной объектно-реляционной модели. Объектно-реляционные СУБД расширяют реляционную модель настолько, чтобы в ней можно было поддерживать объектно-реляционное моделирование данных. Объектно-ориентированные модели были добавлены в международный стандарт SQL для того, чтобы поставщики реляционных СУБД могли перевести свои продукты в объектно-реляционные СУБД, сохраняя при этом совместимость со стандартом. Таким образом, в то время как стандарт SQL – 92 описывает чисто реляционную модель баз данных, SQL: 1999 (ранее известный как SQL 3) описывает объектно-реляционную модель. А SQL:2003 описывает даже больше – объектно-ориентированные возможности.

В этой книге описывается международный стандарт SQL, подготовленный ISO / IEC. В основном он соответствует реляционной модели. Кроме того, в ней рассказывается и об объектно-ориентированных расширениях стандарта, которые появились благодаря SQL: 1999, и дополнительных расширениях, включенных в SQL:2003. Объектно-ориентированные возможности позволяют разработчикам решать с помощью баз данных SQL проблемы, которые не по зубам старой, чисто реляционной парадигме. (Чуть язык не сломал.)

### Соображения по поводу проектирования баз данных

База данных является представлением физической или умозрительной структуры, такой, например, как организация, автосборочное производство или статистика выступлений всех бейсбольных клубов высшей лиги. Точность представления зависит от уровня детализации, достигнутого в проекте базы данных. Сколько усилий прикладывать к этому проекту – это должно зависеть от типа информации, которую вы собираетесь получать из базы данных. Слишком большая детализация – это напрасная трата усилий, времени, да и места на жестких дисках. А слишком малая детализация может свести ценность базы данных на нет. Решайте, какая степень детализации вам нужна сейчас, а какая может потребоваться в будущем, а затем воплотите этот уровень в свой проект (не больше и не меньше). Впрочем, не удивляйтесь, если придется этот уровень корректировать в соответствии с постоянно меняющимися условиям повседневной жизни.

Сегодняшние системы управления базами данных, снабженные привлекательными графическими пользовательскими интерфейсами и интуитивными инструментами проектирования, могут навеять претенденту на звание проектировщика ложное чувство безопасности. Такие системы приводят к тому, что проектирование базы данных кажется сравнимым с созданием электронной таблицы или с выполнением другой, относительно простой задачи. Но это не тот случай. Проектировать базу данных трудно. Если будете неправильно проектировать, то получите базу, которая со временем станет только хуже. Часто проблема не проявляется до тех пор пока вы не потратите немалые усилия на ввод данных. И к тому времени, когда проблема себя проявит, она станет достаточно серьезной. Во многих случаях решение состоит в том, чтобы полностью перепроектировать базу и заново ввести все данные. Единственным утешением при этом будет приобретаемый опыт работы с базами данных.

# Основы SQL

В этой главе…

- Что такое SQL
- Заблуждения, связанные с SQL
- Взгляд на разные стандарты SQL
- Знакомство со стандартными командами и зарезервированными словами SQL
- Представление чисел, символов, дат, времени и других типов данных
- Неопределенные значения и ограничения
- Использование SQL в системе клиент/сервер
- SQL в сети

---

**SQL** – это гибкий язык, который можно использовать самыми разными способами. Он является самым распространенным инструментом, используемым для связи с реляционной базой данных. В этой главе я объясню, чем является SQL и чем он не является, в частности, чем SQL отличается от компьютерных языков других типов. Затем вы познакомитесь с командами и типами данных, которые поддерживает стандартный SQL. Кроме того, я объясню такие основные понятия, как **неопределенные значения** и **ограничения**. И, наконец, будет дан обзор того, как SQL вписывается в среду клиент/сервер, а также в Internet и интранет-сети организаций.

## Чем является SQL и чем он не является

Первое, что надо уяснить насчет SQL, – этот язык не является **процедурным**, как FORTRAN, Basic, C, COBOL, Pascal и Java. Чтобы решить задачу с помощью одного из этих процедурных языков, приходится писать процедуру, которая выполняет одну за другой указанные операции, пока выполнение задачи не будет закончено. Процедура может быть линейной последовательностью или содержать ветвление, но в любом случае программист указывает порядок выполнения.

Иными словами, SQL является **непроцедурным** языком. Чтобы с его помощью решить задачу, сообщите SQL, **что именно** вам нужно, как если бы вы говорили с джином из лампы Аладдина. И при этом не надо говорить, **каким образом** получить для вас то, что вы хотите. Система управления базами данных (СУБД) сама решит, как лучше всего выполнить ваш запрос.

Хорошо. Только что я сказал, что SQL не является процедурным языком. В сущности, это правда. Однако миллионы программистов вокруг (и вы, возможно, один из них) привыкли решать задачи процедурным путем, поэтому в последние годы оказывалось немалое давление, чтобы дополнить SQL некоторыми процедурными возможностями. Поэтому теперь в составе новой версии спецификации SQL, SQL:2003, имеются такие средства процедурного языка, как блоки BEGIN, условные операторы IF, функции и процедуры. Благодаря этим новым средствам, можно хранить программы на сервере с тем, чтобы их могли повторно использовать многие пользователи.

Для иллюстрации того, что я имел в виду, когда говорил "сообщите системе, что именно вам нужно", предположим, что у вас имеется таблица EMPLOYEE с данными о служащих и вы хотите выбрать из нее все строки, соответствующие всем "старшим" работникам. Под "старшими" работниками можно подразумевать каждого, кто старше 40 лет или кто получает более 60000 долларов в год. Нужную вам выборку можно сделать с помощью следующего запроса:

```
SELECT * FROM EMPLOYEE WHERE AGE >40 OR SALARY >60000;
```

Этот оператор выбирает из таблицы EMPLOYEE все строки, в которых или значение столбца AGE (возраст) больше 40 или значение в столбце SALARY (зарплата) больше 60000. SQL сам знает, каким образом надо выбирать информацию. Ядро базы данных

проверяет базу и принимает для себя решение, каким образом следует выполнять запрос. Все, что от вас требуется, – указать, какие данные вам нужны.

***Помни:***

*Запрос – это вопрос, который вы задаете базе данных. Если какие-либо ее данные удовлетворяют условиям вашего запроса, то SQL передает их вам*.

В современных реализациях SQL отсутствуют многие простые программные конструкции, которые являются фундаментальными для большинства других языков. В приложениях для повседневной жизни, как правило, требуются хотя бы некоторые из этих конструкций, поэтому SQL на самом деле представляет собой **подъязык** данных. Даже имея дополнения, появившиеся в SQL вместе со стандартом SQL: 1999 и дополнительные расширения, добавленные в SQL:2003, все равно для создания законченного приложения необходимо использовать вместе с SQL один из программных языков, такой, например, как C.

Выбирать информацию из базы данных можно одним из следующих способов.

- **С помощью однократного непрограммируемого запроса с консоли компьютера, вводя команду SQL и читая на экране результаты ее выполнения. Консоль** – это традиционный термин, означающий компьютерное оборудование, которое выполняет работу клавиатуры и экрана, применяемых в современных ПК. Запросы с консоли пригодны тогда, когда требуется быстрый ответ на конкретный запрос. Для удовлетворения какой-либо текущей потребности вам могут потребоваться такие данные из базы, которые до этого никогда не требовались. Возможно, они вам никогда больше не понадобятся, но сейчас они нужны. Введите с клавиатуры соответствующий SQL-запрос, и через некоторое время на вашем экране появится результат.
- **С помощью программы, которая извлекает из базы данных информацию, а затем создает на основе этих данных отчет, выводимый или на экран, или на печать**. Язык SQL можно использовать и так. Сложный запрос SQL, который, возможно, еще пригодится в будущем, можно поместить прямо в программу. Это позволяет многократно использовать его в дальнейшем. Таким образом, формулировка запроса выполняется один раз. Как вставлять код SQL в программы, написанные на другом языке, рассказывается в главе 15.

## (Очень) Короткая история

Язык SQL, как и теория реляционных баз данных, берет свое начало в одной из исследовательских лабораторий компании IBM. В начале 1970-х годов исследователи из IBM выполняли первые разработки реляционных систем СУБД (или РСУБД), и тогда они создали подъязык данных, предназначенный для работы в этих системах. Пробная версия этого подъязыка была названа **SEQUEL** (**S**tructured **E**nglish **QUE**ry **L**anguage – структурированный английский язык запросов). Однако, когда пришло время официально выпускать их язык запросов в качестве продукта, разработчики захотели сделать так, чтобы люди понимали, что выпущенный продукт отличается от пробной системы СУБД и превосходит ее. Поэтому они решили дать выпускаемому продукту имя, хотя и отличающееся от SEQUEL, но явно принадлежащее к тому же семейству. Так что они назвали его SQL.

О работе, которая велась в IBM над реляционными базами данных и над языком SQL, в информационной отрасли хорошо знали, причем еще до того, как эта компания представила в 1981 году РСУБД SQL / DS. К этому времени компания Relational Software, Inc. (ныне Oracle Corporation) уже выпустила свою первую РСУБД. Эти первоначальные продукты тут же стали стандартом для нового класса систем, предназначенных для управления базами данных. В состав этих продуктов вошел SQL, который фактически стал стандартом для подъязыков данных. Производители других систем управления базами

данных выпустили свои собственные версии SQL. В этих реализациях обычно имелись все основные возможности продуктов IBM, но, впрочем, не только они. Там также имелись расширения, введенные с целью дать преимущество сильным сторонам именно "своей" РСУБД. В результате, хотя почти все поставщики и использовали варианты одного языка SQL, платформенная совместимость была слабой.

***Помни:***
*Реализация – это конкретная СУБД, работающая на конкретной аппаратной платформе*.

Вскоре началось движение за создание общепризнанного стандарта SQL, которого мог бы придерживаться каждый. В 1986 году организация ANSI выпустила официальный стандарт под названием SQL – 86. Этот стандарт был обновлен той же организацией в 1989 году и получил название SQL – 89, а затем, в 1992 году, был назван SQL – 92. Поставщики СУБД, выпуская новые версии своих продуктов, всегда старались приблизить свои реализации к стандарту. Эти усилия и привели к тому, что мечта о настоящей переносимости SQL стала намного ближе к реальности.

Самой последней версией стандарта SQL является SQL:2003 (ISO / IEX 9075 X:2003). В этой книге описан язык SQL, который определяется стандартом SQL:2003. Конечно, любая конкретная реализация SQL в определенной степени отличается от стандарта. Так как полный стандарт SQL:2003 является слишком всеобъемлющим, то от современных реализаций, видимо, не стоит ждать полного ему соответствия. Однако поставщики систем СУБД сейчас работают над тем, чтобы эти системы все же соответствовали основной части стандартного SQL. Полные спецификации стандартов ISO / IEC доступны в Internet по адресу webstore.ansi.org.

## Команды SQL

Язык SQL состоит из ограниченного числа команд, специально предназначенных для управления данными. Одни из этих команд служат для определения данных, другие – для их обработки, а остальные – для администрирования данных. О командах определения и обработки данных рассказывается в главах 4-12, а о командах администрирования данных – в главах 13 и 14.

Чтобы соответствовать стандарту SQL: 2003, в состав реализации должны входить все основные возможности. Кроме того, в ее состав могут входить и расширения этого основного набора (которые также описаны спецификацией SQL:2003). Расширения пока оставим, вернемся к основам. Ниже приведена таблица основных команд SQL:2003.

Если вы из тех программистов, кому нравится проверять новые возможности, то возрадуйтесь.

**Таблица 2.1**. Основные команды SQL:2003.

| | | |
|---|---|---|
| ALTER DOMAIN | CREATE CURSOR | FREE LOCATOR |
| ALTER TABLE | DECLARE TABLE | GET DIAGNOSTICS |
| CALL | DELETE | GRANT |
| CLOSE | DISCONNECT | HOLD LOCATOR |
| COMMIT | DROP ASSERTION | INSERT |
| CONNECT | DROP CHARACTER SET | OPEN |
| CREATE ASSERTION | DROP COLLATION | RELEASE SAVEPOINT |
| CREATE CHARACTER SET | DROP DOMAIN | RETURN |
| CREATE COLLATION | DROP ORDERING | REVOKE |
| CREATE DOMAIN | DROP ROLE | ROLLBACK |
| CREATE FUNCTION | DROP SCHEME | SAVEPOINT |
| CREATE METHOD | DROP SPECIFIC FUNCTION | SELECT |
| CREATE ORDERING | DROP SPECIFIC PROCEDURE | SET CONNECTION |
| CREATE PROCEDURE | DROP SPECIFIC ROUTINE | SET CONSTRAINTS |
| CREATE ROLE | DROP TABLE | SET ROLE |
| CREATE SCHEMA | DROP TRANSFORM | SET SESSION AUTHORIZATION |
| CREATE TABLE | DROP TRANSLATION | SET SESSION CHARACTERISTICS |
| CREATE TRANSFORM | DROP TRIGGER | SET TIME ZONE |
| CREATE TRANSLATION | DROP TYPE | SET TRANSACTION |
| CREATE TRIGGER | DROP VIEW | START TRANSACTION |
| CREATE TYPE | FETCH | UPDATE |
| CREATE VIEW | | |

### Зарезервированные слова

Кроме команд, специальное значение в SQL имеют и некоторые другие слова. Вместе с командами они зарезервированы для специального использования, поэтому эти слова нельзя применять в качестве имен переменных или любым другим способом, для которого они не предназначены. Вы можете легко увидеть, почему таблицам, столбцам и переменным нельзя давать имена из списка зарезервированных слов. Представьте себе, какая путаница возникнет из-за такого оператора:

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

Полный список зарезервированных слов стандарта SQL:2003 приводится в приложении А.

### Типы данных

В разных реализациях SQL поддерживаются различные исторически сложившиеся типы данных. В спецификации SQL:2003 признаны только пять заранее определенных общих типов:

- числовой;
- строковый;

- логический;
- даты-времени;
- интервальный.

Внутри каждого из этих типов может быть несколько подтипов (точный числовой; приблизительный числовой; символьный строковый; битовый строковый; строковый для больших объектов). Кроме встроенных, заранее определенных типов, в SQL:2003 также поддерживаются сконструированные и определяемые пользователем типы.

***Совет:***

*Если вы используете реализацию SQL, в которой поддерживаются типы данных, не описанные в SQL:2003, то для большей переносимости вашей базы данных старайтесь этими типами данных не пользоваться. Перед тем как вы решите создать и использовать определяемый пользователем тип, убедитесь, что в любой СУБД, на которую вам, возможно, захочется перейти в будущем, также поддерживаются определяемые пользователем типы*.

### Точные числовые типы

Как вы, возможно, поняли из названия, точные числовые типы данных позволяют точно выразить значение числа. К этой категории относятся пять типов:

- INTEGER
- SMALLINT
- BIGINT
- NUMERIC
- DECIMAL

### Тип INTEGER

В данных типа **INTEGER** (целый) нет дробной части, и их точность зависит от конкретной реализации SQL. Таким образом, точность не может быть установлена разработчиком базы данных.

***Помни:***

*Точностью числа является максимальное количество цифр, которое у него может быть*.

### Тип SMALLINT

Тип **SMALLINT** (малый целый) также предназначен для целых значений, но его точность в конкретной реализации не может быть больше точности типа INTEGER, имеющейся в данной реализации. В реализациях, работающих на компьютерах IBM System /370, типы SMALLINT и INTEGER обычно представлены соответственно 16 – и 32-битовыми двоичными числами. Впрочем, во многих реализациях SMALLINT и INTEGER являются одним и тем же типом.

Если в таблице из базы данных вы определяете столбец для целых данных и известно, что значения в этом столбце никогда не превысят точность, установленную в вашей реализации для значений типа SMALLINT, то присвойте столбцу тип не INTEGER, а SMALLINT. Таким образом вы, возможно, позволите своей СУБД сэкономить место на диске.

### Тип BIGINT

Тип данных **BIGINT** (большой целый) – это новый тип данных, появившийся вместе с SQL:2003. Он также предназначен для целых значений и определяется как тип, точность которого может быть не намного больше, чем точность данных типа INTEGER, или сильно превышать ее. Предел точности данных типа BIGINT зависит от реализации.

### Тип NUMERIC

В данных типа **NUMERIC** (числовой), кроме целого компонента, может быть и дробный. Для этих данных можно указать точность и масштаб. Точность, как вы помните, – это максимально возможное количество цифр.

**Масштаб** – это количество цифр после запятой. Масштаб не может быть отрицательным или превышать точность числа.

При определении типа NUMERIC необходимо указать требуемые значения точности и масштаба. В определении можно указать только NUMERIC и получить значения по умолчанию. А если вы укажете NUMERIC (p), то получите требуемую точность и значение масштаба по умолчанию. Выражение NUMERIC (p,s) позволяет непосредственно задать и точность, и масштаб. При определении данных вместо параметров p и s нужно ввести соответственно требуемые значения точности и масштаба.

Скажем, например, что в вашей реализации SQL точность по умолчанию для типа данных NUMERIC равна 12, а масштаб по умолчанию равен 6. Если вы укажете, что столбец базы данных имеет тип NUMERIC, то в этом столбце смогут находиться числа вплоть до 999999.999999. Если же, с другой стороны, вы для столбца указываете тип данных NUMERIC (10), то в столбце смогут находиться только числа с максимальным значением 9999.999999. Параметр (10) указывает максимально возможное для числа количество цифр. Когда для столбца будет указан тип данных NUMERIC (10.2), то в столбце могут находиться числа с максимальным значением – 99999999.99. В этом случае хотя и останется всего десять цифр, но справа от десятичной запятой будут находиться только две из них. (Имейте в виду, что синтаксис SQL требует использовать для разделения целой и дробной частей числа не запятую, а десятичную точку, как это принято в США и странах Европы.)

Тип данных NUMERIC предназначен для значений, таких как 595.72. Точность этого значения равна 5 (общее количество цифр), а масштаб – 2 (количество цифр справа от десятичной запятой). Для чисел, таких, как это, подходит тип данных NUMERIC (5.2).

### Тип DECIMAL

Тип данных **DECIMAL** (десятичный) похож на NUMERIC. В нем может быть дробная часть, и для него можно указать точность и масштаб. DECIMAL отличается от NUMERIC тем, что если точность имеющейся реализации SQL будет больше указанного значения, то в реализации будет использоваться большая точность. А если точность или масштаб не указаны, то, как и при использовании типа NUMERIC, в реализации применяются значения по умолчанию.

В столбце, тип которого NUMERIC (5.2), нельзя поместить числа, большие 999.99. Если же тип столбца DECIMAL (5.2), в него всегда можно поместить значения до 999.99. Кроме того, если точность реализации позволяет, то СУБД сможет поместить в этот столбец и значения, большие, чем 999.99.

Если в ваших данных имеются дробные части, тогда применяйте тип NUMERIC или DECIMAL, а если ваши данные всегда состоят из целых чисел, то используйте тип INTEGER или SMALLINT. Когда нужно добиться максимальной переносимости, следует использовать тип NUMERIC, потому что поле, тип которого вы определите, например, как NUMERIC (5.2), будет во всех системах иметь один и тот же диапазон значений.

### Приблизительные числовые типы

У некоторых величин имеется такой большой диапазон возможных значений (большое количество порядков), что компьютер с данным размером регистра не может в точности представить все эти значения. (Размерами регистра являются, например, 32 бита, 64 бита и 128 бит.) Обычно в таких случаях точность не является необходимой, и поэтому будет достаточно иметь близкое приближение. Для работы с такими данными SQL:2003 определяет три приблизительных числовых типа.

## Тип REAL

Тип данных REAL (действительное число) дает возможность задавать числа однократной точности с плавающей запятой, точность которых зависит от реализации. Вообще-то, точность определяется используемым оборудованием. Например, 64-битовая машина дает большую точность, чем 32-битовая.

**Число с плавающей запятой** (floating – point number) – это число с десятичной запятой Десятичная запятая "плавает" или появляется в разных частях числа, в зависимости от значения этого числа. Примерами чисел с плавающей запятой являются 3.1, 3.14 и 3.14159.

## Тип DOUBLE PRECISION

Тип данных **DOUBLE PRECISION** (двойная точность) дает возможность задавать числа двойной точности с плавающей запятой, точность которых опять-таки зависит от реализации Удивительно, что само значение слова DOUBLE (двойной) также зависит от реализации Арифметика двойной точности в основном применяется в научных целях. А для разных научных дисциплин требуется разная точность. Некоторые реализации SQL обслуживают одну категорию пользователей, а другие реализации – соответственно другие категории.

В некоторых системах тип DOUBLE PRECISION имеет и для мантиссы, и для экспоненты как раз в два раза большую вместимость, чем тип REAL. (Если вы забыли то, что учили в средней школе, то вспомним, что любое число можно представить в виде **мантиссы**, умноженной на число десять, возведенное в степень, показатель которой является**экспонентой**. Например, 6626 можно написать в виде 6.626E3. Число 6.626 является мантиссой, которую вы умножаете на десять, возведенное в третью степень (в данном случае 3 является экспонентой).

Вы не получите выигрыша, если будете с помощью приблизительного числового типа данных представлять числа, достаточно близкие к единице (такие как 6626 или даже 6626000). Точные числовые типы работают точно так же и занимают в памяти меньше места. Однако для чисел, которые слишком близко находятся от нуля или слишком далеко от единицы, таких как 6.626E-34 (очень малое число), необходимо использовать приблизительный числовой тип. Для таких величин не подходят точные числовые типы. В других системах тип DOUBLE PRECISION дает побольше, чем двойная вместимость мантиссы, и поменьше, чем двойная вместимость типа REAL для экспоненты. В системах следующего вида тип DOUBLE PRECISION дает для мантиссы вместимость в два раза больше, чем у типа REAL, а для экспоненты – ту же, что и у REAL. В этом случае точность удваивается, а диапазон – нет.

Спецификация SQL:2 OO 3 не пытается жестко определить, что означает DOUBLE PRECISION. Она только требует, чтобы точность числа типа DOUBLE PRECISION была больше, чем точность типа REAL. Хотя это ограничение очень слабое, но, возможно, оно является наилучшим из возможных, если учитывать те немалые различия, которые имеются в оборудовании.

## Тип FLOAT

Тип данных FLOAT (плавающий) является самым полезным, если вы считаете, что ваша база данных однажды должна перейти на аппаратную платформу, в которой размеры регистров отличаются от размеров регистров платформы, для которой вы первоначально спроектировали базу. Используя тип данных FLOAT, можно указывать точность– например, FLOAT (5). Если ваше оборудование поддерживает указанную точность, используя аппаратную одинарную точность, то ваша система использует арифметику с одинарной

точностью. Если указанная точность требует арифметики с двойной точностью, то система использует ее.

***Совет:***

*Использование FLOAT вместо REAL или DOUBLE PRECISION облегчает перенос ваших баз данных на другие системы, так как тип данных FLOAT дает возможность указывать точность. Точность чисел типа REAL или DOUBLE PRECISION зависит от аппаратуры*.

Если вы не уверены, использовать точные числовые типы данных (NUMERIC/DECIMAL) или приблизительные числовые типы (FLOAT / REAL), то выбирайте точные. Точным типам данных требуется меньше системных ресурсов, к тому же они дают точные, а не приблизительные результаты. Если же диапазон возможных значений ваших данных настолько большой, что требуется использовать приблизительные типы данных, то этот факт вы, скорее всего сможете определить для себя заранее.

### Символьные строки

Сейчас в базах данных хранятся данные многих разных типов, в том числе графические изображения звуки и анимация. Надеюсь, что вслед за ними появятся и запахи. Можете вы представить на своем экране трехмерное, размером 1600x1200 пикселей, выполненное в 24-битовом цвете изображение большого ломтя наперченной пиццы, в то время как через вашу супермультимедийную плату воспроизводится фрагмент запаха, записанный в "Ди Филиппи Пицца Гротто"? Такое мультимедиа может вызывать одно лишь разочарование – хотя бы до тех пор пока в систему нельзя будет вводить и данные вкусовых ощущений. Увы, возможно придется ждать очень долго, пока запах и вкус не станут стандартными типами данных SQL. А сейчас типами данных, которые вам придется использовать чаще всего, не считая, конечно, числовых, являются типы символьных строк.

Есть три главных типа символьных данных: фиксированных символьных данных (CHARACTER или CHAR), переменных символьных данных (CHARACTER VARYING или VARCHAR) и данных для больших символьных объектов (CHARACTER LARGE OBJECT или CLOB). Кроме того, есть еще три варианта этих типов данных: **NATIONAL CHARACTER**(строка с национальными символами), **NATIONAL CHARACTER VARYING** (переменная строка с национальными символами) и **NATIONAL CHARACTER LARGE OBJECT** (большой объект с национальными символами).

### Тип CHARACTER

Если вы определяете для столбца тип данных CHARACTER или CHAR, количество символов, которое будет в нем находиться, можно указать, используя синтаксис CHARACTER (x), где x и является нужным вам количеством. Если, например, тип данных столбца был определен как CHARACTER (16), то максимальная длина любых данных, которые можно будет ввести в этот столбец, равна 16 символам. Если аргумент не указан (т.е. нет значения в скобках), тогда для SQL это означает, что длина поля равна одному символу. Если вы вводите данные в поле типа CHARACTER, имеющего определенную длину, и при этом вводите символов меньше, чем может поместиться в поле, то позиции, оставшиеся свободными, будут заполнены пробелами.

### Тип CHARACTER VARYING

Тип данных CHARACTER VARYING полезен тогда, когда вводимые в столбец значения имеют разную длину, но вы не хотите, чтобы поле заполнялось пробелами. Этот тип данных Дает возможность сохранять то количество символов, которое ввел пользователь. Для типа CHARACTER VARYING нет значения по умолчанию. Чтобы указать этот тип данных, используйте синтаксис CHARACTER VARYING (x) или VARCHAR (x), где x – это максимальное разрешенное количество символов.

### Тип CHARACTER LARGE OBJECT

Тип данных CHARACTER LARGE OBJECT (CLOB) впервые появился в SQL: 1999. Как Указывает его имя (означает "большой символьный объект"), он используется вместе с громадными символьными строками, которые для типа CHARACTER слишком велики. Данные – LOB ведут себя во многом так же, как и обычные символьные строки, но на действия, которые можно с ними проводить, имеется ряд ограничений. Для типа CLOB нельзя использовать предикаты PRIMARY KEY (первичный ключ), FOREIGN KEY (внешний ключ), UNIQUE (уникальный). Более того, эти данные нельзя использовать для сравнения, за исключением равенства или неравенства. Из-за того, что у данных CLOB большие размеры, они как правило, всегда остаются на сервере. Вместо них на стороне клиента применяется специальный тип данных, который называется **локатор** LOB (LOB locator). Это параметр, значение которого идентифицирует большой символьный объект.

### Типы NATIONAL CHARACTER, NATIONAL CHARACTER VARYING и NATIONAL CHARACTER LARGE OBJECT

В разных языках используются символы, которые отличаются от любых символов другого языка. Например, в немецком имеются символы, которых нет в наборе символов английского языка. Имеются языки, такие, например, как китайский или японский, в которых набор иероглифов очень сильно отличается от наборов символов других языков. Если вы в своей системе сделаете английский языком по умолчанию, то все равно сможете использовать и другие символьные наборы, так как типы данных NATIONAL CHARACTER, NATIONAL CHARACTER VARYING и NATIONAL CHARACTER LARGE OBJECT работают точно так же, как и CHARACTER, CHARACTER VARYING и CHARACTER LARGE OBJECT, за исключением того, что определяемый вами набор символов отличается от того, который вы используете по умолчанию. Набор символов можно указывать при определении столбца в таблице. Если нужно, то в каждом столбце можно использовать отдельный набор символов. В следующем примере инструкции, создающей таблицу, используется несколько таких наборов:

```
CREATE TABLE XLATE
LANGUAGE_1 CHARACTER (40),
LANGUAGE_2 CHARACTER VARYING (40) CHARACTER SET GREEK,
LANGUAGE_3 NATIONAL CHARACTER (40),
LANGUAGE_4 CHARACTER (40) CHARACTER SET KANJI
);
```

Столбец LANGUAGE_1 (1-й язык) предназначен для символов из набора, используемого в данной реализации по умолчанию. В свою очередь, столбец LANGUAGE_3 (3-й язык) предназначен для символов национального набора данной реализации. А для греческих символов предназначен столбец LANGUAGE_2 (2-й язык). И, наконец, для иероглифов предназначен столбец LANGUAGE_4 (4-й язык).

### Логические данные

Тип данных BOOLEAN (булев) имеет два определенных логических значения, **true**(истина) и **false** (ложь), а также неопределенное значение **unknown** (неизвестное). Если любое из первых двух значений сравнить с NULL или со значением **unknown**, то в результате получится **unknown**.

### Данные типа даты-времени

В SQL:2003 определяются пять различных типов данных, которые относятся к дате и времени. Они называются **типами данных даты-времени** или просто **датой-временем**. Некоторые из этих типов данных довольно сильно перекрывают друг друга, поэтому в отдельных реализациях, которые могут вам встретиться, все пять типов, возможно, и не поддерживаются.

***Внимание:***

*В тех реализациях, в которых не полностью поддерживаются все пять типов данных для даты и времени, при переносе в них баз данных из других реализаций могут*

*возникнуть проблемы. Если с переносом возникли сложности, то проверьте, какие имеются способы представления даты и времени в двух реализациях исходной и той, на которую требуется перейти*.

### Тип DATE

Тип DATE (дата) предназначен для хранения значений даты в следующем порядке: год, и день. Значение года занимает четыре цифры, а месяца и дня – по две. Значения этого могут представлять любую дату, начиная с 0001 года и заканчивая 9999 годом. Длина DATE равна десяти позициям, как, например, для 1957-08-14.

Технические подробности: Так как язык SQL, используя тип DATE, представляет в явном виде все четыре цифры года, то данные SQL никогда не вызывали опасения из-за пресловутой проблемы 2000 года.

### Тип TIME WITHOUT TIME ZONE

Тип **TIME WITHOUT TIME ZONE** (время без часового пояса) предназначен для хранения значений времени в следующем порядке: час, минута и секунда. Значения часа и минуты занимают в точности по две цифры. Секундное значение может занимать две цифры, но может быть и расширено, чтобы иметь необязательную дробную часть. Поэтому время 9 часов 32 минуты и 58.436 секунды утра представляется с помощью этого типа данных как 09:32:58.436.

Точность дробной части зависит от конкретной реализации, но имеет длину не менее шести символов. Значение типа TIME WITHOUT TTME ZONE без дробной части занимает восемь позиций (включая двоеточия), а с дробной частью – девять позиций (вместе с десятичной запятой) плюс число цифр дробной части. Этот тип данных задается или с помощью синтаксиса TME, в результате чего данные представляются в виде, установленном по умолчанию, т.е. без дробной части, или с помощью другого синтаксиса, TIME WITHOUT TIME ZONE (p), где вместо p должно стоять количество цифровых позиций, находящихся справа от десятичной запятой. В предыдущем абзаце представлен пример данных типа TIME WITHOUT TIME ZONE (3).

### Тип TIMESTAMP WITHOUT TIME ZONE

В данных типа **TIMESTAMP WITHOUT TIME ZONE** (дата и время без часового пояса) хранится информация как о дате, так и о времени. У компонентов данных этого типа такие же значения длины и такие же ограничения, как и для данных типа DATE и TIME WITHOUT TIME ZONE, если не считать одного различия. Оно состоит в том, что по умолчанию в данных типа TIMESTAMP WITHOUT TIME ZONE длина дробной части равна шести цифрам, а не нулю. Если в значении типа TIMESTAMP WITHOUT TIME ZONE нет цифр дробной части, то длина этого значения равна 19 позициям, занимаемым в следующем порядке: десять позиций – датой, один пробел служит в качестве разделителя, и восемь позиций – временем. Если цифры дробной части все же имеются (по умолчанию их должно быть шесть), то Длина равна 20 позициям плюс количество этих цифр. Двадцатая позиция предназначена для Десятичной запятой. Устанавливать для поля тип TIMESTAMP WITHOUT TIME ZONE можно с помощью двух видов синтаксиса: TIMESTAMP WITHOUT TIME ZONE или IMESTAMP WITHOUT TIME ZONE (p), где вместо p должно стоять число позиций, предназначенных для цифр дробной части. Вместо p не может стоять отрицательное число, максимальное значение этого параметра зависит от конкретной реализации SQL.

### Тип TIME WITH TIME ZONE

Тип данных **TIME WITH TIME ZONE** (время вместе с часовым поясом) в точности такой же как и TIME WITHOUT TIME ZONE, за исключением того, что в нем еще имеется информация о разнице между местным и **всемирным временем** (UTC, Universal Time), ранее известным как среднее время по Гринвичу (Greenwich Mean Time, GMT). Значение

этой разницы может находиться в диапазоне от – 12:59 до +13:00. Такая дополнительная информация занимает за цифрами времени шесть позиций: дефис в качестве разделителя, знак "плюс" или "минус", а затем разница в часах (две цифры) и минутах (также две цифры) и двоеточие между часами и минутами. Значение типа TIME WITH TIME ZONE без дробной части (как установлено по умолчанию) занимает 14 позиций. Если же дробная часть указана, то длина поля равняется 15 позициям плюс количество цифр дробной части.

### Тип TIMESTAMP WITH TIME ZONE

Тип данных **TIMESTAMP WITH TIME ZONE** (дата и время вместе с часовым поясом) работает точно так же, как и TIMESTAMP WITHOUT TIME ZONE, за исключением того, что в нем еще имеется информация о разнице между местным и всемирным временем. Дополнительная информация занимает за датой и временем шесть позиций (о формате данных часового пояса см. в предыдущем разделе). Для поля с данными часового пояса и без дробной части требуется 25 позиций, а для поля с дробной частью нужно 26 позиций плюс количество цифр дробной части (это количество по умолчанию равно шести).

### Интервалы

**Интервальные** типы данных тесно связаны с типами данных даты-времени. Интервал – это разница между двумя значениями даты-времени. Во многих приложениях, имеющих дело со значениями даты, времени или с теми и другими, иногда требуется определить интервал между двумя датами или значениями времени. SQL поддерживает два различных типа интервалов: **год-месяц и день-время**. Интервал год-месяц – это количество лет и месяцев между двумя датами. А интервал день-время – это количество дней, часов, минут и секунд между двумя моментами в пределах одного месяца. Нельзя смешивать вычисления, где используется интервал год-месяц, с вычислениями, где используется интервал день-время, потому что в месяцах разное количество дней (28, 29, 30 или 31).

### Типы ROW

Тип данных **ROW** (запись) впервые появился в SQL: 1999. Он не является легким для понимания, и вы на своем пути от начального до среднего уровня программирования на SQL, возможно, так никогда с этим типом данных не встретитесь. В конце концов, в 1986-1999 годах люди прекрасно без него обходились.

Кроме всего прочего, примечательно то, что тип данных ROW нарушает правила нормализации, которые доктор И.Ф. Кодд (**E.F. Codd**) объявил на начальной стадии теории реляционных баз данных. Об этих правилах более подробно рассказывается в главе 5. Одной из характеристик, определяющих первую нормальную форму, является то, что поле в табличной строке не может быть многозначным. В поле может находиться одно и только одно значение. Однако тип данных ROW дает возможность объявить целую строку данных находящейся в единственном поле единственной строки таблицы – другими словами, объявить строку, вложенную в строку.

Проанализируйте следующую команду SQL, которая для персональной адресной информации (улица, город, штат, почтовый код) использует тип ROW:

| CREATE ROW TYPE addr_typ ( | |
|---|---|
| Street | CHARACTER VARYING (25) |
| City | CHARACTER VARYING(20) |
| State | CHARACTER (2) |
| PostalCode | CHARACTER VARYING (9) |
| ); | |

После того как тип ROW определен, его можно использовать в определении таблицы (с идентификатором клиента, его фамилией, именем, адресом, телефоном):

| CREATE TABLE CUSTOMER ( | | |
|---|---|---|
| CustID | INTEGER | PRIMARY KEY, |
| LastName | CHARACTER | VARYING (25), |
| FirstName | CHARACTER | VARYING (20), |
| Address | addr_typ | |
| Phone | CHARACTER | VARYING (15) |
| ); | | |

Здесь преимущество состоит в том, что если в базе данных много таблиц с информацией об адресе, например отдельно для клиентов, поставщиков, персонала и акционеров, то детали адресной спецификации необходимо определить только один раз – в определении типа ROW.

## Типы коллекций

После того как с выходом SQL: 1999 была разрушена реляционная строгость, стало возможным использование таких типов данных, которые нарушают первую нормальную форму. Теперь поля могут содержать не один, а целую коллекцию объектов. Тип **ARRAY**(массив) впервые появился уже в SQL: 1999, а вот тип **MULTISET** (мультимножество) – это новинка SQL:2003.

Можно сравнить две коллекции между собой только в том случае, если они содержат один и тот же тип данных, либо ARRAY, либо MULTISET, и сравниваемые типы элементов. Поскольку массивы имеют определенный порядок элементов, соответствующие элементы массивов можно сравнить. Мультимножества такого порядка не имеют, однако их также можно сравнить, но только в том случае, если для одного из двух сравниваемых мультимножеств существует нумерация, которой должна соответствовать нумерация другого мультимножества.

## Типы ARRAY

Тип данных **ARRAY** (массив) нарушает первую нормальную форму (1НФ), но делает это иначе, чем тип ROW. Тип ARRAY, относящийся к типу коллекций, в действительности не является отдельным типом данных в том же смысле, что и типы CHARACTER или NUMERIC. Тип ARRAY просто дает возможность одному из других типов иметь множество значений внутри одного поля в таблице. Скажем, например, что для вашей организации важно поддерживать контакт со своими клиентами, находятся ли они на работе, дома или в дороге. Поэтому среди данных о клиенте может потребоваться и несколько телефонных номеров. Такую возможность можно реализовать, объявив атрибут Phone (телефон) массивом, как показано в следующем коде:

| CREATE TABLE CUSTOMER ( | | |
|---|---|---|
| CustID | INTEGER | PRIMARY KEY, |
| LastName | CHARACTER | VARYING (25), |
| FirstName | CHARACTER | VARYING (20), |
| Address | addr_typ | |
| Phone | CHARACTER | VARYING (15) ARRAY [3] |

```
);
```

Это дает возможность хранить в таблице **CUSTOMER** (клиент) до трех телефонных номеров каждого клиента. Три телефонных номера являются примером повторяющейся группы. Если следовать классической теории реляционных баз данных, то такие группы делать нельзя, однако они – это только один из нескольких примеров нарушения правил со стороны SQL: 1999. Когда доктор И.Ф. Кодд впервые провозгласил свои правила нормализации, то ради целостности данных он пожертвовал функциональной гибкостью. SQL: 1999 вернул назад часть этой гибкости – впрочем, за счет некоторого усложнения структуры. Такое усложнение может привести к ослаблению целостности данных. Это вам напоминание на тот случай, если вы еще не полностью осознаете последствия всех действий, выполняемых с базой данных. Массивы упорядочены таким образом, что каждый элемент массива связан только с одной порядковой позицией в массиве.

### Тип MULTISET

Тип данных **MULTISET** (мультимножество) – это неупорядоченная коллекция. Определенные элементы мультимножества могут быть не связаны, поскольку им не назначается определенная порядковая позиция в мультимножестве.

### Типы REF

Типы REF в основную часть SQL не входят. Это значит, что какая-либо система СУБД, совсем не использующая эти типы данных, все равно может считаться соответствующей стандарту SQL:2003. Тип REF не является отдельным типом данных в том же смысле, что и типы CHARACTER и NUMERIC. На самом деле это указатель на элемент данных, на данные типа ROW или на данные абстрактного типа, размещаемого в строке таблицы. Разыменовывая указатель, можно получить требуемое значение. Не беспокойтесь, если запутались, – в этом вы не одиноки. Использование типов REF требует практического знания принципов объектно-ориентированного программирования (ООП). В книге я старался не слишком глубоко погружаться в эту муть. Поскольку типы REF не входят в основную часть SQL, вам будет лучше совсем их не использовать. Если нужна максимальная переносимость с одних платформ СУБД на другие, то не следует выходить за пределы основной части SQL.

### Определяемые пользователем типы

**Определяемые пользователем типы** (user-defined types, UDT) – это еще один пример новых возможностей, пришедших в SQL: 1999 из мира объектно-ориентированного программирования. Как программист на языке SQL вы больше не ограничены теми типами данных, которые определяются в спецификации SQL:2003. У вас есть возможность определять собственные типы, используя принципы абстрактных типов данных (**Abstract Data Types**, ADT), таких объектно-ориентированных языков программирования, как C++.

Одним из самых важных преимуществ UDT-типов является то, что их можно использовать для устранения "нестыковок" между SQL и базовым языком приложения баз данных. Застарелой проблемой SQL является то, что его заранее определенные типы данных могут не соответствовать типам, которые используются в базовом языке. Теперь же программист баз данных может создавать в SQL такие типы данных, которые не нарушают этого соответствия. UDT-типы имеют инкапсулированные атрибуты и методы. Снаружи можно видеть определение атрибутов и результаты выполнения методов, но конкретные механизмы этого выполнения скрыты от пользователя. Доступ к атрибутам и методам можно еще больше ограничить, указав, что они являются общими (**public**), приватными (**private**) или защищенными (**protected**). Общие атрибуты или методы доступны всем пользователям UDT, в то время как приватные – только самому UDT. Защищенные атрибуты или методы доступны только самому UDT-типу или его подтипам. Из этого можно видеть, что в SQL UDT-тип ведет себя во многом так же, как и класс из объектно-

ориентированного программного языка. Существуют две формы определяемых пользователем типов: отдельный и структурный.

### Отдельные типы данных

Отдельные типы данных – это простейшие формы определяемых пользователем типов. Отдельные типы определяются той особенностью, что они выражаются как единый тип данных. Они создаются на основе одного из ранее определенных типов данных, называющихся **базовыми типами**. Множество отдельных типов, которые все созданы на основе одного базового типа, отличаются друг от друга, и, таким образом, непосредственно сравнивать их между собой нельзя. Например, отдельные типы можно использовать, чтобы различать валюты разных стран. Проанализируйте следующее определение типа:

```
CREATE DISTINCT TYPE Usdollar AS DECIMAL (9.2);
```

В результате на основе заранее определенного типа DECIMAL создан новый тип данных предназначенный для долларов США. Аналогичным образом можно создать другой отдельный тип для евро:

```
CREATE DISTINCT TYPE Euro AS DECIMAL (9.2);
```

Теперь можно создать таблицы USInvoice (счет-фактура в долларах США) и EuroInvoice (счет-фактура в евро), в которых используются эти новые типы. В столбцах обеих таблиц хранятся такие реквизиты: идентификатор счета-фактуры, идентификатор клиента, идентификатор сотрудника, общая сумма продажи, сумма налога, сумма поставки и общий итог.

| CREATE TABLE USInvoice ( | | |
|---|---|---|
| InvID | INTEGER | PRIMARY KEY, |
| CustID | INTEGER, | |
| EmpID | INTEGER, | |
| TotalSale | Usdollar, | |
| Tax | Usdollar, | |
| Shipping | Usdollar, | |
| GrandTotal | Usdollar | |
| ); | | |

| CREATE TABLE EuroInvoice ( | | |
|---|---|---|
| InvID | INTEGER | PRIMARY KEY, |
| CustID | INTEGER, | |
| EmpID | INTEGER, | |
| TotalSale | Euro, | |
| Tax | Euro, | |
| Shipping | Euro, | |
| GrandTotal | Euro | |
| ); | | |

Оба типа, Usdollar и Euro, созданы на основе типа DECIMAL, но величины первого типа нельзя сравнивать с величинами второго, как и с величинами типа DECIMAL. Теперь в SQL, как и в обменном пункте, можно конвертировать доллары в евро, но для этого требуется специальная операция (CAST, означает "приведение"). Сравнения можно проводить, лишь выполнив конвертацию.

### Структурированные типы

Вторая форма определяемого пользователем типа – структурированный тип. Он представляет собой перечень определений атрибутов и методов, а не базируется на отдельном предопределенном исходном типе.

### Конструкторы

При создании структурного UDT-типа СУБД автоматически создает для него функцию-конструктор, давая ей имя, аналогичное имени UDT-типа.

### Мутаторы и наблюдатели

При создании структурированного UDT-типа СУБД также автоматически создает для него**функцию-мутатор** и **функцию-наблюдатель**. При применении функции-мутатора изменяется значение атрибута структурного типа. Функция-наблюдатель работает обратно функции-мутатора. Это позволяет отыскивать значение атрибута структурированного типа. Вы можете включить функции-наблюдатели в операторы SELECT, чтобы отыскать значения в базе данных.

### Подтипы и супертипы

Между двумя структурированными типами могут существовать иерархические отношения. К примеру, тип MusicCDudt имеет подтипы RockCDudt и ClassicalCDudt. Для этих двух подтипов MusicCDudt является супертипом. Если для MusicCDudt нет подтипа, который является супертипом для RockCDudt, то RockCDudt является собственным подтипом MusicCDudt. Если RockCDudt имеет подтип с именем HeavyMetalCDudt, который, в свою очередь, также является подтипом для MusicCDudt, то тип HeavyMetalCDudt уже не будет являться собственным подтипом для MusicCDudt.

Структурированный тип, не имеющий супертипа, называется максимальным супертипом, а структурированный тип, не имеющий подтипа, – конечным подтипом.

### Пример структурированного типа

Структурированные UDT-типы могут быть созданы следующим способом:

| | |
|---|---|
| /* Создаем UDT-тип MusicCDudt */ | |
| CREATE TYPE MusicCDudt AS | |
| /* Задаем атрибуты */ | |
| Title | CHAR(40), |
| Cost | DECIMAL(9.2), |
| SuggestedPrice | DECIMAL(9.2), |
| /* Разрешаем подтипы */ | |
| NOT FINAL; | |
| CREATE TYPE RockCDUdt UNDER MusicCDudt NOT FINAL; | |

Подтип RockCDudt наследует атрибуты своего супертипа MusicCDudt.

```
CREATE TYPE HeavyMetalCDudt UNDER RockCDUdt FINAL;
```

Создадим таблицы, использующие эти типы. Например:

| | |
|---|---|
| CREATE TABLE METALSKU ( | |
| Album | HeavyMetalCDudt, |
| SKU | INTEGER); |

Теперь можно добавить строки в новую таблицу:

```
BEGIN
/* Объявляем временную переменную 'a' */
DECLARE a = HeavyMetalCDudt;
/* Выполняем функцию-конструктор */
```

```
SET a = HeavyMetalCDudt ();
/* Выполняем первую функцию-мутатор */
SET a = a.title('Edward the Great');
/* Выполняем вторую функцию-мутатор */
SET a = a.cost(7.50);
/* Выполняем третью функцию-мутатор */
SET a = a.suggestedprice(15/99);
INSERT INTO METALSKU VALUES (a, 31415926);
END
```

### Сводка типов данных

В таблице 2.2 перечислены различные типы данных и показаны литералы, которые соответствуют каждому из этих типов.

**Таблица 2.2**. Типы данных.

| Тип данных | Пример значения |
|---|---|
| CHARACTER (20) | 'Любительское радио' |
| VARCHAR (20) | 'Любительское радио' |
| CLOB (1000000) | 'В этой строке миллион символов..' |
| SMALLINT, BIGINT ИЛИ INTEGER | 7500 |
| NUMERIC или DECIMAL | 3425.432 |
| REAL, FLOAT ИЛИ DOUBLE PRECISION | 6.626e-34 |
| BLOB (1000000) | '1001001110101011010101010101… ' |
| BOOLEAN | 'true' |
| DATE | DATE '1957-08-14' |
| TIME (2) WITHOUT TIME ZONE (аргумент указывает количество цифр в дробной части) | TIME '12:46:02.43' WITHOUT TIME ZONE |
| TIME (3) WITH TIME ZONE | TIME '12:46:02.432-08:00' WITH TIME ZONE |
| TIMESTAMP WITHOUT TIME ZONE (0) | TIMESTAMP '1957-08-14 12:46:02' WITHOUN TIME ZONE |
| TIMESTAMP WITH TIME ZONE (0) | TIMESTAMP '1957-08-14 12:46:02-08:00' WITH TIME ZONE |
| INTERVAL DAY | INTERVAL '4' DAY |
| ROW | ROW (Street VARCHAR (25), City VARCHAR (20), Stat CHAR (2), PostalCode VARCHAR (9)) |
| ARRAY | INTEGER ARRAY [15] |
| MULTISET | Нет точного применения типа multiset |
| REF | Это не тип, а указатель |
| USER DEFINED TYPE | Тип валюты, созданный на основе decimal |

***Помни:***

*Имеющаяся у вас реализация SQL может не поддерживать все типы данных, перечисленные в этом разделе. Более того, в вашей реализации могут поддерживаться нестандартные типы, которые здесь не описываются. (Каждый из вас успел уже как профессионал набраться опыта, "намотать километраж" и т.д. Так что вы знаете, где искать то, чего еще не знаете.)*

### Неопределенные значения

***Помни:***

*Если в поле базы данных находятся какие-то данные, то в этом поле имеется определенное значение. А если поле не содержит никаких данных, то говорят, что у него неопределенное значение. Неопределенное значение (null) в числовом поле – это не одно и то же, что нуль. А в символьном поле неопределенное значение – это не одно и то же,*

*что пустая строка. И нуль и пустая строка являются определенными значениями. Неопределенное же значение указывает на то, что имеющееся в поле значение неизвестно*.

Встречаются случаи, когда поле может иметь неопределенное значение. В следующем списке приведены некоторые из этих случаев и даны примеры каждого из них.

- **Значение существует, но вам оно пока что неизвестно**. До того, как была точно вычислена масса кварка, вы в самой верхней строке таблицы QUARM (кварк) установили в поле MASS (масса) неопределенное значение.
- **Значение пока что не существует**. В строке SQL For Dummies, 5th Edition таблицы BOOKS (книги) вы установили в поле TOTAL_SOLD (всего продано) неопределенное значение, так как первые данные о продажах за квартал еще не поступили.
- **Поле для данной строки неприменимо**. В строке С-3РО таблицы EMPLOYEE (наемный работник) вы установили в поле SEX (пол) неопределенное значение, так как С-3РО – это андроид, у которого пола нет.
- **Значение выходит за пределы установленного диапазона**. В строке Oprah Winfrey (Опра Уинфри) таблицы EMPLOYEE вы установили в поле SALARY (зарплата) неопределенное значение, так как для этого поля вы задали тип NUMERIC (8.2), а оклад, предусмотренный в контракте Опра, превышает 999999.99 доллара.

***Совет:***

*Поле может иметь неопределенное значение по самым разным причинам. Так чад не торопитесь с выводами относительно того, что означает конкретное неопределенное значение*.

## Ограничения

Ограничения вы устанавливаете на данные, вводимые кем-либо в таблицу базы данных. Например, известно, что значения, вводимые в определенный числовой столбец, должны находиться в пределах определенного диапазона. А если кто-то вводит число, которое не попадает в этот диапазон, то такой ввод будет ошибочным. От таких ошибок и защищает установленное на столбец ограничение – вводить в него только значения из определенного диапазона.

Традиционно сложилось так, что если прикладная программа использует базу данных, то она и накладывает на эту базу любые ограничения. Однако в самых последних продуктах у вас есть возможность устанавливать ограничения на данные непосредственно из СУБД. Этот подход дает несколько преимуществ. Если одна и та же база данных используется множеством приложений, то вам придется устанавливать ограничения только один раз, а не столько, сколько имеется приложений. Кроме того, устанавливать ограничения на уровне базы данных обычно проще, чем на уровне приложения. Во многих случаях вам будет достаточно только добавить предложение в свой оператор **CREATE** (создать).

Об ограничениях и утверждениях (**assertions**), которые тоже являются ограничениями, но применяются к более чем одной таблице, подробно рассказывается в главе 5.

## Использование SQL в системе клиент/сервер

SQL – это подъязык данных, который работает в одно – или многопользовательской системе. Особенно хорошо SQL работает в системе клиент/сервер. В такой системе пользователи работают на множестве клиентских машин, соединенных с серверным компьютером. И эти пользователи могут иметь доступ – через локальную сеть или другие

каналы связи – к базе данных, расположенной на сервере, с которым соединены их машины. В прикладной программе, работающей на клиентском компьютере, создаются команды SQL. Та часть системы СУБД, которая находится на клиентском компьютере, передает эти команды на сервер по каналу связи, соединяющему сервер с клиентом. А та часть СУБД, которая находится на сервере, интерпретирует и выполняет полученную команду SQL, а затем по каналу связи отправляет результаты назад, клиенту. В виде SQL можно закодировать очень сложные операции, а затем на сервере декодировать их и выполнить. Такого рода система позволяет эффективнее всего использовать пропускную способность канала связи.

Если вы с помощью SQL получаете данные через систему клиент/сервер, то по каналу связи от сервера на клиентский компьютер попадут только нужные вам данные. И наоборот, простая система с разделением ресурсов и с минимальным "интеллектом" сервера должна гонять туда-сюда по каналу связи огромные блоки данных. И все это ради того, чтобы вы смогли получить крохотное количество нужной информации. Не приходится и говорить, что такого рода пересылки данных могут очень замедлить работу. Архитектура клиент/сервер, дополняя характеристики SQL, дает возможность в малых, средних и больших сетях получать хорошую производительность при умеренных расходах.

## Сервер

Сервер не делает ничего, пока не получит запрос от клиента. Он только стоит и ждет. Но если требуется одновременно обслужить множество клиентов, то серверам приходится реагировать довольно оперативно. Они обычно отличаются от клиентских машин тем, что имеют быстрые дисковые массивы. Серверы настроены так, чтобы обеспечивать скорейший доступ к данным и скорейшую их передачу. Но так как им приходится обрабатывать трафик, идущий одновременно со множества клиентских машин, то на сервере должен быть очень быстрый процессор.

### Что такое сервер

**Сервер** (полное название – **сервер базы данных**) является той частью системы клиент/сервер, где находится база данных. Кроме того, на нем находится серверная часть системы управления базой данных. Команды, которые приходят от клиентов, интерпретируются в ней, а затем переводятся в операции, выполняемые в базе данных. Серверные программы преобразуют в определенный формат результаты, полученные при выполнении каждого запроса, и отправляют эти результаты клиенту, от которого пришел запрос.

### Что делает сервер

Работа у сервера относительно простая и понятная. Все, что ему нужно делать, – это читать, интерпретировать и выполнять команды, приходящие по сети от клиентов. Эти команды Должны быть написаны на одном из имеющихся подъязыков данных. Подъязык нельзя считать полным языком – он выполняет только часть функций языка. Подъязык данных занят только обработкой данных. В нем имеются операции добавления, обновления, удаления и выборки данных, но нет таких управляющих структур, как циклы, локальные переменные, Функции или операции ввода-вывода на различные устройства. Из используемых сегодня подъязыков данных самым известным является SQL, и он уже стал промышленным стандартом. SQL вытеснил патентованные подъязыки данных на всех классах машин. С появлением стандарта SQL: 1999 язык SQL пополнился новыми возможностями, которые отсутствуют У традиционных подъязыков. Однако SQL:2003 все еще не является полноправным языком Программирования общего назначения. Поэтому, чтобы создать приложение, работающее с базой данных, его следует использовать вместе с языком-оболочкой.

## Клиент

Клиентская часть системы клиент/сервер (**клиент**) состоит из двух компонентов: аппаратного и программного. Аппаратным компонентом является клиентский компьютер и его интерфейс, предназначенный для соединения с локальной сетью. Эта аппаратура может быть очень похожа на ту, что используется на сервере, или даже идентична ей. Клиент отличается от сервера прежде всего программным обеспечением.

### Что такое клиент

Основная работа клиента состоит в том, чтобы поддерживать пользовательский интерфейс. С точки зрения пользователя клиентской машиной является компьютер, а приложением – пользовательский интерфейс. Пользователь даже может и не знать, что в процессе участвует сервер. Обычно сервер не виден – он может находиться в другой комнате. Но кроме пользовательского интерфейса в состав клиента входят еще и прикладная программа, а также клиентская часть СУБД. Прикладная программа выполняет нужную вам специальную задачу, такую как работа со счетами дебиторов или оформление заказов. Клиентская часть СУБД выполняет команды прикладной программы и обменивается с серверной частью данными и командами SQL.

### Что делает клиент

Клиентская часть СУБД выводит информацию на экран и реагирует на пользовательский ввод, переданный с помощью клавиатуры, мыши или другого устройства ввода. Кроме того, клиент также может обрабатывать данные, приходящие через линию связи или из других станций сети. Весь "интеллект" приложения сосредоточен в клиентской части СУБД. Эта клиентская часть как раз и интересует разработчиков. Ведь в серверной части всего лишь монотонно, механически обрабатываются запросы, пришедшие от клиента.

### Использование SQL в Intenet/интернет

Работа с базами данных через Internet или интранет кардинально отличается от работы в традиционной системе клиент/сервер. Основное различие между ними находится в клиентской части. В традиционной системе клиент/сервер большинство функций СУБД выполняется на клиентской машине. А в базе данных, работающей через Internet, большая часть системы СУБД (если не вся) находится на сервере. В клиентской же части может не быть ничего, кроме Web-браузера. В большинстве случаев в ней находится браузер вместе со своим расширением, таким как включаемый модуль Netscape или элемент управления ActiveX. Таким образом, "интеллект" системы концентрируется на сервере. Такая концентрация имеет несколько преимуществ, перечисленных ниже.

- Достигается дешевизна клиентской части системы (браузера).
- Используется стандартный пользовательский интерфейс.
- Клиентскую часть легко поддерживать.
- Используется стандартная связь клиент/сервер.
- Имеется общее средство вывода мультимедийных данных.
- Чтобы защищать информацию от несанкционированного доступа или повреждения, необходимо на Web-сервере и в клиентском браузере поддерживать надежную систему шифрования.
- В браузерах не выполняется в достаточной степени проверка вводимых данных на правильность.
- Между находящимися на разных серверах таблицами базы данных может нарушиться синхронизация.

Впрочем, имеются клиентские и серверные расширения, предназначенные для решения этих проблем. Благодаря этим расширениям Internet вполне подходит для установки приложений, работающих с базами данных. Архитектура интранет похожа на ту, что применяется в Internet, но вопрос защиты в них не является предметом сильного беспокойства. Так как в организациях, где поддерживается интранет, физически контролируются все клиентские машины, а также серверы и сама сеть, соединяющая эти

компоненты, то система интранет не так открыта для происков хакеров. Впрочем, и в интранет ошибки, допускаемые при вводе данных, и нарушение синхронизации в базах данных остаются предметом беспокойства. В число главных неудобств работы с базами данных через Internet входят проблемы защиты и целостности данных.

## Компоненты SQL

В этой главе…

- Создание баз данных
- Обработка данных
- Защита баз данных

**SQL** – это язык, специально разработанный, чтобы создавать и поддерживать данные в реляционных базах. И хотя компании, поставляющие системы для управления такими базами, предлагают свои реализации SQL, развитие самого языка определяется и контролируется стандартом ISO/ANSI. Этот стандарт пересматривался последний раз в 2003 году. Все реализации в большей или меньшей степени отличаются от стандарта. Но как можно более полное следование стандарту– главное условие для работы базы данных и связанных с ней приложений на более чем одной платформе.

SQL не является программным языком общего назначения, но некоторые достаточно мощные средства у него все же имеются. Все необходимые действия по созданию,

изменению, поддержке базы данных и обеспечению ее безопасности выполняются с помощью входящих в состав SQL трех языков.

- Язык определения данных (**Data Definition Language**, DDL). Это та часть SQL, которая используется для создания (полного определения) базы данных, изменения ее структуры и удаления базы после того, как она становится ненужной.
- Язык манипулирования данными (**Data Manipulation Language**, DML). Предназначен для поддержки базы данных. С помощью этого мощного инструмента можно точно указать, что именно нужно сделать с данными, находящимися в базе, – ввести, изменить или выбрать нужные.
- Язык управления данными (**Data Control Language**, DCL). Защита базы данных от различных вариантов повреждения. При правильном использовании DCL обеспечивает защиту базы, а степень защищенности зависит от используемой реализации. Если реализация не обеспечивает достаточной защиты, то довести защиту до нужного уровня необходимо при разработке прикладной программы.

В этой главе вы познакомитесь с DDL, DML и DCL.

## Язык определения данных

Язык определения данных (DDL) – это часть языка SQL, которая используется для создания, изменения и уничтожения основных элементов реляционной базы данных. В число этих элементов могут входить таблицы, представления, схемы, каталоги, кластеры и, возможно, не только они. В этом разделе говорится о контейнерной иерархии, которая связывает между собой эти элементы, и рассматриваются команды, выполняемые с элементами базы данных.

В главе 1 упоминались таблицы и схемы, и говорилось, что схема – это общая структура, в состав которой входят таблицы. Таким образом, таблицы и схемы являются двумя элементами **контейнерной иерархии** реляционной базы данных. Эту иерархию можно представить таким образом.

- Таблицы состоят из столбцов и строк.
- Схемы состоят из таблиц и представлений.
- Схемы находятся в каталогах.

Сама же база данных состоит из каталогов. Кое-где можно встретить и такое название базы данных, как **кластер**.

## Создание таблиц

Таблица базы данных представляет собой двумерный массив, состоящий из строк и столбцов. Создать таблицу можно с помощью команды языка **SQL CREATE TABLE** (создать таблицу). В команде следует указать имя и тип данных каждого столбца.

После того как таблица создана, можно приступать к ее заполнению данными. (Впрочем, загружать данные – это дело языка DML, а не DDL.) Если требования меняются, то изменить структуру уже созданной таблицы можно с помощью команды ALTER TABLE (изменить таблицу). Со временем таблица может перестать быть полезной или устареть. И если час таблицы пробил, то уничтожить ее можно с помощью команды DROP (прекратить). Имеющиеся в SQL разные формы команд – CREATE (создать) и ALTER (изменить), а также DROP – как раз и представляют собой язык DDL.

Скажем, вы проектируете базы данных и не хотите, чтобы их таблицы постепенно, по мере обновления имеющихся в них данных, стали бы "неудобоваримыми". Чтобы обеспечить поддержку целостности данных, вы принимаете решение: структура таблиц этой

базы должна быть наилучшим образом нормализована. **Нормализация**, которая сама по себе является широким полем для исследования, – это способ задания такой структуры таблиц баз данных, в которой обновление данных не создавало бы аномалий. В каждой создаваемой таблице столбцы соответствуют атрибутам, которые тесно связаны друг с другом.

Вы, например, можете создать таблицу CUSTOMER (клиент), в которой имеются такие атрибуты: CUSTOMER.CustomerID (идентификатор клиента), CUSTOMER.FirstName (имя), CUSTOMER.LastName (фамилия), CUSTOMER.Street (улица), CUSTOMER.Sity (город), CUSTOMER.State (штат), CUSTOMER.Zipcode (почтовый код) и CUSTOMER.Phone (телефон). Все эти атрибуты имеют отношение к описанию клиента, а не любого другого объекта. В них находится более-менее постоянная информация о клиентах вашей организации.

В большинстве систем управления базами данных таблицы этих баз можно создавать с помощью графических инструментов. Однако такие таблицы можно создавать и с помощью команды языка SQL. Вот, например, команда, при выполнении которой создается таблица CUSTOMER:

```
CREATE TABLE CUSTOMER (
CustomerID   INTEGER        NOT NULL,
FirstName    CHARACTER (15),
LastName     CHARACTER (20) NOT NULL,
Street       CHARACTER (25),
City         CHARACTER (20),
State        CHARACTER (2),
Zipcode      INTEGER
Phone        CHARACTER (13));
```

Для каждого столбца указывается его имя (например, CustomerID), тип данных (например, INTEGER) и, возможно, одно или несколько ограничений, например NOT NULL (не может быть неопределенным значением).

На рис. 3.1 показана часть таблицы CUSTOMER с теми данными, которые могут быть введены в нее.

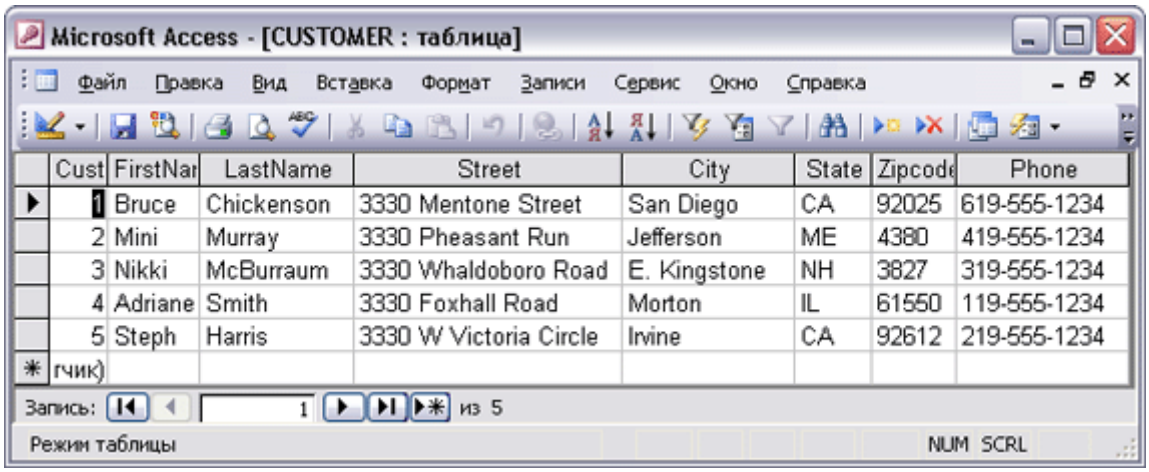


**Рис. 3.1**. Таблица CUSTOMER, которую можно создать с помощью команды Create Table

***Помни:***

*Если используемая вами реализация SQL не полностью соответствует SQL.2003, то синтаксис, которым вам придется пользоваться, может и не совпадать с приведенным в этой книге. За специальной информацией обратитесь к документации к СУБД.*

Представьте, что перед вами стоит задача создать базу данных для своей организации. Вдохновленные перспективой создать полезную, прекрасную и совершенно безупречную структуру, представляющую огромную важность для будущего вашей компании, вы садитесь за компьютер и начинаете вводить команды CREATE из языка SQL. Так?

Не совсем. На самом деле это – неверный подход. Когда вдохновение и энтузиазм не дожидались, пока будет проведено тщательное планирование, многие проекты по созданию баз данных с самого начала получались ущербными. Пусть вы уверены, что имеете в уме четкое представление о том, какая структура должна быть у базы данных. Однако даже в этом случае запишите все на бумаге, а лишь после этого прикасайтесь к клавиатуре. Ниже приведены процедуры, о которых не стоит забывать при планировании базы данных.

- Определите все таблицы, которые вам нужны.
- Определите, какие столбцы должны быть в каждой таблице.
- Присвойте каждой таблице **первичный ключ**, в уникальности которого вы уверены. (О первичных ключах говорится в главах 4 и 5.
- В каждой таблице должен быть как минимум один столбец, общий с какой-либо другой таблицей базы данных. Такие общие столбцы служат для логического соединения, позволяющего информации в одной таблице ссылаться на соответствующую информацию в другой.
- Приведите каждую таблицу по меньшей мере в третью нормальную форму (ЗНФ), гарантирующую от аномалий ввода, удаления или обновления. (О нормализации баз данных рассказывается в главе 5.

Только создав проект на бумаге и проверив, насколько хорошо он смотрится, вы будете готовы перенести его в компьютер, используя команду CREATE языка SQL.

## Место для представления

Иногда из таблицы CUSTOMER (клиент) вам требуется получить определенную информацию. При этом не нужно просматривать все подряд, а только конкретные столбцы и строки. В таком случае требуется представление (view).

**Представления** – это виртуальные таблицы. В большинстве реализаций они не являются в таблицах метаданных, и данные на самом деле поступают из таблиц, на основе которых это представление создано. Его данные больше нигде не хранятся. Одни представления состоят из определенных столбцов и строк одной таблицы. Другие же, которые называются **многотабличными представлениями**, получаются не менее чем из двух таблиц.

## Однотабличное представление

Иногда данные, которые дадут ответ на ваш вопрос, находятся в единственной таблице базы данных. А если вся необходимая вам информация находится в одной таблице, то можно создать однотабличное представление данных. Скажем, например, что нужно

просмотреть имена (first name), фамилии (last name) и телефонные номера (phone) всех клиентов, которые живут в штате Нью-Хэмпшир (который обозначается аббревиатурой NH). Тогда на основе таблицы CUSTOMER можно создать представление, содержащее только те данные, которые вам нужны. Оно создается при выполнении следующей команды:

```
CREATE VIEW NH_CUST AS
SELECT CUSTOMER.FirstName,
CUSTOMER.LastName,
CUSTOMER.Phone
FROM CUSTOMER
WHERE CUSTOMER.State = 'NH';
```

Диаграмма на рис. 3.2 показывает, каким образом представление создается из таблицы CUSTOMER.

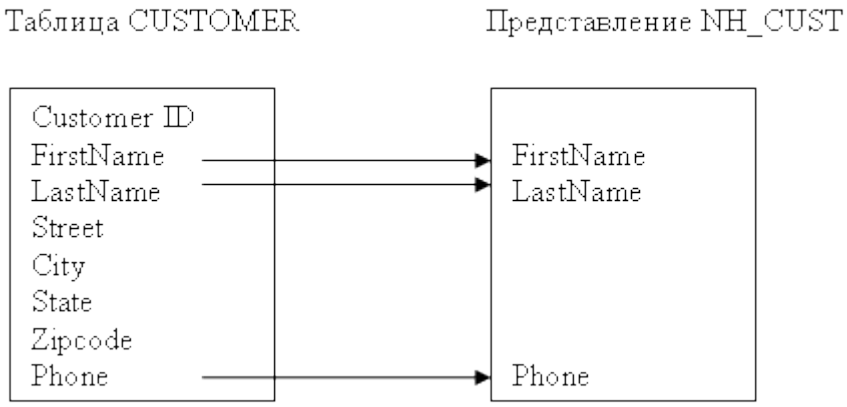


**Рис. 3.2**. Создание представления NH_CUST из таблицы CUSTOMER

***Совет:***

*Этот код безупречно правильный, но немного громоздкий. Ту же самую операцию можно выполнить, набирая команды и покороче. Это возможно тогда, когда имеющаяся у вас реализация SQL допускает, что если в перечисленных атрибутах не указаны ссылки на таблицу, то все атрибуты относятся к таблице предложения FROM. Если ваша система в состоянии сделать это разумное допущение, то команду можно сократить до следующих строк:*

```
CREATE VIEW NH_CUST AS
SELECT FirstName, LastName, Phone
FROM CUSTOMER
WHERE STATE = 'NH';
```

Хотя этот вариант записи проще, подобное представление может неправильно работать после применения команд ALTER TABLE. Конечно, если оператор JOIN (соединить) не используется, такого не случится. А для представлений с операторами JOIN лучше использовать полные имена. Об операторах JOIN рассказывается в главе 10.

### Создание многотабличного представления

Чтобы получать ответы на имеющиеся вопросы, часто приходится выбирать данные не менее чем из двух таблиц. Скажем, вы работаете в магазине спорттоваров, и для рассылки рекламы по почте вам нужен список клиентов, купивших у вас в прошлом году лыжное снаряжение. Скорее всего, потребуется информация из следующих таблиц: CUSTOMER (клиент), PRODUCT (товар), INVOICE (счет-фактура) и INVOICE_LINE (строка счета-фактуры). На их основе можно создать многотабличное представление, которое покажет нужные данные. Создав представление, его можно использовать снова и снова. При каждом таком использовании представление отображает последние изменения в таблицах, на основе которых это представление создано.

В базе данных магазина спорттоваров имеются четыре таблицы: CUSTOMER, PRODUCT, INVOICE и INVOICE_LINE. Структура каждой из них показана в табл. 3.1.

**Таблица 3.1**. Таблицы базы данных магазина спорттоваров.

| Таблица | Столбец | Тип данных | Ограничение |
|---|---|---|---|
| **CUSTOMER** | **Customer iD** (идентификационный номер клиента) | INTEGER | NOT NULL (не может быть неопределенным значением) |
| | **FirstName** (имя) | CHARACTER (15) | |
| | **LastName** (фамилия) | CHARACTER (20) | NOT NULL |
| | **Street** (улица) | CHARACTER (25) | |
| | **City** (ГОРОД) | CHARACTER (20) | |
| | **State** (штат) | CHARACTER (2) | |
| | **Zipcode** (ПОЧТОВЫЙ КОД) | INTEGER | |
| | **Phone** (телефон) | CHARACTER (13) | |
| **PRODUCT** | **Product id** (идентификационный номер товара) | INTEGER | NOT NULL |
| | **Name** (название) | CHARACTER (25) | |
| | **Description** (описание) | CHARACTER (30) | |
| | **Category** (категория) | CHARACTER (15) | |
| | **Vendor id** (идентификационный номер поставщика) | INTEGER | |
| | **VendorName** (наименование поставщика) | CHARACTER (30) | |
| **INVOICE** | **InvoiceNumber** (номер счета-фактуры) | INTEGER | NOT NULL |
| | **CustomeriD** (идентификационный номер покупателя) | INTEGER | |
| | **InvoiceDate** (дата выписки счета-фактуры) | DATE | |
| | **Totalsale** (всего продано на сумму) | NUMERIC (9.2) | |
| | **TotalRemitted** (всего оплачено) | NUMERIC (9.2) | |
| | **Formof Payment** (форма платежа) | CHARACTER (10) | |
| **INVOICE_LINE** | **LineNumber** (номер строки) | INTEGER | NOT NULL |
| | **InvoiceNumber** (номер счета-фактуры) | INTEGER | |
| | **Product id** (идентификационный номер товара) | INTEGER | |
| | **Quantity** (количество) | INTEGER | |
| | **SalePrice** (продано по цене) | NUMERIC (9.2) | |

Обратите внимание, что в некоторых столбцах табл. 3.1 имеется ограничение NOT NULL (не может быть неопределенным значением). Эти столбцы являются или первичными ключами

соответствующих таблиц, или вы решили, что есть другие причины, по которым их значения обязательно должны быть определенными. Первичный ключ таблицы должен однозначно идентифицировать каждую ее строку. Значение этого ключа в каждой строке должно быть определенным. (Подробно о ключах говорится в главе 5.)

Таблицы связываются друг с другом посредством общих столбцов. Ниже описаны связи между таблицами. (Отношения таблиц представлены на рис. 3.3.)

- Таблицу CUSTOMER связывает с другой таблицей, INVOICE, отношение "один ко многим". Один клиент может сделать множество покупок, в результате чего получится множество счетов-фактур. Однако каждый счет-фактура имеет отношение к одному и только одному клиенту.
- Таблицу INVOICE связывает с таблицей INVOICE_LINE также отношение "один ко многим". Ведь в счете-фактуре может быть множество строк, но каждая строка находится в одном и только одном счете-фактуре.
- Таблицу PRODUCT с таблицей INVOICE_LINE связывает отношение "один ко многим". Каждый товар может быть во множестве строк в одном или многих счетах-фактурах. Однако каждая строка относится к одному и только одному товару.

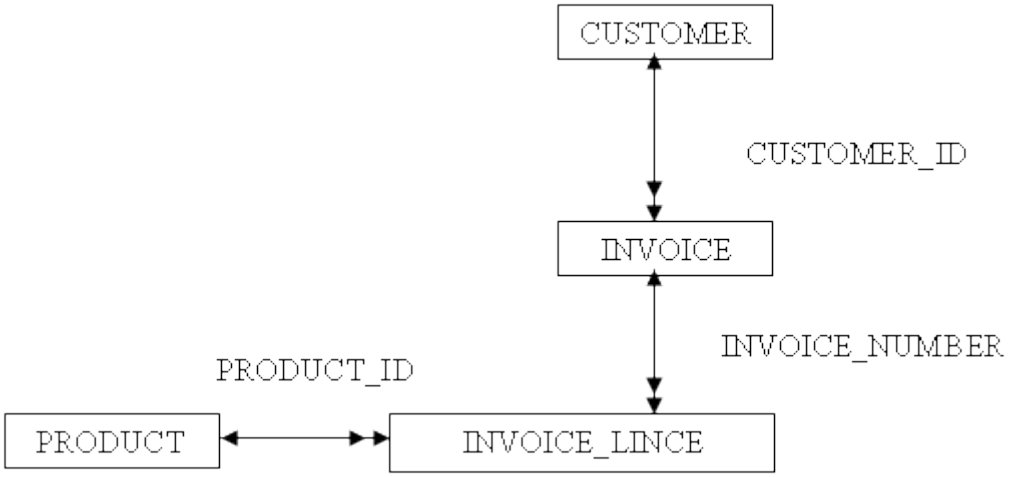


**Рис. 3.3**. Структура базы данных магазина спорттоваров

Таблица CUSTOMER поддерживает отношение с таблицей INVOICE, используя их общий столбец CustomerlD. А отношение таблицы INVOICE с таблицей INVOICE_LINE поддерживается с помощью общего столбца InvoiceNumber. Отношение же таблицы PRODUCT с таблицей INVOICE_LINE поддерживается с помощью общего столбца ProductDD. В сущности эти отношения и делают саму базу реляционной, т.е. работающей на основе отношений.

Чтобы получить информацию о тех клиентах, которые купили лыжное оборудование, необходимы данные из следующих полей: FirstName, LastName, Street, City, State и Zipcode из таблицы CUSTOMER; Category – из таблицы PRODUCT; InvoiceNumber – из таблицы INVOICE, а также LineNumber– из таблицы INVOICE_LINE. Нужное представление можно создавать поэтапно, используя для этого следующие команды:

```
CREATE VIEW SKI_CUST1 AS
SELECT FirstName,
LastName, Street,
City,
State,
Zipcode,
InvoiceNumber
FROM CUSTOMER JOIN INVOICE
USING (CUSTOMER_ID);
CREATE VIEW SKI_CUST2 AS
SELECT FirstName,
LastName,
Street,
```

```
City,
State,
Zipcode,
ProductID
FROM SKI_CUST1 JOIN INVOICE_LINE
USING (InvoiceNumber);
CREATE VIEW SKI_CUST3 AS
SELECT FirstName,
LastName,
Street,
City,
State,
Zipcode,
Category
FROM SKI_CUST2 JOIN PRODUCT
USING (ProductID);
CREATE VIEW SKI_CUST AS
SELECT DISTINCT FirstName,
LastName,
Street,
City,
State,
Zipcode
FROM SKI_CUST3
WHERE CATEGORY = 'Ski';
```

Эти операторы CREATE VIEW соединяют данные из множества таблиц, используя для этого оператор JOIN. Диаграмма всего этого процесса показана на рис. 3.4.

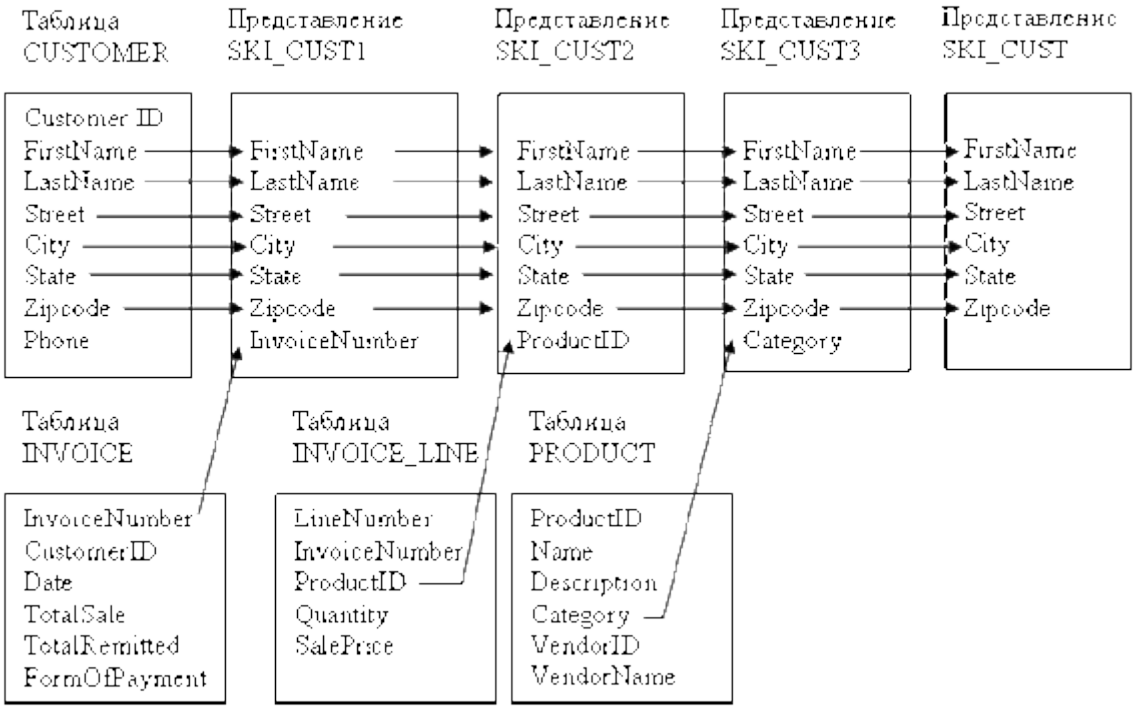


**Рис. 3.4**. Создание многотабличного представления с помощью оператора JOIN

Ниже приведены положения для четырех операторов CREATE VIEW.

- Первый оператор соединяет столбцы из таблицы CUSTOMER со столбцом из таблицы INVOICE и создает представление SKI_CUST1.
- Второй оператор соединяет представление SKLCUST1 со столбцом из таблицы INVOICE_LINE, создавая таким образом представление SKI_CUST2.
- Третий оператор соединяет представление SKLCUST2 со столбцом из таблицы PRODUCT и создает представление SKI_CUST3.

- Четвертый оператор отбрасывает все строки, где в поле категории товара отмечено не 'Ski' (лыжи). В результате получается представление SKI_CUST, в котором находятся имена, фамилии и адреса тех клиентов, которые хотя бы один раз купили товары категории 'Ski'. Каждому из этих клиентов, даже если он покупал лыжи много раз, в представлении SKI_CUST будет соответствовать только одна запись. Это достигается благодаря ключевому слову DISTINCT (отдельный), которое находится в SELECT четвертого оператора CREATE VIEW. (Об операторах JOIN подробно говорится в главе 10.)

### Сборка таблиц в схемы

Таблица состоит из строк и столбцов и обычно соответствует какому-либо объекту, такому, например, как множество клиентов, товаров и счетов-фактур. Для полезной работы обычно требуется информация о нескольких (или многих) объектах, имеющих между собой какие-либо отношения. Таблицы, соответствующие этим объектам, вы располагаете вместе, согласно логической схеме. (**Логическая схема** – это организационная структура совокупности таблиц, связанных между собой отношениями.)

В системе, где может сосуществовать несколько несвязанных друг с другом проектов, можно соединить все таблицы, связанные друг с другом отношениями, в одну схему. А из таблиц, не вошедших в эту схему, можно образовать другие схемы. Чтобы таблицы из одного проекта не оказались случайно в другом, схемам следует дать имена. У каждого проекта имеется своя собственная схема, которую по имени можно будет отличать от других схем. Некоторые табличные имена (например, CUSTOMER, PRODUCT и т.д.) могут встречаться сразу в нескольких проектах. Если есть хоть малейший шанс, что возникнет путаница с именами, необходимо в именах таблиц указывать имя схемы (примерно так: ИМЯ_СХЕМЫ.ИМЯ_ТЛБЛИЦЫ). Если имя схемы не указано, SQL будет считать, что эта таблица относится к схеме, подразумеваемой по умолчанию.

***Помни:***

*У базы данных, кроме логической, есть еще и физическая схема. Физическая схема – это способ, с помощью которого данные и соответствующие им компоненты, например индексы, физически размещаются на диске компьютера. И когда в книге говорится о схеме базы данных, то имеется в виду логическая схема, а не физическая.*

### Заказ по каталогу

Для по-настоящему больших баз данных даже множества схем может оказаться недостаточно. В больших распределенных средах таких баз дублирование встречается даже в именах схем. Чтобы этого не было, в SQL предусмотрен еще один уровень контейнерной иерархии – каталог. **Каталог** – это набор схем, имеющий свое специальное имя.

При указании имени таблицы можно также использовать имена ее каталога и схемы. Таким образом, гарантируется, что никто не перепутает две таблицы с одним и тем же именем, находящиеся в схемах с одинаковым именем. Имя таблицы с указанием каталога имеет следующий формат: ИМЯ_КАТАЛОГА.ИМЯ_СХЕМЫ.ИМЯ_ТАБЛИЦЫ.

Наивысшим уровнем контейнерной иерархии базы данных являются кластеры. Впрочем, редко в какой системе надо строить полную контейнерную иерархию. В большинстве случаев можно вполне ограничиться каталогами. В каталогах находятся схемы, в схемах – таблицы и представления, а в таблицах и представлениях – столбцы и строки.

В каталоге также находится **информационная схема**. В этой схеме находятся системные таблицы, а в них хранятся метаданные, относящиеся к другим схемам. В главе 1 база данных была определена как самоописательное собрание интегрированных записей. Метаданные в системных таблицах как раз и делают базу данных самоописательной.

Каталоги можно различать по именам. Поэтому в базе данных можно иметь множество каталогов. В каждом каталоге, в свою очередь, может быть множество схем, а в каждой

схеме – таблиц. И, конечно же, в каждой таблице может быть множество столбцов и строк. Взаимоотношения внутри иерархии базы данных показаны на рис. 3.5.

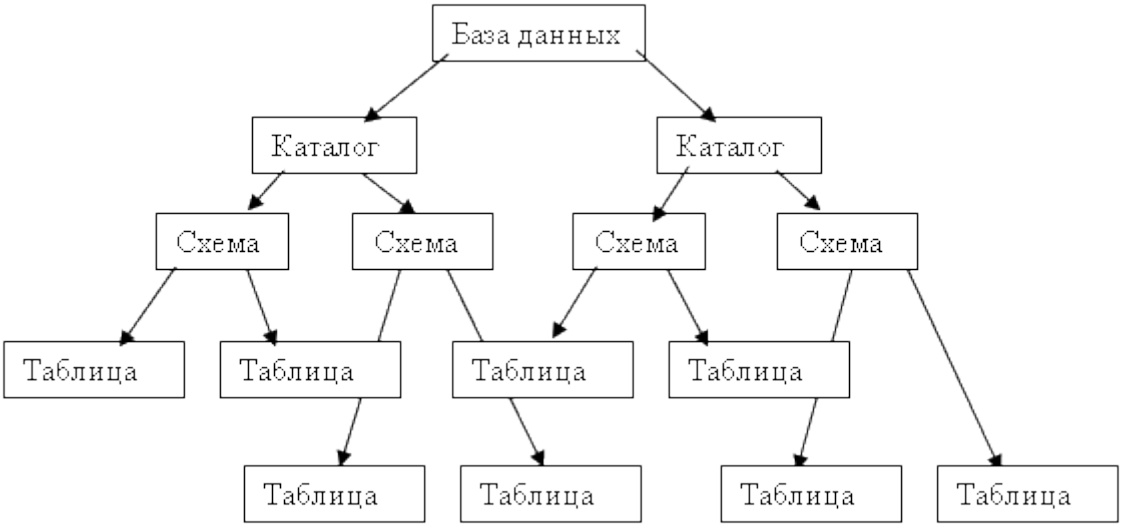


**Рис. 3.5**. Иерархическая структура типичной базы данных SQL

### Знакомство с командами DDL

Язык определения данных (DDL) работает со структурой базы данных, в то время как язык манипулирования (он будет описан позже) – с данными, которые находятся в этой структуре. DDL состоит из следующих трех команд.

- Для создания основных структур базы данных используются разные формы команды**CREATE** (создать).
- Для изменения созданных структур применяется команда **ALTER** (изменить).
- Команда **DROP** (прекратить) применяется к таблице, чтобы не только удалить табличные данные, но и разрушить саму структуру этой таблицы.

В следующих разделах кратко описываются команды DDL. В главах 4 и 5 эти команды используются в примерах.

### CREATE

Команда языка SQL CREATE может создавать объекты SQL нескольких видов, в том числе схемы, домены, таблицы и представления. С помощью оператора CREATE SCHEMA (создать схему) можно создать схему, идентифицировать ее владельца и указать символьный набор по умолчанию. Вот, например, как может выглядеть такой оператор:

```
CREATE SCHEMA SALES
AUTHORIZATION SALES_MGR
DEFAULT CHARACTER SET ASCII_FULL;
```

С помощью оператора CREATE DOMAIN (создать домен) устанавливаются ограничения на те значения, которые могут быть в столбце, или указывается порядок сопоставления. Устанавливаемые на домен ограничения определяют, какие объекты могут и какие не могут в нем находиться. Создавать домены можно после того, как установлена схема. Следующий пример демонстрирует, как можно использовать эту команду:

```
CREATE DOMAIN AGE AS INTEGER
CHECK (AGE > 20);
```

Таблицы создаются с помощью оператора CREATE TABLE (создать таблицу), а представления – с помощью CREATE VIEW (создать представление). В этой главе уже приводились примеры использования операторов CREATE TABLE и CREATE VIEW. Когда с помощью оператора CREATE TABLE создается новая таблица, то в том же операторе на ее

столбцы можно также установить ограничения. Впрочем, иногда требуется установить ограничения, которые относятся не только к таблице, но и ко всей схеме. В таких случаях используется оператор CREATE ASSERTION (создать утверждение).

Кроме того, имеются операторы CREATE CHARACTER SET (создать символьный набор), CREATE COLLATION (создать сопоставление) и CREATE TRANSLATION (создать трансляцию), которые предоставляют широкие возможности по созданию новых символьных наборов, последовательностей сопоставления или таблиц трансляции. (**Последовательности сопоставления** определяют порядок, в котором будут проводиться операции сравнения или сортировки. **Таблицы трансляции** управляют преобразованием символьных строк из одного символьного набора в другой.)

### ALTER

Таблица не обязательно навсегда останется такой, какой ее создали. Как только ее начинают использовать, то вдруг обнаруживается, что в ней нет чего-то такого, что обязательно должно было быть. Чтобы изменить таблицу, добавив, изменив или удалив ее столбец, воспользуйтесь командой ALTER TABLE (изменить таблицу). Команду ALTER можно применять не только к таблицам, но также к столбцам и доменам.

### DROP

Удалить таблицу из схемы базы данных легко. Надо только использовать команду DROP TABLE <имя_таблицы> (прекратить поддержку таблицы). В результате стираются все данные этой таблицы, а также метаданные, которые определяют ее в словаре данных, – после чего таблицы как будто и не было.

## Язык манипулирования данными

Как уже говорилось в этой главе, DDL является частью языка SQL, предназначенной для создания, модификации или разрушения структур базы данных. Непосредственно с данными язык DDL не работает. Для этого предназначена другая часть SQL – **язык манипулирования данными** (**Data Manipulation Language**, DML). Некоторые операторы DML можно читать как обычные предложения на английском языке, и эти операторы легко понять. Однако другие операторы DML могут быть, наоборот, очень сложными – как раз из-за того, что SQL дает необъятные возможности работы с данными. Если в операторе DML имеется множество выражений, предложений, предикатов или подзапросов, то даже просто понять, для чего этот оператор предназначен, может оказаться по-настоящему трудным делом. Поработав с некоторыми из них, вы, возможно, захотите переключиться на что-нибудь более легкое, например, на хирургию мозга или квантовую электродинамику. Впрочем, все не так плохо. Дело в том, что такие сложные операторы SQL можно мысленно разбивать на простые части и анализировать одну за другой.

Можно использовать такие операторы DML: **INSERT** (вставить), **UPDATE** (обновить),**DELETE** (удалить) и **SELECT** (выбрать). Они могут состоять из разных частей, в том числе из множества предложений. А в каждом предложении могут быть выражения со значениями, логические связки, предикаты, итоговые функции и подзапросы. Все они позволяют точнее отделять друг от друга записи базы данных и получать из своих данных больше информации. В главе 6 рассказывается о том, как работают команды DML, а более подробно о самих командах речь пойдет в главах 7-12.

### Выражения со значениями

Чтобы комбинировать два или несколько значений, можно использовать выражения со значениями. В соответствии с разными типами данных имеется девять видов таких выражений:

- числовые;
- строковые;

- даты-времени;
- интервальные;
- логические;
- определяемые пользователем;
- записи;
- коллекции.

Типы **логические, определяемые пользователем, записи** и **коллекции** появились в SQL вместе со стандартом SQL: 1999. В некоторых реализациях они вообще еще не поддерживаются. Прежде чем использовать один из этих типов, необходимо убедиться, что он входит в состав вашей реализации.

### Выражения с числовыми значениями

Чтобы комбинировать числовые значения, используйте операторы сложения (+), вычитания (-), умножения (*) и деления (/). В следующих строках приведено несколько примеров выражений с числовыми значениями:

```
12-7
15/3-4
6 * (8+2)
```

Значения из этих примеров являются числовыми литералами. Значениями могут быть также имена столбцов, параметры, базовые переменные или подзапросы – при условии, что их значения являются числовыми. Вот несколько примеров:

```
SUBTOTAL + TAX + SHIPPING
6 * MILES/HOURS
:months/12
```

Двоеточие в последнем примере говорит о том, что следующий за ним терм (months – месяцы) является или параметром, или базовой переменной.

### Выражения со строковыми значениями

В выражениях со строковыми значениями может находиться оператор конкатенации (||). С его помощью, как показано в табл. 3.2, две текстовые строки объединяются в одну.

**Таблица 3.2**. Примеры конкатенации строк.

| Выражение | Результат |
|---|---|
| 'военная' \| \| 'разведка' | 'военная разведка' |
| 'абра' \| \| 'кадабра' | 'абракадабра' |
| CITY\| \| ' ' \| \|STATE\| \| ' '\| \|ZIP | Общая строка с названиями города, штата и с почтовым кодом, которые отделены друг от друга пробелами |

***Внимание:***
*В некоторых реализациях SQL вместо || в качестве оператора конкатенации используют +.*

Есть реализации, в которых вместо конкатенации используются строковые операторы, но стандарт SQL:2OO3 эти операторы не поддерживает.

### Выражения со значениями даты-времени и интервальными значениями

Выражения со значениями даты-времени, оказывается, работают (кто бы мог подумать!) со значениями даты и времени. В таких выражениях могут появляться данные типа DATE, TIME, TIMESTAMP и INTERVAL. Результатом выполнения выражений со значениями даты-времени всегда является другое значение даты-времени. К значению этого типа можно прибавить (или отнять от него) какой-либо интервал, а также задать часовой пояс.

Вот пример выражения со значениями даты-времени (название DUE_DATE означает "срок возврата", а INTERVAL 7' DAY – "интервал в 7 дней"):

```
DUE_DATE + INTERVAL '7' DAY
```

Такое выражение можно использовать в библиотечной базе данных, чтобы определять, когда отправить напоминание "должникам". А вот новый пример, где, правда, указывается не дата, а время:

```
TIME '18:55:48' AT LOCAL
```

Ключевые слова AT LOCAL означают, что указано местное время.

Выражения со значениями интервалов работают с промежутками времени между значениями даты-времени. Имеются два вида интервалов: **год-месяц** и **день-время**. Их нельзя использовать в одном и том же выражении.

Приведем пример использования интервалов. Скажем, кто-то возвращает в библиотеку книгу после истечения крайнего срока. Тогда, используя выражение со значениями интервалов, такое, например, как приведено в следующем примере, можно вычислить, сколько прошло дней задержки после крайнего срока, и соответственно назначить пеню (названия DATE_RETURNED и DATE_DLIE означают соответственно "дата возврата" и "крайний срок"):

```
(DATE_RETURNED-DATE_DUE) DAY
```

Так как интервал может быть типа год-месяц либо день-время, то следует указать, какой из них использовать. В последнем примере с помощью ключевого слова DAY (день) выбран второй.

### Выражения с логическими значениями

Выражение с логическими, или булевыми, значениями проверяет, является ли значение предиката истинным. Примером такого выражения может быть:

```
(CLASS = SENIOR) IS TRUE
```

Если это условие выбора строк из таблицы со списком учеников-старшеклассников, то будут выбраны только записи, соответствующие ученикам выпускных классов. А чтобы выбрать записи учеников других классов, можно использовать следующее выражение:

```
NOT (CLASS = SENIOR) IS TRUE
```

То же самое условие можно выразить и по-другому:

```
(CLASS = SENIOR) IS FALSE
```

Чтобы получить все строки, имеющие в столбце CLASS неопределенное значение, используйте:

```
(CLASS = SENIOR) IS UNKNOWN
```

### Выражения со значениями, определяемыми пользователем

О типах, определяемых пользователями, см. в главе 2. Благодаря такой возможности пользователи могут определять собственные типы данных, а не довольствоваться теми, которые есть на "складе" SQL. Если есть выражение, имеющее элементы данных какого-либо типа, определяемого пользователем, то значением этого выражения должен быть элемент того же типа.

### Выражения со значениями типа записи

Выражение со значениями типа записи, как это ни удивительно, определяет значение типа записи. Значение типа записи может состоять из одного выражения с каким-либо значением либо из множества таких выражений. Например:

```
('Джозеф Тайкосинер', 'заслуженный профессор в отставке', 1918)
```

Это строка таблицы сотрудников факультета, содержащей поля имени, фамилии, статуса и года начала работы на факультете.

### Выражения со значениями типа коллекции

Значением выражения типа коллекции является массив.

### Выражения со значениями типа ссылки

Значением выражения типа ссылки является ссылка на некоторый другой компонент базы данных, например столбец таблицы.

### Предикаты

**Предикаты** – это используемые в SQL эквиваленты логических высказывании. Примером высказывания является следующее выражение:

```
"Ученик учится в выпускном классе".
```

В таблице, содержащей информацию об учениках, домен столбца CLASS (класс) может быть набором таких значений: SENIOR (выпускной), JUNIOR (предпоследний), SOPHOMORE (второй старший класс), FRESHMAN (первый старший класс) и NULL (неизвестен). Предикат CLASS = SENIOR можно использовать для отсева тех строк, для которых его значение ложно, оставляя, соответственно, только те строки, для которых значение этого предиката истинно. Иногда в какой – либо строке значение этого предиката не известно (т.е. представляет собой NULL). В таком случае строку можно отбросить или оставить в зависимости от конкретной ситуации.

CLASS = SENIOR – это пример **предиката сравнения**. В SQL имеется шесть операторов сравнения. В простом предикате сравнения используется только один из этих операторов. Предикаты сравнения и примеры их использования приведены в таблице 3.3.

**Таблица 3.3**. Операторы и предикаты сравнения.

| Оператор | Сравнение | Выражение |
|---|---|---|
| = | Равно | CLASS = SENIOR |
| <> | Не равно | CLASS <> SENIOR |
| < | Меньше | CLASS < SENIOR |
| > | Больше | CLASS > SENIOR |
| <= | Меньше или равно | CLASS <= SENIOR |
| >= | Больше или равно | CLASS >= SENIOR |

***Внимание:***

*В последнем примере только первые два выражения имеют смысл (CLASS = SENIOR и CLASS <> SENIOR). Это объясняется тем, что SOPHOMORE считается больше, чем SENIOR, потому что в последовательности сопоставления, установленной по умолчанию (т.е. когда сортировка выполняется по алфавиту), SO следует после SE. Однако такая интерпретация, по всей вероятности, – не то, что вам нужно*.

### Логические связки

Логические связки позволяют из простых предикатов строить сложные. Скажем вам нужно в базе данных по ученикам средней школы найти информацию о юных дарованиях. Два логических высказывания, которые относятся к этим ученикам, можно прочитать следующим образом:

```
"Ученик учится в выпускном классе".
"Ученику еще нет 14 лет".
```

Чтобы отделить нужные вам записи, можно с помощью логической связки AND (и) создать составной предикат, например, как этот:

```
CLASS = SENIOR AND AGE < 14
```

Если используется связка AND, то чтобы составной предикат был истинным, истинными должны быть оба входящих в него предиката. А если нужно, чтобы составной предикат был истинным тогда когда истинный какой-либо из входящих в него предикатов, то используйте логическую связку OR (или). Третьей логической связкой является NOT (отрицание). Строго говоря, эта связка не соединяет два предиката. Она применяется к единственному предикату и заменяет его логическое значение на противоположное. Возьмем, например, следующее выражение:

```
NOT (CLASS = SENIOR)
```

Это значение истинно только тогда, когда значение CLASS на самом деле не равно SENIOR.

### Итоговые функции

Иногда информация, которую вы хотите получить из таблицы, не связана с содержимым отдельных строк, но относится к данным таблицы, взятым в целом. Для таких ситуаций стандарт SQL: 2003 предусматривает пять итоговых функций: COUNT, MAX, MIN, SUM и AVG Каждая из этих функций выполняет действие по получению данных, относящихся к множеству строк, а не только к одной.

### COUNT

Функция COUNT возвращает число строк указанной таблицы. Чтобы в базе данных сред. ней школы, используемой в качестве примера, подсчитать число самых юных учеников выпускных классов, воспользуйтесь следующим оператором (название GRADE означает "класс"):

```
SELECT COUNT (*)
FROM STUDENT
WHERE Grade =12 AND AGE < 14;
```

### MAX

Функция MAX используется для определения максимального значения столбца. Скажем, требуется найти самого старшего ученика вашей школы. Естественно, таких переростков может быть несколько. Строку с его данными возвращает следующий оператор:

```
SELECT FirstName, LastName, Age
FROM STUDENT
WHERE Age = (SELECT MAX(Age) FROM STUDENT);
```

В результате появляются данные по всем старшим ученикам, т.е. если возраст самого старшего ученика равен 23 годам, этот оператор возвращает данные по всем ученикам с возрастом 23 года.

В этом запросе используется подзапрос. Этот подзапрос, SELECT MAX(Age) FROM STUDENT, находится внутри главного запроса.

### MIN

Функция MIN работает точно так же, как и MAX, за исключением того, что MIN ищет в указанном столбце не максимальное, а минимальное значение. Чтобы найти самых юных учеников школы, можно использовать следующий запрос:

```
SELECT FirstName, LastName, Age
FROM STUDENT
WHERE Age = (SELECT MIN(Age) FROM STUDENT);
```

В результате появляются данные по самым младшим ученикам вашей школы.

### SUM

Функция SUM складывает значения из указанного столбца. Столбец должен иметь один из числовых типов данных, а значение суммы не должно выходить за пределы диапазона, предусмотренного для этого типа. Таким образом, если столбец имеет тип данных SMALLINT, то полученная сумма не должна превышать верхний предел, имеющийся у этого типа данных. В таблице INVOICE (счет-фактура) из базы данных, о которой уже говорилось в этой главе, хранятся данные по всем продажам. Чтобы найти общую сумму в долларах для всех продаж, иные которых хранятся в базе, используйте функцию SUM следующим образом:

```
SELECT SUM(TotalSale) FROM INVOICE;
```

### AVG

Функция AVG возвращает среднее арифметическое всех значений указанного столбца. Функция SUM, AVG применяется только к столбцам с числовым типом данных. Чтобы найти среднее арифметическое значение продаж, учитывая все финансовые операции, хранящиеся в базе, используйте функцию AVG следующим образом:

```
SELECT AVG(TotalSale) FROM INVOICE;
```

Имейте в виду, что неопределенные значения значениями не считаются, так что если в каких-либо строках в столбце TotalSale (всего продано) находятся неопределенные значения, то при подсчете средней продажи эти строки игнорируются.

### Подзапросы

**Подзапросами** (см. выше раздел "Итоговые функции") являются запросы, находящиеся внутри какого-либо запроса. В любом месте оператора SQL, где можно использовать выражение, можно также использовать и подзапрос. Подзапросы являются мощным инструментом для связывания информации из одной таблицы с информацией из другой. Дело в том, что запрос к одной из таблиц можно встроить в другой запрос. С помощью вложенных подзапросов можно иметь доступ к информации более чем из двух таблиц. Если правильно пользоваться подзапросами, то из базы данных можно получить почти любую нужную информацию.

### Язык управления данными

В **языке управления данными (Data Control Language, DCL)** имеются четыре команды: COMMIT (завершить), ROLLBACK (откат), GRANT (предоставить) и REVOKE (отозвать). Все эти команды связаны с защитой базы от случайного или умышленного повреждения.

### Транзакции

Базы данных наиболее уязвимы именно тогда, когда в них вносят изменения. Изменения могут быть опасными даже для однопользовательских баз. Аппаратный или программный сбой, происшедший во время изменения, может застать базу данных в переходном состоянии – между состоянием в момент начала изменений и состоянием, которое было бы в момент завершения этих изменений.

С целью защиты базы данных язык SQL ограничивает операции, которые могут ее изменить, так что они выполняются только в пределах транзакций. Во время транзакции SQL записывает каждую операцию с данными в файл журнала. Если транзакцию, перед тем как она будет завершена оператором COMMIT, что-то прервет, то можно восстановить первоначальное состояние системы с помощью оператора ROLLBACK. Этот оператор обрабатывает журнал транзакций в обратном порядке, отменяя все действия, имевшие место во время транзакции. Выполнив откат базы данных до состояния, в котором она была перед началом транзакции, можно выяснить, что вызвало неполадки, а затем попробовать еще раз выполнить транзакцию.

Повреждения базы данных или некорректные результаты возможны в многопользовательской системе даже тогда, когда нет никаких аппаратных или программных отказов. Серьезные неприятности могут быть вызваны совместными действиями двух или нескольких пользователей но отношению к одной и той же таблице, т.е. имеющих к ней доступ в одно и то же время. VL решает эту проблему, допуская внесение изменений только в пределах транзакций.

***Совет:***
*База данных может пострадать из-за сбоев в аппаратном или программном обеспечении. Современные СУБД стараются свести подобную возможность к минимуму. Для этого они все операции с базой данных выполняют в пределах транзакций. Выполнив операции, находящиеся в транзакции, СУБД завершают транзакции одним оператором COMMIT. В современных системах управления базами данных также ведутся журналы транзакций. Это делается для того, чтобы в случае неприятностей с аппаратным обеспечением, программами или персоналом гарантировать защиту данных. После завершения транзакции данные защищены от всех системных отказов, если только не считать самых катастрофических; в случае ее неудачи го проведения возможен откат транзакции назад к ее начальной фазе и – после устранения причин неполадок – повторное выполнение этой транзакции*.

Поместив все операции, воздействующие на базу данных, в транзакции, вы можете изолировать действия одного пользователя от действий другого. Если необходима уверенность, что результаты, получаемые из базы данных, являются точными, то такая изоляция очень важна.

***Технические подробности:***
*Возможно, вы удивитесь, что совместные действия двух пользователей могут при. вести к некорректным результатам. Скажем, Донна читает запись какой-либо таблицы из базы данных. Через мгновение Дэвид меняет в той записи значение числового поля. А теперь в то же поле Донна записывает число, полученное на основе значения, прочитанного ею вначале. И так как она не знает об изменении, сделанном Дэвидом, то ее операция является некорректной*.

Другая неприятность может произойти, когда Донна вносит в запись какие-то значения, а Дэвид затем эту запись читает. И если Донна проводит откат своей транзакции, то Дэвид не знает об этой операции и выполняет свои действия на основе прочитанного им значения, которое не совпадает со значением, имеющимся в базе после отката. То, что смешно в кинокомедии, не всегда приятно в реальной жизни.

## Пользователи и полномочия

Кроме повреждения данных, вызванного проблемами с оборудованием и программами или неумышленными совместными действиями двух пользователей, целостности данных угрожает и другая большая опасность. Это сами пользователи. Некоторым людям вообще нельзя иметь доступ к данным. Другим – только ограниченный доступ к некоторым данным и никакого доступа к остальным. А кое-кто должен иметь неограниченный доступ ко всем данным. Поэтому вам нужна система, предназначенная для классификации пользователей по категориям и присвоения этим пользователям в соответствии с их категорией определенных полномочий доступа.

Создатель схемы указывает, кого следует считать ее владельцем. Являясь владельцем схемы, вы можете предоставлять полномочия доступа пользователям. Любые полномочия, не предоставленные вами явно, являются недействительными. Вы также можете отозвать уже предоставленные вами полномочия. Пользователю, перед тем как получить

предоставляемый вами доступ к файлам, необходимо подтвердить свою личность, пройдя для этого процедуру аутентификации. Что собой представляет эта процедура – зависит от конкретной реализации SQL.

SQL дает возможность защищать следующие объекты базы данных.

- Таблицы.
- Столбцы.
- Представления.
- Домены.
- Символьные наборы.
- Сопоставления.
- Трансляции.

О символьных наборах, сопоставлениях и трансляциях рассказывается в главе 5. Стандарт SQL:2003 поддерживает различные виды защиты: **защиту просмотра**,**добавления**, **модификации**, **удаления**, **применения ссылок** и **использования** баз данных, а также виды защиты связанные с выполнением внешних процедур.

Доступ разрешается с помощью оператора GRANT (разрешить), а аннулируется с помощью тора REVOKE (отозвать). Управляя использованием команды SELECT, DCL позволяет определить тех, кто может видеть объекты базы данных, такие, например, как таблица, столбец или представление. В случае команды INSERT DCL позволяет определить тех, кто может добавлять в таблицу новые строки. То, что команда UPDATE может применяться только авторизованными пользователями, дает возможность назначать пользователей, ответственных за изменение табличных строк, и аналогично в случае команды DELETE – тех, кто может такие строки удалять.

Если в одной таблице базы данных имеется столбец, который для этой таблицы является внешним ключом, а для другой таблицы из этой базы – первичным, то для первой таблицы, если она ссылается на вторую, можно установить ограничение. Дело в том, что когда одна таблица ссылается на другую, то владелец первой из них, вероятно, сможет получать информацию о содержимом второй. Владельцу же второй таблицы, возможно, захочется этот доступ пресечь. Такая возможность дается в виде оператора GRANT REFERENCES (предоставить доступ по ссылке). В следующем разделе рассказывается о проблеме, связанной с "предательской" ссылкой, и о том, как оператор GRANT REFERENCES решает эту проблему. Применяя оператор GRANT USAGE (предоставить использование), можно назначать пользователей, которым позволено использование или просмотр содержимого домена, набора символов, сопоставления или трансляции. (Об этом рассказывается в главе 13.)

Операторы SQL, с помощью которых предоставляют или отзывают полномочия, приведены в табл. 3.4.

**Таблица 3.4**. Виды защиты.

| Действие по защите | Оператор |
|---|---|
| Позволяет просматривать таблицу | GRANT SELECT |
| Не позволяет просматривать таблицу | REVOKE SELECT |
| Позволяет вставлять строки в таблицу | GRANT INSERT |
| Не позволяет вставлять строки в таблицу | REVOKE INSERT |
| Позволяет менять значения в строках таблицы | GRANT UPDATE |
| Не позволяет менять значения в строках таблицы | REVOKE UPDATE |

| Действие по защите | Оператор |
|---|---|
| Позволяет удалять строки из таблицы | GRANT DELETE |
| Не позволяет удалять строки из таблицы | REVOKE DELETE |
| Позволяет ссылаться на таблицу | GRANT REFERENCES |
| Не позволяет ссылаться на таблицу | REVOKE REFERENCES |
| Позволяет использовать домен, набор символов, сопоставление или трансляцию | GRANT USAGE ON DOMAIN, REVOKE USAGE ON CHARACTER SET, REVOKE USAGE ON COLLATION, REVOKE USAGE ON TRANSLATION |
| Не позволяет использовать домен, набор сопоставление или трансляцию | REVOKE USAGE ON DOMAIN, REVOKE USAGE ON CHARACTER SET, REVOKE USAGE ON COLLATION, REVOKE USAGE ON TRANSLATION |

Разным пользователям, в зависимости от их потребностей, можно предоставить доступ разного уровня. Несколько примеров такой возможности показывают следующие команды:

```
GRANT SELECT
ON CUSTOMER
TO SALES_MANAGER;
```

В этом случае один пользователь, менеджер по продажам, получает возможность просматривать таблицу CUSTOMER (клиент).

В следующем примере показана команда, благодаря которой каждый пользователь имеющий доступ к системе, получает возможность просматривать розничный прайс-лист:

```
GRANT SELECT
ON RETAIL_PRICE_LIST
TO PUBLIC;
```

Ниже приведен пример команды, которая дает возможность менеджеру по продажам видоизменять розничный прайс-лист. Менеджер по продажам может менять содержимое имеющихся строк, но добавлять или удалять строки он не может.

```
GRANT UPDATE
ON RETAIL_PRICE_LIST
TO SALES_MANAGER;
```

В следующем примере приведена команда, позволяющая менеджеру по продажам добавлять в розничный прайс-лист новые строки:

```
GRANT INSERT
ON RETAIL_PRICE_LIST
TO SALES_MANAGER;
```

А теперь благодаря команде из следующего примера менеджер по продажам может также удалять из таблицы ненужные строки:

```
GRANT DELETE
ON RETAIL_PRICE_LIST
TO SALES_MANAGER;
```

### Ограничения ссылочной целостности угрожают вашим данным

Возможно, вы думаете, что если можете контролировать функции просмотра, создания, изменения и удаления в таблице, то вы надежно защищены. В большинстве случаев это правда. Однако с помощью непрямого метода опытный хакер все равно имеет возможность сделать подкоп под вашу базу.

Правильно спроектированная реляционная база данных имеет ссылочную целостность, т.е. данные в одной таблице из базы данных согласуются с данными во всех других таблицах. Чтобы обеспечить ссылочную целостность, проектировщики баз данных применяют к таблицам такие ограничения, которые относятся к вводимым в таблицу

данным. И если ваша база данных имеет ограничения ссылочной целостности, то какой-либо пользователь может создать новую таблицу, где в качестве внешнего ключа используется столбец из засекреченной таблицы вашей базы. Вполне возможно, что в таком случае этот столбец можно использовать в качестве канала кражи конфиденциальной информации.

Скажем, вы, например, являетесь знаменитым аналитиком с Уолл-Стрит. Многие верят в точность вашего биржевого анализа, и если вы рекомендуете подписчикам своего бюллетеня какие-либо ценные бумаги, то многие люди их покупают, и стоимость этих бумаг растет. Ваш анализ хранится в базе данных, в которой находится таблица FOUR_STAR. В этой таблице содержатся самые лучшие рекомендации, предназначенные для следующего выпуска вашего бюллетеня. Естественно, что доступ к FOUR_STAR ограничен, чтобы ни слова не просочилось в массу инвесторов, пока бюллетень не дойдет до ваших платных подписчиков.

Вы будете находиться в уязвимом положении, если кому-то удастся создать таблицу, в качестве внешнего ключа которой используется поле таблицы FOUR_STAR, содержащее названия ценных бумаг. Вот, например, команда, создающая такую таблицу:

```
CREATE TABLE HOT_STOCKS (
STOCK CHARACTER (30) REFERENCES FOUR_STAR
);
```

Теперь хакер может вставить в свою таблицу HOT_STOCKS названия всех ценных бумаг с Нью-йоркской фондовой биржи. Те названия, которые будут успешно вставлены, подскажут ему, что именно находится в вашей конфиденциальной таблице. Благодаря быстродействию компьютеров хакеру не потребуется много времени, чтобы вытащить весь ваш список ценных бумаг.

Вы сможете защитить себя от проделок, аналогичных показанной в предыдущем примере, если будете остерегаться вводить операторы такого рода:

```
REFERENCES (STOCK)
ON FOUR_STAR
TO IMASECRET_HACKER;
```

***Совет:***

*Не предоставляйте полномочия тем, кто может ими злоупотребить. Конечно, гарантии у людей на лбу не написаны. Но если вы кому-либо не собираетесь давать свой новый автомобиль для дальней поездки, то, скорее всего, не должны также предоставлять этому человеку и полномочия REFERENCES на ценную таблицу*.

Этот пример показывает первую уважительную причину, чтобы осторожно обращаться с полномочиями REFERENCES. А ниже указаны еще две причины, чтобы быть осторожными с этим видом полномочий.

- Если кто-то другой установил в таблице HOT_STOCKS ограничение с помощью ключевого слова RESTRICT (ограничить), а вы пытаетесь из своей таблицы удалить строку, то СУБД сообщит, что вам этого делать нельзя, так как будет нарушена ссылочная целостность.
- Вы решаете, что для уничтожения вашей таблицы нужна команда DROP (прекратить), и обнаруживаете, что уничтожить свое ограничение (или свою таблицу) вначале должен кто-то другой.

Следовательно, предоставление другим лицам возможности устанавливать ограничения целостности на вашу таблицу не только создает потенциальную угрозу безопасности, но иногда и усложняет вашу работу.

### Делегирование ответственности за безопасность

Если вы хотите сохранять свою систему в безопасности, то должны строго ограничить полномочия доступа, которые вы предоставляете, и круг тех людей, кому вы

предоставляете эти полномочия. Однако те, кто не может работать из-за отсутствия доступа, скорее всего, будут постоянно вам надоедать. Чтобы иметь возможность сосредоточиться, вам придется кому-то делегировать часть своей ответственности за безопасность базы данных. В SQL такое Делегирование выполняется с помощью предложения WITH GRANT OPTION (с возможностью предоставления). Проанализируйте следующий пример:

```
GRANT UPDATE
ON RETAIL_PRICE_LIST
TO SALES_MANAGER WITH GRANT OPTION
```

Этот оператор похож на приведенный в предыдущем примере с GRANT UPDATE в том смысле, что дает возможность менеджеру по продажам обновлять розничный прайс – лист. Но, кроме того, новый оператор еще дает ему право предоставлять полномочия на обновление любому, кому он захочет. И если вы используете такую форму оператора GRANT, то обязаны не только быть уверены, что менеджер по продажам разумно использует предоставленные полномочия, но также должны быть уверены, что он будет осторожно предоставлять подобные полномочия другим пользователям.

***Внимание:***

*Крайняя доверчивость означает и крайнюю уязвимость. Будьте чрезвычайно осторожны, используя подобные операторы:*

```
GRANT ALL PRIVILEGES
ON FOUR_STAR
TO BENEDICT_ARNOLD WITH GRANT OPTION
```

Здесь пользователь BENEDICT_ ARNOLD получает **все** полномочия FOUR_STAR с возможностью передачи этих полномочий другим лицам.

## Создание и поддержка простой базы данных

В этой главе…

- Создание, изменение и удаление таблицы из базы данных с помощью инструмента RAD.

- Создание, изменение и удаление таблицы из базы данных с помощью SQL.
- Перенос базы данных в другую СУБД.

---

В течение своей истории компьютерные технологии менялись так быстро, что в череде их технологических "поколений" иногда нетрудно и запутаться. Вначале для работы с большими базами данных использовались языки высокого уровня, так называемые языки третьего поколения – FORTRAN, COBOL, Basic, Pascal и C. Затем вошли в употребление языки, специально предназначенные для использования с базами данных, например dBASE, Paradox и R:BASE. (А к какому поколению отнести эти языки? Может, к третьему с половиной?) Самым последним этапом этого прогресса является появление сред разработки, в которых приложения создаются с минимумом процедурного программирования или совсем без такового. Это, например, такие среды, как Access, Delphi, IntraBuilder или C++Builder– языки четвертого поколения. С помощью этих графических объектно-ориентированных инструментов (их еще называют инструментами **быстрой разработки приложений** (**rapid application development**), или **RAD-инструментами**) из элементов управления можно собирать готовые приложения.

Как вы узнали в главах 1-3, SQL полноценным языком не является. Таким образом, ни в одну из упомянутых категорий он не входит. Хотя в SQL используются команды, аналогичные командам языков третьего поколения, но, в сущности, он, подобно языкам четвертого поколения, является непроцедурным. Впрочем, не имеет значения, к какому классу отнести SQL. Ведь его можно использовать в сочетании с инструментами разработки как третьего, так и четвертого поколений. Код SQL можно писать самостоятельно, а можно с помощью графических инструментов, и тогда соответствующий код будет генерироваться средой разработки. Все равно, к удаленной базе данных пойдут только команды SQL.

В этой главе рассказывается, как с помощью RAD-инструмента создать, изменить и удалить простую таблицу, а затем вы узнаете, как то же самое проделать с помощью SQL.

### Создание простой базы данных с помощью RAD-инструмента

Люди пользуются базами данных потому, что им нужно сохранять важную информацию. Иногда такая информация является простой, а иногда – нет. Но в любом случае хорошая система управления базами данных должна предоставить ту информацию, которая вам нужна. В некоторых СУБД можно использовать только SQL. А в других, которые называются RAD-инструментами, имеется объектно-ориентированная графическая среда.

Также имеются СУБД, поддерживающие оба этих подхода. В следующих разделах с помощью графического инструмента, предназначенного для проектирования баз данных, будет создана простая база, состоящая из одной таблицы. Это делается для того, чтобы вы могли увидеть, из каких операций состоит изучаемый процесс. Хотя я буду использовать Microsoft Access, но и в других средах разработки, работающих в Windows, процедура создания базы данных почти такая же.

### Правдоподобный сценарий

Первый ваш шаг по созданию базы данных – решить, какую информацию следует в нее заносить. Вот вам правдоподобный пример. Представьте, что вы только что выиграли 101 миллион долларов в лотерее Пауэрболл. (В реальной жизни в вас чаще попадет молния или метеорит.) И тут, откуда ни возьмись, начинают появляться люди, о которых вы не слышали годами, и даже друзья, о которых вы уже забыли. У некоторых из них имеются безошибочные и беспроигрышные деловые предложения, в которые требуются ваши инвестиции. У других есть достойные инициативы, которые могли бы выиграть от вашей поддержки. Как хороший распорядитель своего нового богатства вы понимаете, что не все деловые предложения одинаково хороши. Поэтому, чтобы не упустить ни одной из возможностей и сделать справедливый и беспристрастный выбор, вы принимаете решение – поместить все предложения в базу данных.

Вы решили, что в базу по каждому предложению будут заноситься данные таких видов.

- Имя (**First Name**).
- Фамилия (**Last Name**).
- Адрес (**Address**).
- Город (**City**).
- Штат или провинция (**State or province**).
- Почтовый код (**Postal code**).
- Телефон (**Phone**).
- Кто таков (**How Known**) (ваши взаимоотношения с тем, кто внес предложение).
- Само предложение (**Proposal**).
- Бизнес или благотворительность (**Business or charity**).

Кроме того, не желая слишком заниматься подробностями, вы решили заносить все эти данные в единственную таблицу базы данных. Запустив среду разработки Access, вы начинаете пристально всматриваться в экран (рис. 4.1).

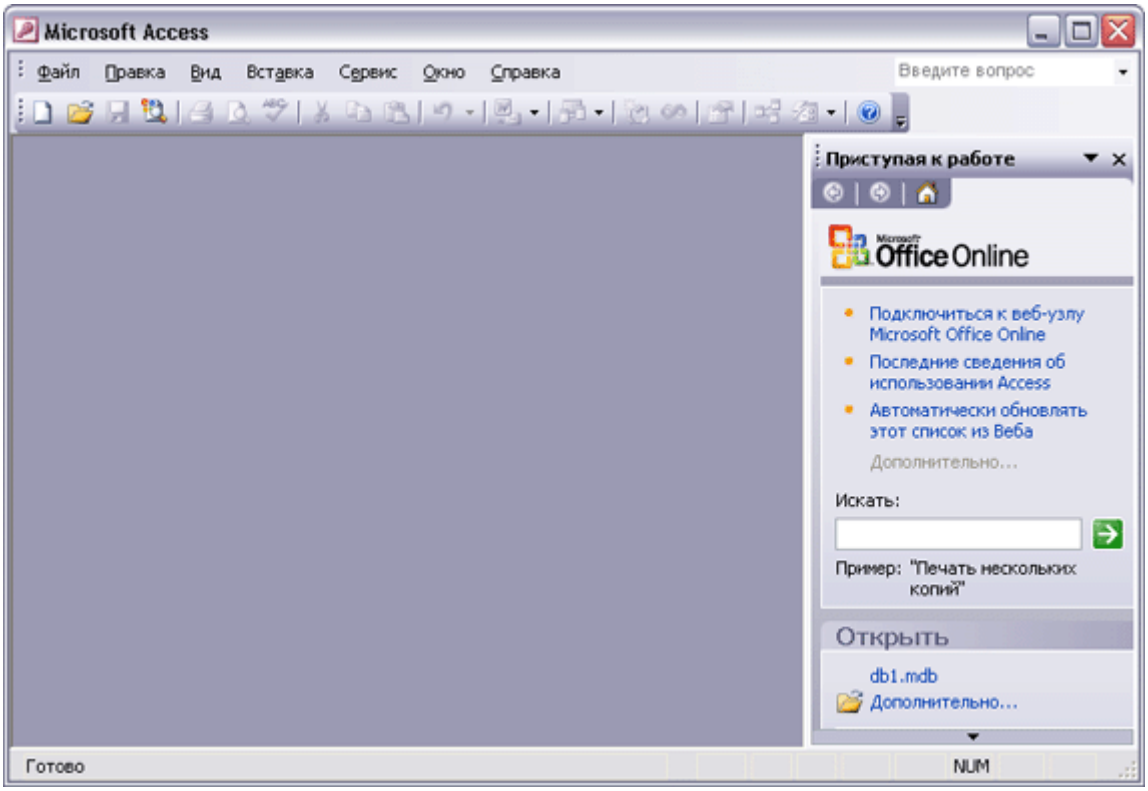


**Рис. 4.1**. Экран, появившийся при создании новой базы данных в Microsoft Access

### Экран проекта дает полный контроль

Экран, представленный на рис. 4.1, содержит намного больше информации, чем выводили программы СУБД предыдущих поколений. В 1980-е годы общение с типичной системой СУБД начиналось с пустого экрана, оттеняемого подсказкой из одного символа. С тех пор управление базами данных прошло долгий путь, и теперь намного легче узнать о том, с чего надо начинать. В правой части рабочего пространства находится область задач Приступая к работе с несколькими разделами.

- Раздел Microsoft Office Online содержит команды, которые предоставляют доступ к интерактивным ресурсам Microsoft, связанным с приложением Access, и поле Искать.
- Раздел Открыть содержит список баз данных, которые были открыты в последнее время, и команду Создать файл.

Чтобы создать таблицу базы данных в приложении Access, выполните следующие действия.

1. Откройте программу Access и выберите в разделе Открыть (область задач Приступая к работе) команду Создать файл. Отобразится область задач Создание файла.
2. В разделе Создание выберите команду Новая база данных. Отобразится диалоговое окно Файл новой базы данных.
3. Присвойте какое-либо имя базе данных и сохраните ее в нужной папке. По умолчанию база данных будет сохранена в папке Мои документы, однако вы можете сохранить файл базы данных в любой выбранной вами папке. Поскольку в этом примере мы будем создавать базу данных, в которой будут отслеживаться данные о выигрышах в лотерее Пауэрбол, я назвал ее POWER.

   После выполнения всех действий отобразится окно POWER: база данных.

4. Выберите опцию Создать таблицу в режиме конструктора. Можно выбрать опцию Создать таблицу с помощью Мастера. Но в данном случае это будет не лучший выбор, потому что мастера не отличаются достаточной гибкостью. Мастер таблиц создает таблицу, состоящую из списка заранее определенных столбцов. Третий вариант создания базы данных Создание таблицы путем ввода данных также не достаточно подходит для создания серьезных проектов, поскольку предполагает относительно данных много допущений, принятых по умолчанию в Access.

   Дважды щелкните на опции, чтобы появилось окно для создания таблицы (рис. 4.2).

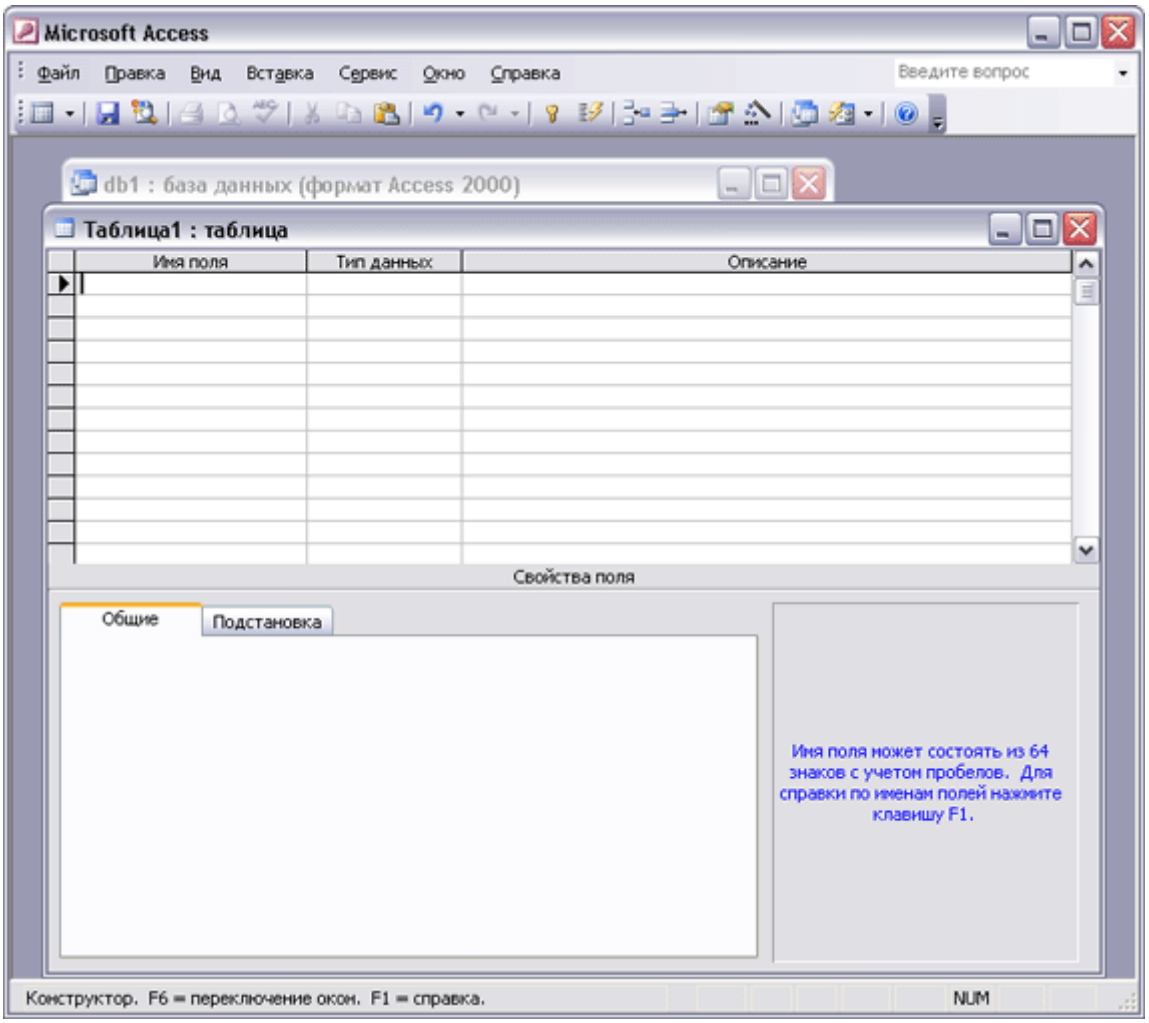


**Рис. 4.2**. Диалоговое окно создания таблицы

5. Введите в поля Имя поля, Тип данных и Описание информацию об атрибутах каждого поля вашей таблицы. После того как вы сделаете необходимую запись в столбце Имя поля, в столбце Тип данных отобразится раскрывающийся список, из которого вы можете выбрать подходящий для этого поля тип данных.

   Обратите внимание, в правом нижнем углу на рис. 4.3 приведены значения, принятые по умолчанию для некоторых свойств поля. По необходимости вы можете сделать записи значений свойств для всех полей.

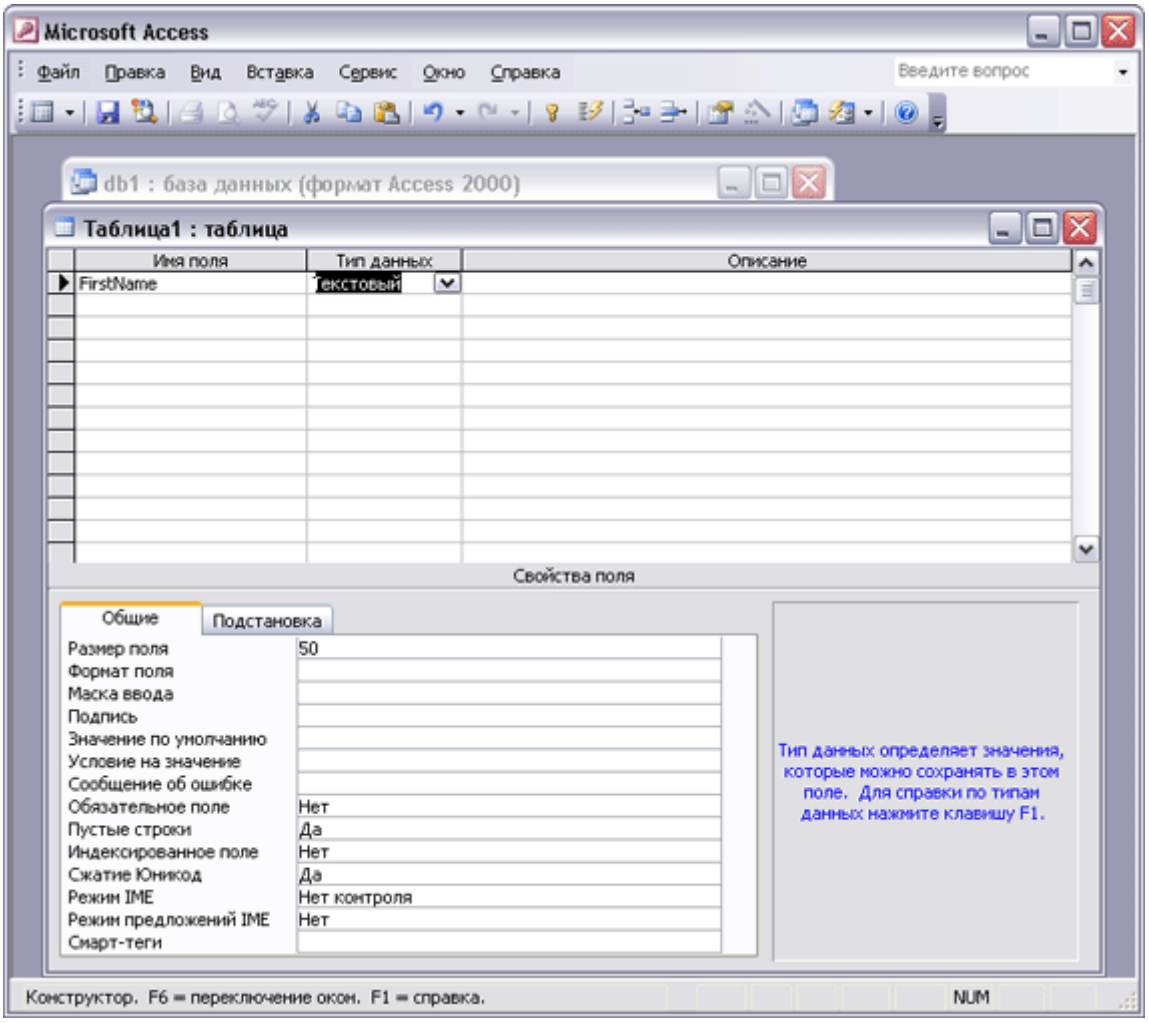


**Рис. 4.3**. Значения по умолчанию для свойств поля FirstName, показанные в диалоговом окне создания таблицы

***Технические подробности:***

*В Access вместо названия столбец используется название поле. Первоначальные системы обработки файлов не были реляционными, и в них использовались термины "файл", "поле" и "запись", которые характерны для систем плоских файлов*.

Не исключено, что вы захотите оставить или соответствующим образом изменить имеющиеся значения. Например, для создаваемого вами поля **FirstName** (имя) значение свойства Размер поля по умолчанию составляет 50 символов. Возможно, это значение больше, чем нужно. И если для этого свойства задать, например, значение 15 символов, то будет сэкономлен определенный объем дискового пространства. На рис. 4.4 приведено диалоговое окно создания таблицы после того, как введены значения свойств для всех полей.

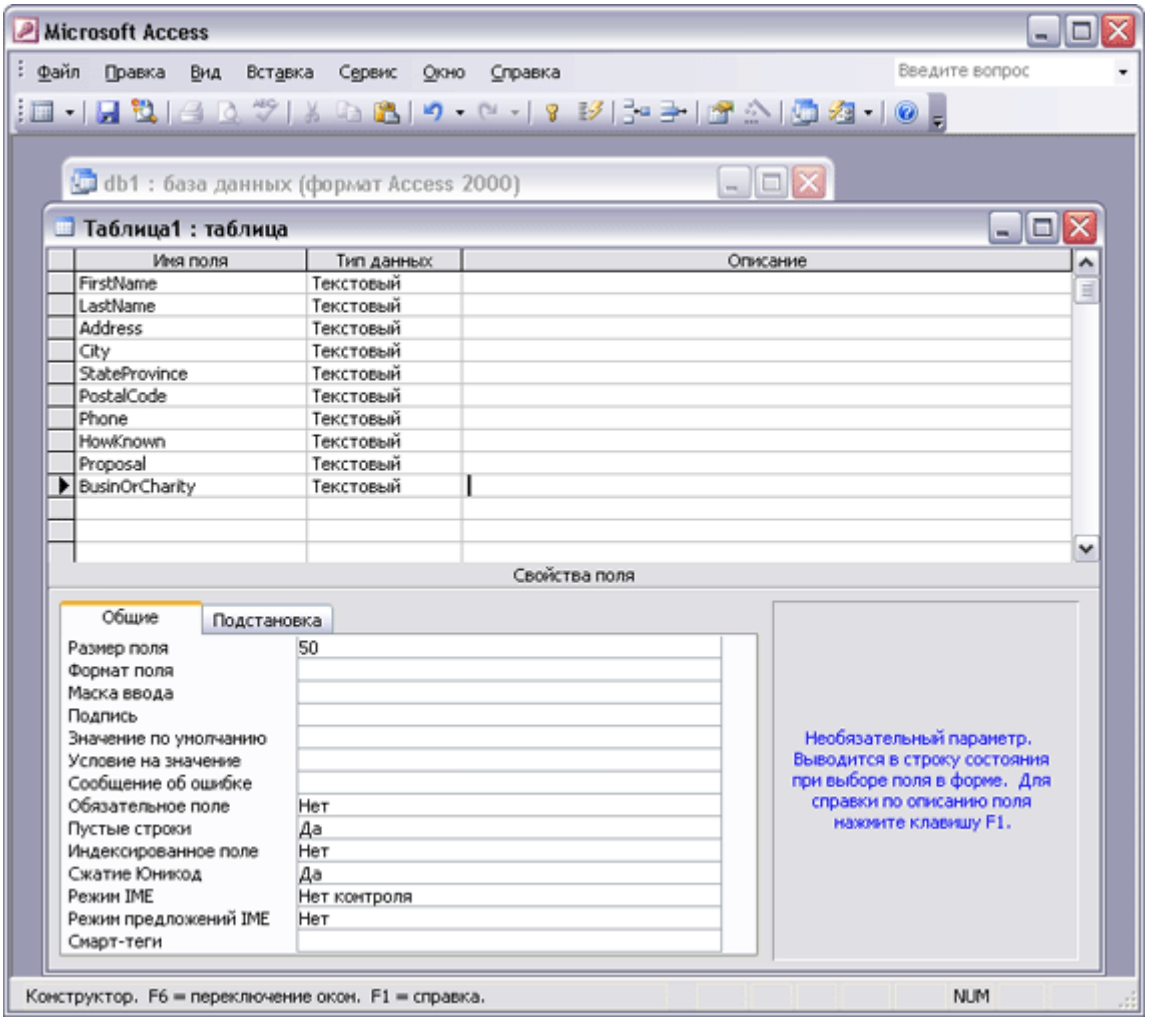


**Рис. 4.4**. Диалоговое окно создания таблицы со всеми определенными полями

6. Теперь, когда таблица определена, сохраните ее, выбрав команду Файл – Сохранить.

На рис. 4.5 представлено диалоговое окно с предложением ввести имя таблицы, которую вы хотите сохранить. Свою таблицу я назвал PowerDesign. Эта таблица, созданная в режиме конструктора, будет состоять из победителей лотереи Пауэрбол.

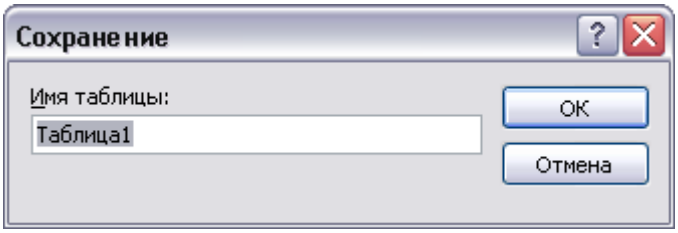


**Рис. 4.5**. Текстовое поле Имя таблицы в диалоговом окне Сохранить как

***Внимание:***

*При попытке сохранить новую таблицу появляется еще одно диалоговое окно (рис. 4.6). В этом окне сообщается, что вы не определили первичный ключ, и задается вопрос, не хотите ли вы сделать это сейчас. О первичных ключах мы поговорим чуть позднее, в разделе "Определение первичного ключа", а сейчас щелкните на кнопке Нет. Первичный ключ сейчас определять не надо, поэтому не задерживайтесь на нем. Сейчас ваша задача – сохранить таблицу*.

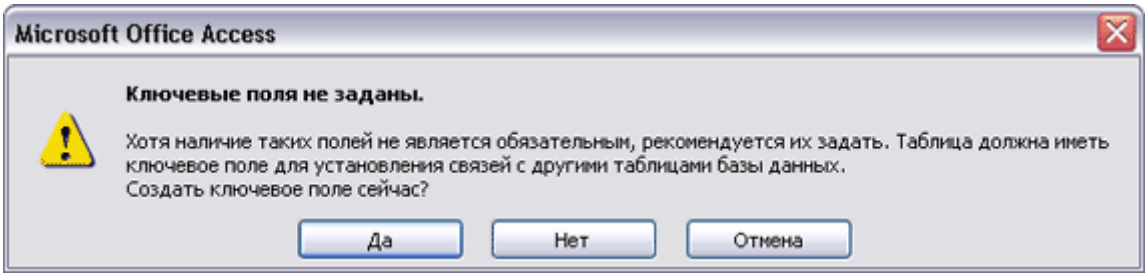


**Рис. 4.6**. Диалоговое окно определения первичного ключа

Сохранив таблицу, вы, возможно, решите, что первоначальный замысел нуждается в улучшении. (Об этом вы прочитаете в разделе "Изменение структуры таблицы".) Со своими заманчивыми предложениями к вам подходило столько людей, и в результате выяснилось, что некоторые из этих ребят одновременно и тезки и однофамильцы. Чтобы не было путаницы, вы решили к каждой записи из таблицы базы данных добавить уникальный номер предложения. Таким образом, вы сможете отличить одного Дэвида Ли от другого.

### Изменение структуры таблицы

Как правило, созданные вами таблицы не получатся с первого раза такими, как надо. Если вы работаете для кого-то, то будьте уверены – ваш клиент ждет, пока вы наконец-то создадите базу данных, чтобы кое-что вам сообщить. И тут вы узнаете, что руководство желает, чтобы вводилась информация еще по одному виду данных, а возможно, даже по нескольким.

Если вы создаете базу данных для себя, то недостатки в ее структуре, которых совсем не было видно до реального создания базы, уже после создания этой структуры неизбежно всплывут на поверхность. Возможно, что к вам, например, начинают поступать предложения не только из США. Тогда нужно добавить столбец **Country** (страна). Или вы решили, что будет полезен еще и адрес электронной почты. В любом случае придется вернуться назад и переделать то, что вы создали. Такая возможность– переделывать уже созданное– имеется во всех RAD-инструментах. Чтобы показать типичный пример, я, используя Access, внесу некоторые изменения в созданную мной таблицу. Другие инструменты работают подобным образом.

Предположим, нужно добавить уникальный номер предложения, чтобы можно было различать предложения от разных людей с одинаковыми именами и фамилиями. Если уж на то пошло, то можно также добавить еще два поля. Это, во-первых, еще одно поле Address2 (адрес 2), которое пригодится для тех, у кого несколько адресов, и, во-вторых, поле Country – для предложений из других стран.

Чтобы вставить новую строку и внести изменения, соответствующие требованиям, выполните следующие действия.

1. В диалоговом окне определения таблицы поместите курсор в самую верхнюю строку и выберите команду Вставить – Строки (рис. 4.7).

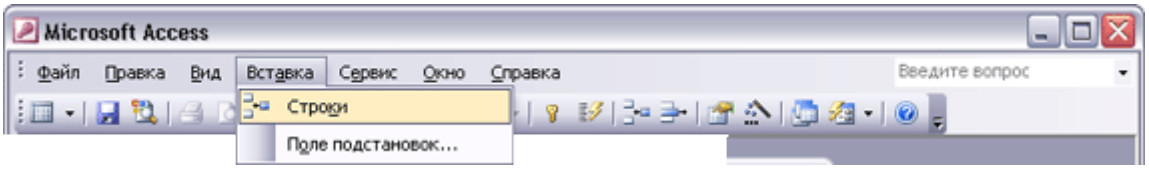


**Рис. 4.7**. В определение таблицы PowerDesign вставлена новая строка

Там, где находится курсор, появится новая незаполненная строка, а все остальные строки сдвинутся вниз.

2. Введите заголовки для тех столбцов, которые хотите использовать в своей таблице. В поле Имя поля я поместил ProposalNumber (номер предложения), в поле Тип данных – AutoNumber (счетчик), а в Описание – фразу "Unique identifier for each row of the

PowerDesign table" (уникальный идентификатор для каждой строки таблицы POWER). AutoNumber – числовой тип данных, значение которого автоматически увеличивается для каждой последующей строки таблицы. Похожим образом под полями Address (адрес) и PostalCode (почтовый код) я соответственно вставил поля Address2 (адрес 2) и Country (страна). Результат представлен на рис. 4.8.

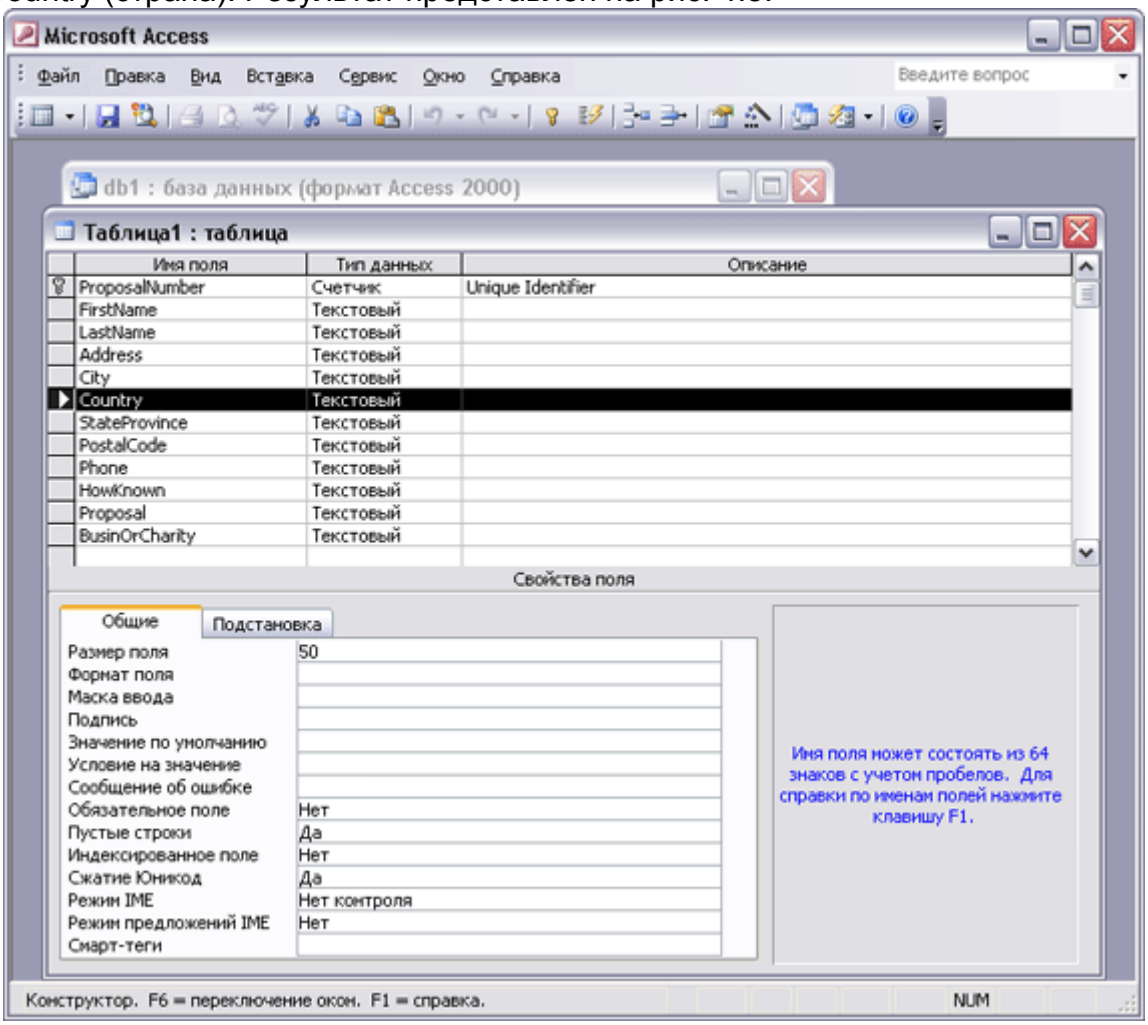


**Рис. 4.8**. Переделанное определение таблицы должно выглядеть примерно так

### Определение первичного ключа

Первичный ключ таблицы – это поле, которое однозначно определяет каждую строку.

Поле **ProposalNumber** (номер предложения) является хорошим кандидатом на роль первичного ключа таблицы PowerDesign, потому что значения этого поля однозначно определены для каждой строки таблицы. Это единственное поле, относительно которого вы можете быть уверены, что оно нигде в таблице не дублируется. Для того чтобы назначить это поле первичным ключом таблицы, поместите курсор в строке ProposalNumber, находящейся в диалоговом окне определения данных, а затем щелкните на пиктограмме Первичный ключ, которая расположена в центре панели инструментов Проектирование таблиц. В результате в самом левом столбце диалогового окна определения таблицы появится пиктограмма ключа. Это означает, что поле ProposalNumber теперь является первичным ключом таблицы PowerDesign.

На рис. 4.9 показано, как выглядит окно определения таблицы после объявления первичного ключа.

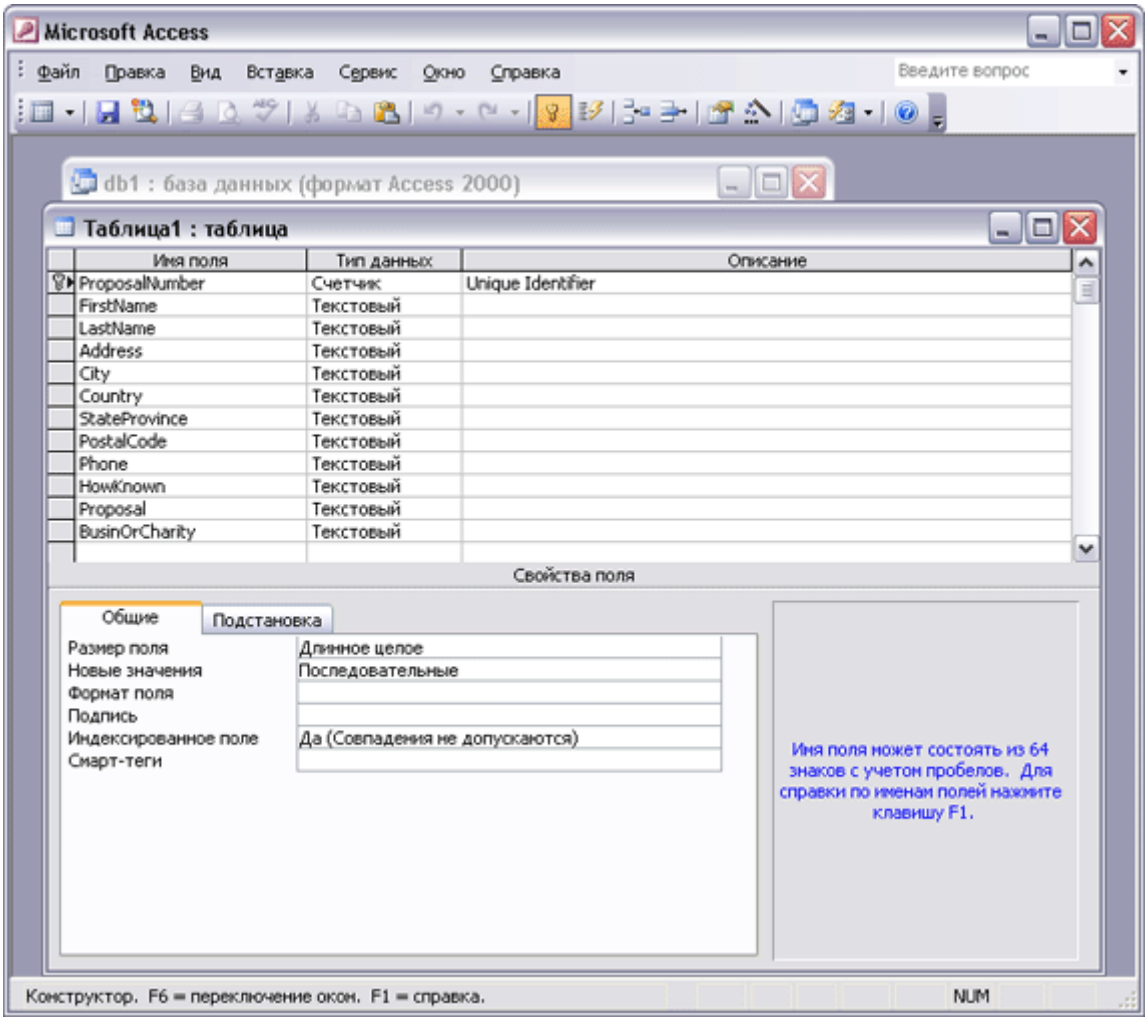


**Рис. 4.9**. Поле ProposalNumber объявлено первичным ключом

### Создание индекса

Так как количество получаемых вами предложений об инвестициях и благотворительности может легко дойти до нескольких тысяч, то нужен способ, с помощью которого можно быстро выбрать интересующие вас записи. Вы сможете выполнить эту задачу самыми разными способами. Например, просмотреть все предложения, сделанные вашими братьями. Эти предложения можно эффективно выбрать, если использовать содержимое поля **LastName** (фамилия), как показано в следующем примере:

```
SELECT * FROM PowerDesign
WHERE LastName = 'Marx';
```

Впрочем, такая стратегия не работает для предложений, сделанных всеми вашими шуринами, деверями, свояками *(по-английски любой из них называется одинаково – "brother-in-law". – Примеч. пер.)*, но эти предложения можно получить, используя другое поле,**HowKnown** (кто таков), как показано в следующем примере:

```
SELECT * FROM PowerDesign
WHERE HowKnown = 'brother-in-law';
```

Что ж, запросы эти работают, но они, возможно, не будут работать очень быстро, если таблица PowerDesign достаточно большая (десятки тысяч записей). SQL перебирает всю таблицу построчно, отыскивая значения, которые удовлетворяют предложению WHERE. Работу можно значительно ускорить, применяя в таблице PowerDesign **индексы**. (**Индекс** – это таблица с указателями. Каждая строка в индексе указывает на соответствующую строку в таблице с данными.)

Для каждого из способов, которые требуются для доступа к вашим данным, можно определять свой индекс. И если вы в таблице с данными добавляете, изменяете или удаляете строки, то снова сортировать эту таблицу не нужно – следует только обновить ее индексы.

А индекс можно обновить намного быстрее, чем сортировать целую таблицу. Создав индекс, строки которого выстроены в нужном порядке, вы с помощью этого индекса можете почти мгновенно переходить к нужным строкам таблицы данных.

***Совет:***

*Так как поле* ***ProposalNumber*** *(номер предложения) одновременно и уникальное, и короткое, то с его помощью можно быстрее всего добраться к отдельной записи. Поэтому первичный ключ любой таблицы всегда должен быть индексирован. В Access это делается автоматически. Однако, чтобы использовать это поле, необходимо знать его значение в нужной вам записи. Вам могут потребоваться и дополнительные индексы, создаваемые на основе других полей, таких как* ***LastName*** *(фамилия),* ***PostalCode****(почтовый код) или* ***HowKnown*** *(кто таков). Если в таблице данных, проиндексированной по LastName, в результате поиска будет найдена первая строка, где значением этого поля является Marx, то будут найдены и все строки с таким же значением этого поля. В индексе ключи для всех этих строк идут друг за другом. Поэтому все строки, относящиеся к Chico, Groucho, Harpo, Zeppo и Karl, можно получить почти так же быстро, как и для одного только Chico*.

В результате создания индекса возникает дополнительная нагрузка на вашу систему, от чего ее работа будет немного замедляться. Это замедление необходимо сравнивать с увеличением скорости доступа к записям в результате использования индекса. Если индексировать поля, часто используемые для доступа к записям из большой таблицы, то замедление работы оправдано. Однако если создавать индексы для полей, которые никогда не будут использоваться для доступа к записям, то потери времени во много раз превысят его экономию. Также не имеет смысла создавать индексы для полей, которые не позволяют отличить одну запись от другой. Например, поле BusinessOrCharity (бизнес или благотворительность) просто разбивает все записи в таблице только на две категории, поэтому хороший индекс на основе этого поля создать нельзя.

***Помни:***

*Эффективность индекса в разных реализациях бывает разной. Если перенести базу данных с одной платформы на другую, то индексы, лучше всех работавшие в первой системе, могут плохо работать во второй. Случается, что база данных хуже работает с индексами, чем при их полном отсутствии. Индексы приходится заново настраивать для каждой конкретной конфигурации СУБД и аппаратного обеспечения. Какая из разных схем индексирования в общем работает лучше остальных, приходится определять опытным путем, принимая во внимание в каждом случае скорость получения данных и их обновления*.

Чтобы создать индексы для таблицы PowerDesign, щелкните на пиктограмме Индексы, расположенной справа от пиктограммы Первичный ключ на панели инструментов Конструктор таблиц. Появится диалоговое окно Индексы, в котором уже есть поля Postal-Code и ProposalNumber. Диалоговое окно Индексы представлено на рис. 4.10.

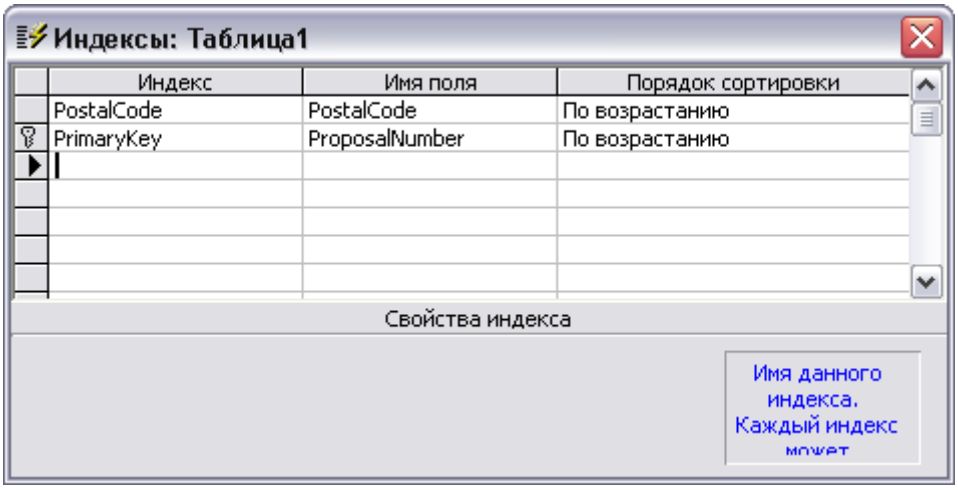


**Рис. 4.10**. Диалоговое окно Индексы

***Совет:***

*Access автоматически создает индекс для поля PostalCode, поскольку это поле часто используется для поиска данных*.

Вы можете заметить, что, в отличие от ProposalNumber, поле PostalCode может не являться первичным ключом, и его значения не обязательно уникальны. Можно создать индексы для полей **LastName** и **HowKnown**, так как они, вероятно, также будут использоваться для доступа к данным. На рис. 4.11 изображено, как выглядят эти новые индексы.

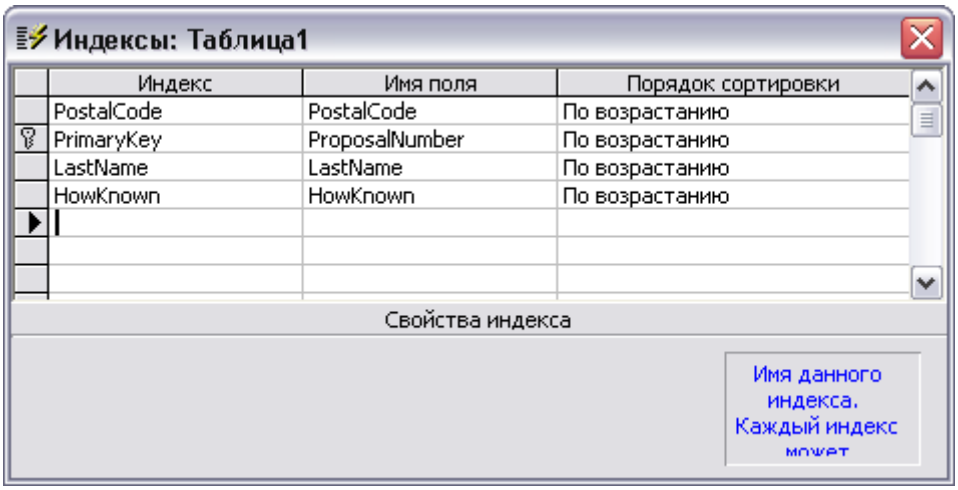


**Рис. 4.11**. Определение индексов для полей LastName и HowKnom

Создав все нужные вам индексы, можно сохранить новую табличную структуру, выбрав команду Файл – Сохранить или щелкнув на пиктограмме с изображением дискеты.

***Совет:***

*Конечно, если вы применяете не Microsoft Access, то все, что говорилось в этом разделе, к вам не относится. Впрочем, при создании таблицы базы данных и ее индексов с помощью другого, отличного от Access, RAD-инструмента выполняется примерно та же последовательность действий*.

### Удаление таблицы

Перед тем как таблица PowerDesign приобрела нужную вам структуру, вы, возможно, успели создать несколько промежуточных вариантов этой таблицы, которые не являются окончательными. Присутствие в системе этих вариантов может впоследствии запутывать пользователей. Поэтому, пока вы еще помните, что к чему, лучше всего эти таблицы удалить. Для удаления таблицы выберите ее имя в окне базы данных и щелкните на пиктограмме (рис. 4.12).

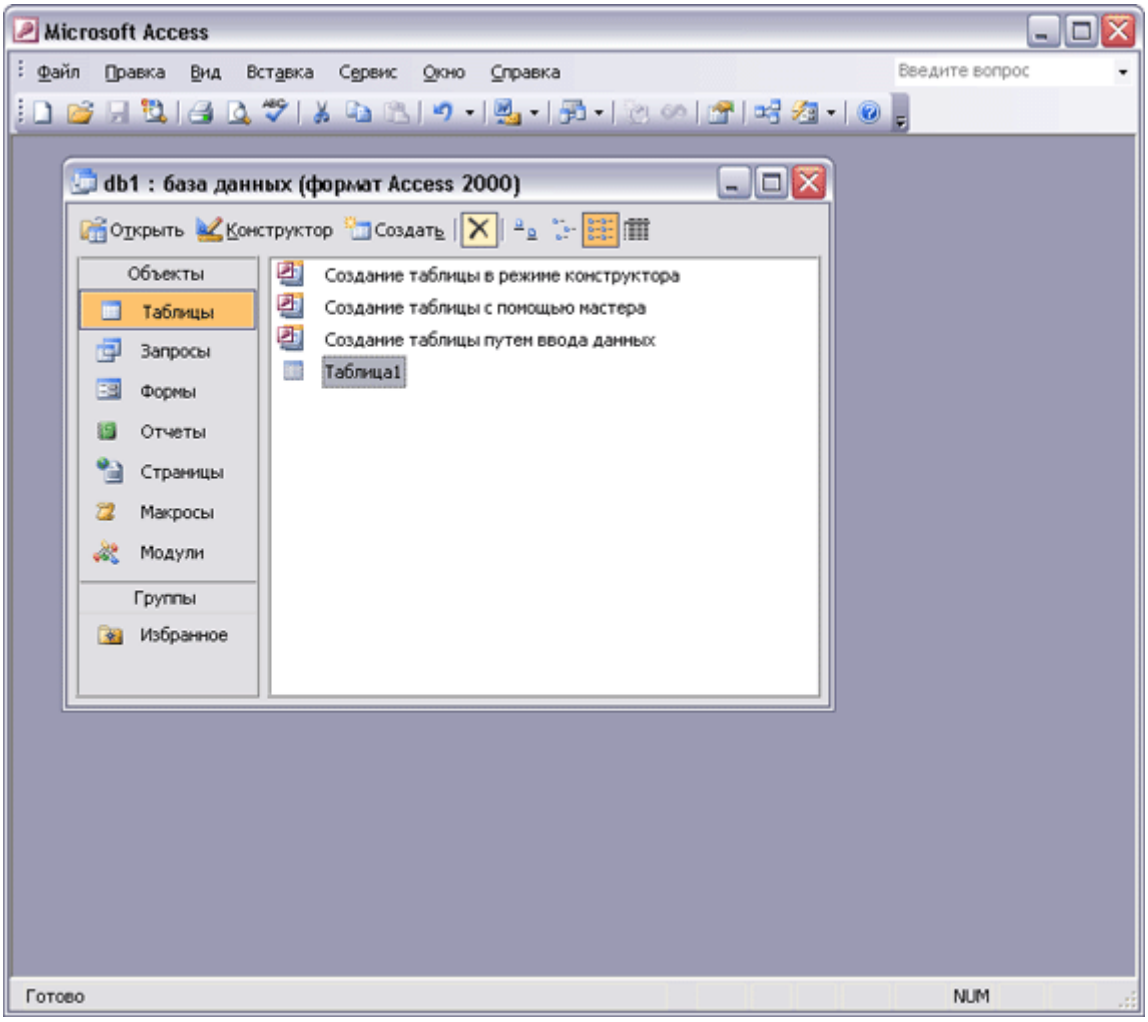


**Рис. 4.12**. Чтобы удалить таблицу, выберите ее имя и щелкните на пиктограмме

Приложение Access выдаст сообщение с вопросом, действительно ли вы хотите удалить выбранную таблицу. После подтверждения таблица будет немедленно удалена.

***Помни:***

*Если Access удаляет таблицу, то также удаляются все связанные с ней таблицы и все ее индексы*.

## Создание таблицы Power Design с помощью DDL

Все действия по определению базы данных, которые можно выполнять с помощью RAD-инструмента, такого как Access, можно также выполнять и с помощью SQL. В этом случае вместо щелчков мышью на элементах меню выполняется ввод команд с помощью клавиатуры. Те, кто предпочитает манипулировать графическими объектами, считают, что RAD-инструменты являются легкими и естественными для понимания и изучения. Другие же, кому больше нравится складывать слова в предложения, имеющие определенную логику, считают, что более легкими и естественными являются все-таки команды SQL. Так как некоторые вещи легко представить, используя объектную парадигму, а с другими легко справляться с помощью SQL, то полезно хорошо знать оба метода.

В следующих разделах будет применяться SQL, чтобы выполнять те же действия по созданию, изменению и удалению таблицы, для которых в предыдущем разделе использовался RAD-инструмент.

## Использование языка SQL с приложением Microsoft Access

Приложение Access было разработано как инструмент быстрой разработки приложений (RAD), не требующий программирования. Несмотря на то что можно писать и выполнять команды на языке SQL непосредственно в Access, чтобы сделать это, можно зайти также с "черного хода". Для того чтобы открыть основной редактор, который используется для ввода SQL-кода, выполните следующие действия.

1. Откройте базу данных, а затем выберите из списка Объекты опцию Запросы.
2. В области задач, расположенной в правой части окна, выберите опцию Создание запроса в режиме конструктора. Отобразится диалоговое окно Добавить таблицу.
3. Выберите любую из таблиц, щелкните на кнопках Добавить и Закрыть. Не обращайте внимание на курсор, который мигает в только что созданном окне Запрос.
4. В главном меню Access выберите команду Вид Режим SQL. Отобразится окно редактора со стартовым оператором языка SQL SELECT.
5. Удалите оператор SELECT, а затем введите необходимый оператор SQL.
6. Закончив работу, щелкните на пиктограмме Сохранить. Access предложит ввести имя для только что созданного запроса.
7. Введите имя запроса и щелкните на кнопке ОК.

Только что созданная команда будет сохранена и выполнена позже как запрос. К сожалению, Access не выполняет весь диапазон команд SQL. К примеру, оно не выполняет команду CREATE TABLE. Однако после создания таблицы можно выполнять практически любое необходимое преобразование с находящимися в ней данными.

### Создание таблицы

При работе с полнофункциональной СУБД, например, такой, как Microsoft SQL Server, Oracle 9i или IBM DB2, в процессе создания таблицы с помощью SQL необходимо вводить ту же информацию, что и при создании таблицы с помощью какого-либо RAD-инструмента. Разница здесь в том, что RAD-инструмент помогает это делать, предоставив в ваше распоряжение диалоговое окно создания таблицы (или какую-либо подобную структуру) и не позволяя вводить неправильные имена полей, типы или размеры. SQL столько внимания вам не уделит. Работая с SQL, следует точно знать с самого начала, что именно надо делать. Необходимо ввести целый оператор CREATE TABLE, прежде чем SQL обратит на него внимание, не говоря уже о том, чтобы сообщить, нет ли в операторе каких-либо ошибок.

Следующая команда создает таблицу, идентичную созданной ранее:

| CREATE TABLE PowerSQL | |
|---|---|
| ProposalNumber | SMALL INT |
| FirstName | CHAR (15), |
| LastName | CHAR (20), |
| Address | CHAR (30), |
| City | CHAR (25), |
| StateProvince | CHAR (2), |
| PostalCode | CHAR (10), |
| Country | CHAR (30), |
| Phone | CHAR (14), |
| HowKnown | CHAR (30), |
| Proposal | CHAR (50), |
| BusinOrCharity | CHAR (1); |

Как видно, информация в сущности та же, что и при создании таблицы с помощью инструментов RAD (как описывалось ранее в этой главе). Можно отдать предпочтение любому способу создания таблиц. Впрочем, что хорошо в языке SQL – так это его

универсальность. Один и тот же стандартный синтаксис будет работать в любой системе управления базами данных.

***Помни:***
*Любые усилия, вложенные в изучение SQL, будут оправдываться в течение долгого времени, потому что быстро сходить со сцены этот язык не собирается. А усилия, вложенные в то, чтобы стать экспертом в среде разработки, вероятно, принесут меньшую отдачу. И каким бы прекрасным ни был последний RAD-инструмент, будьте уверены – в течение двух-трех лет его заменит более совершенная технология. Замечательно, если за это время вы сможете возместить усилия, вложенные в изучение данного инструмента! Если сможете, то пользуйтесь им. А если не сможете, то будет мудрее придерживаться старого и испытанного средства. Знание SQL намного дольше будет приносить дивиденды*.

### Создание индекса

**Индексы** – очень важная часть любой реляционной базы данных. Они служат указателями в тех таблицах, в которых содержатся нужные данные. С помощью индекса можно прямо перейти к определенной записи, не выполняя для ее поиска последовательного, запись за записью, просмотра таблицы. Для больших таблиц индексы просто необходимы. Без индексов результат из действительно очень большой таблицы придется, возможно, ждать не секунды, а годы. (Ладно, предположим, что вам не придется ждать годами. Однако некоторые операции выборки данных, если их запустить на выполнение, могут действительно продолжаться достаточно долго. И если у вас в запасе нет ничего лучшего, что могло бы ускорить работу, то вы, вероятно, прервете операцию, не получив никаких результатов. Ведь жизнь все равно продолжается.)

Удивительно, но в спецификации SQL:2003 нет средств для создания индекса. У поставщиков СУБД должны быть собственные реализации индексов. А так как эти реализации не стандартизированы, то вполне могут отличаться друг от друга. Большинство поставщиков реализует средство создания индекса, расширяя SQL оператором CREATE INDEX (создать индекс). Но даже если в двух реализациях используются одни и те же операторы, то способы их выполнения могут быть разными. В различных реализациях используются разные варианты синтаксиса этой команды. Необходимо внимательно изучать документацию по имеющимся у вас СУБД, чтобы знать, как можно создавать индексы с помощью этих систем.

### Изменение структуры таблицы

Для изменения структуры существующей таблицы можно использовать команду SQL ALTER TABLE (изменить таблицу). Интерактивные средства SQL, находящиеся на вашей клиентской станции, не такие удобные, как RAD-инструмент. Этот инструмент показывает табличную структуру, которую затем можно изменить. А используя SQL, необходимо заранее знать и структуру таблицы, и то, каким образом эту структуру следует изменять. В том месте экрана, где находится приглашение, необходимо для внесения изменения ввести соответствующую команду. Впрочем, если в прикладной программе нужно поместить операторы изменения таблицы, то обычно самый легкий способ это сделать – все-таки использовать SQL.

Чтобы добавить в таблицу PowerSQL второе поле для адреса, используйте следующую команду DDL:

```
ALTER TABLE PowerSQL
ADD COLUMN Address__2 CHAR (30);
```

Чтобы расшифровать этот код, не нужно быть гуру SQL. В действительности это может сделать даже профан со слабыми познаниями в английском. Эта команда изменяет таблицу с названием PowerSQL, добавляя в нее новый столбец, который называется Address2,

имеет тип данных CHAR и длину 30 символов. Приведенный пример показывает, насколько легко менять структуру таблиц в базе данных, используя для этого команды DDL из SQL.

Стандарт SQL:2003 разрешает использовать этот оператор для добавления в таблицу столбца, а также удаления имеющегося столбца, как показано в следующем примере:

```
ALTER TABLE PowerSQL
DROP COLUMN Address2;
```

### Удаление таблицы

Таблицу, которая вам не нужна и только зря занимает место на диске, удалить достаточно легко. Всего лишь воспользуйтесь командой DROP TABLE (удалить таблицу), такой, например, как следующая:

```
DROP TABLE PowerSQL;
```

Что может быть проще? Если такой командой удалить таблицу, то будут удалены все ее данные и метаданные. От таблицы не останется и следа.

### Удаление индекса

***Внимание:***

*Удаляя таблицу с помощью команды DROP TABLE, вы одновременно удаляете и все относящиеся к ней индексы. Впрочем, иногда требуется оставить таблицу, но удалить один из ее индексов. Стандарт SQL:2003 не определяет команду DROP INDEX (удалить индекс), но в большинстве реализаций она все-таки есть. Эта команда пригодится тогда, когда ваша система замедлит свою работу до черепашьей скорости и обнаружится, что таблицы в ней индексированы не лучшим образом. Исправление индексов должно привести к резкому увеличению производительности системы. Это, правда, может опечалить пользователей, привыкших в ожидании своих результатов устраивать перекур*.

### Переносимость

В любой из реализаций SQL, которую вы используете, могут быть расширения, дающие ей возможности, не предусмотренные стандартом SQL:2OO3. Одни из этих возможностей, скорее всего, появятся в следующем выпуске спецификации SQL. Другие же характерны только для конкретной реализации и, вероятно, никогда не войдут в стандарт.

Часто эти расширения могут упростить создание нужного вам приложения, и у вас возникнет искушение ими воспользоваться. Это тоже вариант, но учтите – не исключено, что в таком случае придется чем-то пожертвовать. Если когда-нибудь потребуется перенести ваше приложение на другую реализацию SQL, то вам, возможно, придется переписывать те части этого приложения, в которых используются расширения, не поддерживаемые в новой среде. Подумайте о возможности такого переноса, который может произойти когда-нибудь в будущем, а также о том, является ли используемое вами расширение уникальным или все-таки достаточно распространенным.

При долгосрочном использовании приложения, наверное, будет лучше не пользоваться нестандартным расширением, даже если в результате получается некоторая экономия времени. С другой стороны, подобное решение может быть излишней перестраховкой. Тщательно обдумайте каждую из этих возможностей. Чем больше вам известно об имеющихся реализациях и о тенденциях, связанных с их разработкой, тем лучшими могут быть ваши решения.

# Создание многотабличной реляционной базы данных

В этой главе…

- Что должно быть в базе данных
- Определение отношений между элементами базы данных
- Связывание таблиц с помощью ключей
- Проектирование целостности данных
- Нормализация базы данных

---

В этой главе будет представлен пример создания многотабличной базы данных. Первый шаг при проектировании такой базы – решить, что в ней должно быть, а чего не должно. Второй шаг состоит в том, чтобы установить, каким образом имеющиеся в базе элементы будут связаны друг с другом, и создать таблицы с учетом этой информации. Я расскажу, как использовать ключи для получения быстрого доступа к индивидуальным табличным записям и индексам.

База данных должна не просто хранить данные, но и защищать их от повреждений. Далее в этой главе обсуждаются методы защиты целостности данных. Одним из самых важных методов является нормализация, поэтому я расскажу о различных "нормальных" формах и проблемах, решаемых с их помощью.

## Проектирование базы данных

При проектировании базы данных выполните следующие основные действия (подробно о каждом из них рассказывается в последующих разделах).

1. Решите, какие объекты должны быть в вашей базе данных.
2. Установите, какие из этих объектов должны быть таблицами, а какие – столбцами этих таблиц.
3. Определите таблицы в соответствии с тем, как вы решили организовать объекты.

Возможно, вы захотите назначить ключом какой-либо столбец таблицы или комбинацию ее столбцов. **Ключи** позволяют быстро найти в таблице нужную вам строку.

В последующих разделах подробно описываются эти действия, а также рассматриваются некоторые другие вопросы, связанные с проектированием базы данных.

**Действие 1: определение объектов**.

Первое действие, которое предстоит выполнить при проектировании базы данных, – это решить, какие аспекты системы являются достаточно важными, чтобы быть включенными в модель. Каждый такой аспект рассматривайте как объект и составьте список этих объектов – всех, которые только придут вам в голову. И не пытайтесь гадать, каким образом эти объекты связаны друг с другом. Всего лишь попробуйте внести их в список.

***Совет:***

*Было бы полезно иметь команду из нескольких человек, знакомых с системой, которую вы моделируете. Они могли бы проводить "мозговой штурм" и реагировать на высказанные друг другом идеи. Вместе вы, вероятно, разработаете более полный и точный набор объектов*.

Когда вы решите, что получился достаточно полный набор объектов, можете приступать к следующему действию: определить, каким образом эти объекты связаны друг с другом. Некоторые из них являются главными и играют важную роль в получении нужных вам результатов. Другие же являются подчиненными объектами по отношению к главным. Кроме того, следует окончательно решить, какие объекты вообще не являются частью модели.

**Действие 2: определение таблиц и столбцов**.

**Главные объекты** переводятся в таблицы базы данных. И у каждого из главных объектов имеется набор связанных между собой атрибутов, которые переводятся в столбцы соответствующей таблицы. Например, во многих базах данных, применяемых в бизнесе, имеется таблица CUSTOMER (клиент), в которой хранятся имена клиентов, адреса и другая информация о них. Каждый из атрибутов клиента, такой, например, как имя, улица, город, штат, почтовый код, номер телефона и адрес в Internet, становится столбцом в таблице CUSTOMER.

Имеются достаточно легкие правила, позволяющие быстро определить, что должно стать таблицами и какие из атрибутов системы должны принадлежать каждой такой таблице. Возможно, у вас есть причины, чтобы присвоить какой-либо атрибут одной таблице, но также есть причины, чтобы присвоить его другой таблице. Вам следует принять решение на основе той информации, которую нужно получать из базы данных, и того, каким образом должна использоваться эта информация.

***Внимание:***

*При проектировании структуры базы данных важно учитывать интересы ее будущих пользователей, а также лиц, принимающих решения на основе ее информации. Если созданная вами "разумная" структура не будет соответствовать способу, каким эти люди работают с информацией, то в лучшем случае ваша система может разочаровать своих пользователей. Она может даже выдавать неверную информацию, что намного хуже, чем просто быть трудной в использовании. Такого не должно быть! Хорошо подумайте, перед тем как принимать решение о структуре таблиц*.

Рассмотрим пример, который показывает процесс создания многотабличной базы данных. Скажем, вы только что основали VetLab – клиническую микробиологическую лабораторию, где проводятся анализы проб, присланных из ветеринарных фирм. Конечно, вам бы хотелось иметь данные:

- о клиентах;
- о выполненных анализах;
- о сотрудниках;
- о заказах;
- о результатах.

У каждого их этих объектов имеются связанные между собой атрибуты. Для клиента такие атрибуты – название, адрес и другая информация для контакта. Для анализа – название и стандартная оплата. Для каждого сотрудника – его адрес, телефон, должность, квалификация и уровень оплаты. О заказе необходимо знать, кто его заказчик, когда заказ был оформлен и какой именно анализ в нем указан. А что касается результатов анализов, то необходимо знать данные, полученные при его проведении, был ли анализ предварительным или окончательным, а также номер его заказа.

**Действие 3: точное определение таблиц**.

Теперь для каждого объекта вам необходимо точно определить таблицу, а для каждого атрибута – столбец. В табл. 5.1 показаны таблицы базы данных VetLab.

**Таблица 5.1**. Таблицы базы данных VetLab.

| Таблица | Столбцы |
|---|---|
| **CLIENT** (фирма-клиент) | **Client Name** (название фирмы-клиента) |
| | **Address 1** (адрес 1) |
| | **Address 2** (адрес 2) |
| | **City** (город) |
| | **State** (штат) |
| | **Postal Code** (почтовый код) |
| | **Phone** (телефон) |
| | **Fax** (факс) |
| | **Contact Person** (контактный представитель) |
| **TESTS** (анализы) | **Test Name** (название анализа) |
| | **Standard charge** (стандартная цена) |
| **EMPLOYEE** (сотрудник) | **Employee Name** (фамилия сотрудника) |
| | **Address 1** (адрес 1) |
| | **Address 2** (адрес 2) |
| | **City** (город) |
| | **State** (штат) |
| | **Postal Code** (почтовый код) |
| | **Home Phone** (домашний телефон) |
| | **Office Extension** (телефонный номер в офисе) |
| | **Hire Date** (дата приема на работу) |
| | **Job classification** (трудовая классификация) |
| | **Hourly/Salary/Commission** (почасовая оплата/зарплата/комиссионные) |
| **ORDERS** (заказы) | **Order Number** (номер заказа) |
| | **Client Name** (название фирмы-клиента) |
| | **Test ordered** (заказанный анализ) |
| | **Responsible Salesperson** (сотрудник, принявший заказ) |
| | **Order Date** (дата заказа) |
| **RESULTS** (результаты) | **Result Number** (номер результата) |
| | **Order Number** (номер заказа) |
| | **Result** (результат) |
| | **Date Reported** (сообщенная дата) |

| Таблица | Столбцы |
|---|---|
| | **Preliminary / Final** (предварительный/окончательный) |

Таблицы, определенные в табл. 5.1, можно создать или с помощью инструмента для быстрой разработки приложений (**Rapid Application Development**, RAD), или с помощью языка определения данных (**Data Definition Language**, DDL), входящего в состав SQL, как показано ниже.

```
CREATE TABLE CLIENT (
ClientName          CHARACTER (30),     NOT NULL,
Address1            CHARACTER (30),
Address2            CHARACTER (30),
City                CHARACTER (25),
State               CHARACTER (2),
PostalCode          CHARACTER (10),
Phone               CHARACTER (13),
Fax                 CHARACTER (13),
ContactPerson       CHARACTER (30));
CREATE TABLE TESTS (
TestName            CHARACTER (30)      NOT NULL,
StandardCharge      CHARACTER (30));
CREATE TABLE EMPLOYEE (
EmployeeName        CHARACTER (30)      NOT NULL,
Address1            CHARACTER (30),
Address2            CHARACTER (30),
City                CHARACTER (25),
State               CHARACTER (2),
PostalCode          CHARACTER (10),
HomePhone           CHARACTER (13),
OfficeExtension     CHARACTER (4),
HireDate            DATE,
JobClassification   CHARACTER (10),
HourSalComm         CHARACTER (1));
CREATE TABLE ORDERS (
OrderNumber         INTEGER             NOT NULL,
ClientName          CHARACTER (30),
TestOrdered         CHARACTER (30),
Salesperson         CHARACTER (30),
OrderDate           DATE);
CREATE TABLE RESULTS (
ResultNumber        INTEGER             NOT NULL,
OrderNumber         INTEGER
Result              CHARACTER (50),
DateReported        DATE,
PrelimFinal         CHARACTER (1));
```

Эти таблицы относятся друг к другу посредством общих атрибутов (столбцов).

- Таблица CLIENT связана с таблицей ORDERS с помощью столбца ClientName.
- Таблица TESTS связана с таблицей ORDERS с помощью столбца TestName (TestOrdered).
- Таблица EMPLOYEE связана с таблицей ORDERS с помощью столбца EmployeeName (Salesperson).
- Таблица RESULTS связана с таблицей ORDERS с помощью столбца OrderNumbe

Есть простой способ сделать таблицу неотъемлемой частью реляционной базы. Надо связать эту таблицу с помощью общего столбца как минимум с еще одной таблицей из той же базы. Отношения между таблицами показаны на рис. 5.1.

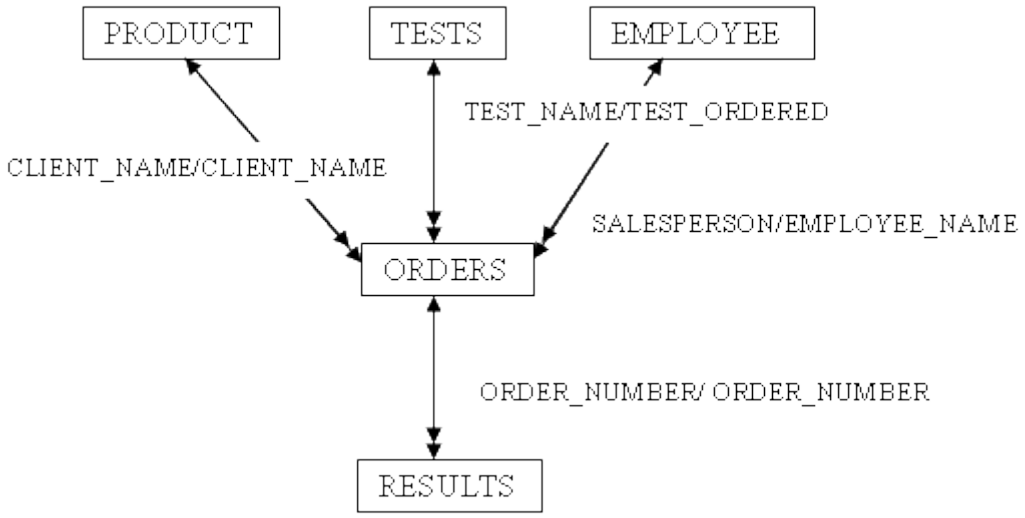


**Рис. 5.1**. Таблицы и связи базы данных VetLab

На рис. 5.1 показаны четыре различных отношения типа "один ко многим". В изображении отношения одна стрелка указывает на сторону "один", а двойная – на сторону "многие".

- Один клиент может сделать множество заказов, но каждый заказ делается одним и только одним клиентом.
- Каждый анализ может быть указан во многих заказах, но каждый заказ оформляется на один и только один анализ.
- Каждый заказ принимается одним и только одним сотрудником (т.е. представителем вашей компании), но каждый представитель может принимать (и, как вы надеетесь, принимает) множество заказов.
- В результате выполнения каждого заказа может получиться несколько предварительных результатов и один окончательный, но каждый результат относится к одному и только одному заказу.

Как видно на рисунке, атрибут, который связывает одну таблицу с другой, может в каждой из этих таблиц называться по-своему. Однако тип данных у этих атрибутов должен совпадать.

### Домены, символьные наборы, сопоставления и трансляции

Хотя главными компонентами базы данных являются таблицы, но другие ее элементы также играют определенную роль. В главе 1 **домен** табличного столбца был определен как набор всех значений, которые допустимы для этого столбца. Создание с помощью ограничений четко определенных доменов табличных столбцов – это важная часть проектирования базы данных.

Реляционными базами данных пользуются не только те, кто разговаривает на американском варианте английского языка. С этими базами можно работать, используя и другие языки, даже те, у которых другие символьные наборы. Даже если база данных создана с использованием только английского языка, некоторые приложения могут запросить специальные символьные наборы. SQL:2003 позволяет точно определить тот набор, который вам нужно использовать. Фактически можно для каждого табличного столбца использовать отдельный символьный набор. В языках, отличных от SQL, подобной гибкости обычно нет.

**Сопоставление**, или **последовательность сопоставления**, – это набор правил, которые определяют, каким образом сравниваются друг с другом строки, состоящие из элементов определенного символьного набора. Каждый символьный набор имеет свое сопоставление по умолчанию. В сопоставлении по умолчанию для символьного набора ASCII B следует после A, а C – после B. Поэтому при сравнении считается, что A меньше B, а C больше B. С другой стороны, SQL:2OO3 дает возможность применять к символьному набору и другие сопоставления. Повторяю снова, что в других языках подобной степени гибкости обычно нет.

Иногда данные в базе кодируются с помощью одного символьного набора, но работать с ними нужно с помощью другого набора. У вас, например, есть данные, закодированные в немецком символьном наборе, но те немецкие символы, которые не входят в набор ASCII, на вашем принтере не печатаются. **Трансляция** – это функциональная возможность SQL:2003, позволяющая преобразовывать символьные строки из одного набора в другой. Трансляция, например, позволяет преобразовывать один символ в два, в частности немецкий **u** – в **ue** из ASCII, или может преобразовывать символы из нижнего регистра в верхний. Можно даже преобразовать один алфавит в другой, например алфавит языка иврит в символы ASCII.

### Ускорение работы базы данных с помощью ключей

В проектировании баз данных используется хорошее правило: в таблице каждая строка отличается от любой другой, т.е. каждая строка должна быть уникальной. Иногда вам может потребоваться для определенной цели – например, для статистического анализа – извлечь из своей базы некоторые данные и создать на их основе таблицы, где строки не обязательно являются уникальными. Для данного конкретного случая правило отсутствия дублирования может не выполняться. Однако в общем его нужно строго придерживаться.

**Ключ** – это атрибут или сочетание атрибутов, которое однозначно определяет строку в таблице. Чтобы получить доступ к строке, необходимо иметь способ, позволяющий отличить эту строку от всех остальных. Так как ключи уникальны, они позволяют выполнять такой доступ. Более того, ключ никогда не должен содержать неопределенное значение. При использовании неопределенных ключей две строки, содержащие такие ключевые поля, были бы неотличимы друг от друга.

Что касается примера с ветеринарной лабораторией, то здесь в качестве ключей вы можете назначить подходящие для этого столбцы. В таблице CLIENT хороший ключ получается из столбца ClientName. Этот ключ может отличить любого клиента от всех остальных. Таким образом, ввод значения в этот столбец становится обязательным для каждой строки таблицы CLIENT. Из столбцов TestName и EmployeeName получаются хорошие ключи для таблиц TESTS и EMPLOYEE. То же относится к столбцам OrderNumber и ResultNumber таблиц ORDERS и RESULTS соответственно. В любом случае надо проверять, вводится ли в каждую строку уникальное значение ключа.

Ключи могут быть двух видов: **первичные и внешние**. Ключи, о которых говорилось в предыдущем абзаце, на самом деле являются первичными. Благодаря им гарантируется уникальность строк. О внешних ключах рассказывается в этом разделе чуть позже.

## Первичные ключи

Чтобы реализовать в базе данных VetLab идею ключей, при создании таблицы можно сразу указывать ее первичный ключ. В следующем примере будет достаточно одного столбца (при условии, что у фирм-клиентов VetLab разные названия):

| CREATE TABLE CLIENT ( | | |
|---|---|---|
| ClientName | CHARACTER (30), | PRIMARY KEY, |
| Address1 | CHARACTER (30), | |
| Address2 | CHARACTER (30), | |
| City | CHARACTER (25), | |
| State | CHARACTER (2), | |
| PostalCode | CHARACTER (10), | |
| Phone | CHARACTER (13), | |
| Fax | CHARACTER (13), | |
| ContactPerson | CHARACTER (30)); | |

Здесь ограничение NOT NULL (не может быть неопределенным значением), которое было в предыдущем определении таблицы CLIENT, заменено другим ограничением – **PRIMARY KEY** (первичный ключ). Второе ограничение подразумевает первое, потому что первичный ключ не может иметь неопределенное значение.

Несмотря на то что большинство СУБД позволяет создавать таблицу без единого ключа, важно помнить, что все таблицы базы данных должны иметь первичный ключ. Поэтому нужно ввести ограничение NOT NULL в таблицы TESTS, EMPLOYEE, ORDERS и RESULTS вместе с ограничением PRIMARY KEY, как показано в следующем примере:

```
CREATE TABLE TESTS (
TestName CHARACTER (30) PRIMARY KEY,
StandardCharge CHARACTER (30));
```

Иногда в таблице ни один единичный столбец не может гарантировать уникальность строки. В таких случаях можно использовать составной ключ. Он является сочетанием столбцов, совместное использование которых гарантирует уникальность. Представьте, что некоторые клиенты VetLab – это фирмы, имеющие свои отделения в нескольких городах. В таком случае поля ClientName будет недостаточно, чтобы различить два разных отделения одного и того же клиента. Чтобы решить эту проблему, можно определить следующий составной ключ:

| CREATE TABLE CLIENT ( | | |
|---|---|---|
| ClientName | CHARACTER (30), | PRIMARY KEY, |
| Address1 | CHARACTER (30), | |
| Address2 | CHARACTER (30), | |
| City | CHARACTER (25), | PRIMARY KEY, |
| State | CHARACTER (2), | |
| PostalCode | CHARACTER (10), | |
| Phone | CHARACTER (13), | |
| Fax | CHARACTER (13), | |
| ContactPerson | CHARACTER (30)); | |

## Внешние ключи

**Внешний ключ** – это столбец или группа столбцов в таблице, соответствующие первичному ключу (т.е. ссылающиеся на первичный ключ) из другой таблицы базы данных.

Внешний ключ сам по себе может и не быть уникальным, но должен однозначно называть столбец в той таблице, на которую он ссылается.

Если столбец ClientName – это первичный ключ таблицы CLIENT, то каждая строка этой таблицы должна иметь в столбце ClientName уникальное значение. В свою очередь, в таблице ORDERS ClientName является внешним ключом. Этот внешний ключ соответствует первичному ключу таблицы CLIENT, но в таблице ORDERS он может и не быть уникальным. На самом деле вы, конечно же, надеетесь, что внешний ключ не является уникальным. Ведь если бы каждая из фирм ваших клиентов сделала у вас только один заказ и больше этого не повторяла, то ваш бизнес довольно быстро бы прекратился. На самом деле вы надеетесь, что каждой строке таблицы CLIENT соответствует много строк таблицы ORDERS, показывая этим, что почти все ваши клиенты постоянно пользуются вашими услугами.

Следующее определение таблицы ORDERS показывает, каким образом в операторе CREATE можно задавать внешние ключи:

```
CREATE TABLE ORDERS (
OrderNumber     INTEGER     NOT NULL,
ClientName      CHARACTER (30),
TestOrdered     CHARACTER (30),
Salesperson     CHARACTER (30),
OrderDate       DATE);
CONSTRAINT BRANCHFK FOREIGN KEY (ClientName)
REFERENCES CLIENT (ClientName),
CONSTRAINT TestFK FOREIGN KEY (TestOrdered)
REFERENCES TESTS (TestName),
CONSTRAINT SalesFK FOREIGN KEY (Salesperson)
REFERENCES EMPLOYEE (EmployeeName),);
```

Внешние ключи таблицы ORDERS связывают ее с первичными ключами таблиц CLIENT, TESTS и EMPLOYEE.

## Работа с индексами

Спецификация SQL:2003 к теме индексов не обращается, но это не значит, что они являются редкой или даже необязательной частью системы баз данных. Индексы поддерживаются каждой реализацией SQL, но общего соглашения по их поддержке не существует. В главе 4 было показано, как создать индекс с помощью RAD-инструмента Microsoft Access. Чтобы разобраться, как индексы используются в конкретной системе управления базами данных, необходимо обратиться к ее документации.

### Что такое индекс

Данные в таблице обычно отображаются в том порядке, в каком их в нее первоначально ввели. Однако такой порядок может не иметь ничего общего с тем порядком, в котором затем требуется эти данные обрабатывать. Скажем, что вы, например, хотите обрабатывать таблицу CLIENT в такой последовательности, чтобы значения в столбце ClientName располагались в алфавитном порядке. Но на такую сортировку записей таблицы требуется определенное время. И чем таблица больше, тем сортировка проходит дольше. Что если в вашей таблице сто тысяч записей? Или миллион? А ведь в некоторых приложениях таблицы такого размера не являются редкостью. Даже при выполнении

лучших алгоритмов сортировки, чтобы выстроить записи таблицы в нужном порядке, все равно придется в таком случае проделать примерно двадцать миллионов сравнений и миллионы перестановок. И пусть у вас очень быстрый компьютер, но подождать все-таки придется.

Индексы могут оказаться прекрасным средством экономии времени. **Индекс** – это подчиненная таблица, или таблица поддержки, которая сопровождает свою таблицу данных.

Каждой строке таблицы данных соответствует определенная строка таблицы индекса. Однако порядок расположения строк в таблице индекса другой.

Небольшой пример таблицы данных показан в табл. 5.2.

**Таблица 5.2**. Таблица CLIENT.

| ClientName | Address1 | Address2 | City | State |
|---|---|---|---|---|
| Butternut Animal Clinic | 5 Butternut Lane | | Hudson | NH |
| Amber Veterinary, Inc. | 470 Kolvir Circle | | Amber | MI |
| Vets R Us | 2300 Geoffrey Road | Suite 230 | Anaheim | CA |
| Doggie Doctor | 32 Terry Terrace | | Nutley | NJ |
| The Equistrian Center | Veterinary Department | 7890 Paddock Parkway | Gallup | NM< |
| Dolphin Institute | 1002 Marine Drive | | Key West | FL |
| J.C.Campbell, Credit Vet | 2500 Main Street | | Los Angeles | CA |
| Wenger's Worm Farm | 15 Bait Boulevard | | Sedona | AZ |

Строки следуют в таком порядке, что значения в столбце ClientName располагаются не по алфавиту. Строки находятся в том порядке, в каком их кто-то вводил.

Индекс для этой таблицы CLIENT может выглядеть примерно так, как табл. 5.3.

**Таблица 5.3**. Индекс по названию клиента для таблицы CLIENT.

| ClientName | Указатель к таблице данных |
|---|---|
| Amber Veterinary, Inc. | 2 |
| Butternut Animal Clinic | 1 |
| Doggie Doctor | 4 |
| Dolphin Institute | 6 |
| J.C.Campbell, Credit Vet | 7 |
| The Equistrian Center | 5 |
| Vets R Us | 3 |
| Wenger's Worm Farm | 8 |

В индексе находятся поле, которое является его основой (в данном случае это поле ClientName), и указатель к таблице данных. Значение указателя, расположенное в каждой строке индекса, является номером соответствующей строки таблицы данных.

### Зачем нужен индекс

Предположим, мне нужно обрабатывать таблицу в том порядке, при котором значения из поля ClientName будут выстроены по алфавиту. При этом у меня есть индекс, в котором значения из этого поля выстроены именно по алфавиту. Тогда свою работу я смогу выполнять так же быстро, как если бы сами записи таблицы данных были выстроены в том же порядке. У меня есть возможность последовательно перемещаться по строкам индекса и

при этом немедленно переходить, используя указатель индекса, на те строки таблицы данных, которые соответствуют строкам индекса.

При использовании индекса время обработки таблицы пропорционально N, где N – количество ее строк. А при выполнении той же операции, но без индекса, время обработки таблицы пропорционально **NlgN**, где **igN** – логарифм N по основанию 2. Для небольших таблиц разница между этими значениями получается незначительная, но для больших – огромная. Некоторые операции с большими таблицами без помощи индексов требуют слишком много времени.

Например, у вас есть таблица с 1000000 записей (N = 1000000) и на обработку каждой записи уходит одна миллисекунда (одна тысячная секунды). Если у вас есть индекс, то на обработку всей таблицы уйдет только 1000 секунд, т.е. меньше 17 минут. Однако, чтобы получить тот же результат без индекса, таблицу придется обрабатывать 100000x20 раз. Таким образом, этот процесс должен занять 20000 секунд, т.е. больше пяти с половиной часов. Думаю, вы согласитесь, что разница между семнадцатью минутами и пятью с половиной часами довольно-таки существенная. Это и есть та разница, которая создается индексированием записей.

### Поддержание индекса

Если индекс создан, то его необходимо поддерживать. К счастью, вместо вас индексы поддерживает ваша система СУБД, которая обновляет их каждый раз, когда вы обновляете соответствующие таблицы данных. На этот процесс требуется немного дополнительного времени, но эти затраты того стоят. После того как вы создали индекс и СУБД его поддерживает, индекс всегда готов ускорять обработку ваших данных. И неважно, сколько раз он будет для этого использоваться.

***Совет:***

*Ясно, что лучше всего создавать индекс тогда, когда создается соответствующая ему таблица данных. Если индекс создается в самом начале и тогда же начинает поддерживаться, то вам не придется мучиться позднее с его созданием. Тогда на эту операцию придется тратить целый сеанс, длящийся довольно долго. Попробуйте предвидеть все способы, которые вам потребуются для доступа к данным, а затем для каждой такой возможности создайте свой индекс*.

В некоторых продуктах СУБД имеется возможность отключать поддержку индексов. Такое, возможно, придется делать в некоторых практических приложениях, где на обновление индексов уходит много времени, которое нужно экономить. Иногда даже приходится производить обновление индексов специальной операцией, выполняемой в промежутках между часами пик.

***Внимание:***

*Не попадитесь на удочку, создавая индексы для получения относящихся к заказам данных, которые вы вряд ли когда-нибудь будете использовать. С поддержкой индекса связана определенная потеря производительности, потому что эта поддержка является дополнительной операцией, которую компьютер должен выполнять каждый раз, когда модифицирует поле индекса или добавляет (удаляет) строку в таблице данных. Чтобы добиться оптимальной производительности, создавайте только те индексы, которые действительно собираетесь использовать в качестве ключей получения данных, и только для таблиц, имеющих большое количество строк. Ведь в противном случае индексы только ухудшат производительность*.

***Совет:***

*Вам, возможно, потребуется составить нечто похожее на месячный или квартальный отчет, где нужны данные, выстроенные в каком-либо необычном порядке. В таком случае, перед тем как генерировать отчет, создайте индекс. Затем, после того как*

*отчет будет готов, удалите этот индекс. Он не должен обременять СУБД, которой пришлось бы его поддерживать в течение долгого времени между отчетами*.

## Обеспечение целостности

База данных представляет ценность лишь тогда, когда вы в достаточной степени уверены, что находящиеся в этой базе данные правильные. Например, неправильные данные в медицинских, авиационных и космических базах данных могут привести к человеческим жертвам. Неправильные данные в других приложениях приводят не к таким плачевным последствиям, но нанести ущерб все же могут. Поэтому проектировщик базы данных должен гарантировать, что в нее неправильные данные никогда не попадут.

Некоторые неприятности нельзя остановить на уровне базы данных. Об их заблаговременном предотвращении (перед тем как они нанесут ущерб базе) должен позаботиться прикладной программист. Каждый, кто каким-либо образом отвечает за работу с базой данных, должен осознавать, какие опасности угрожают целостности ее данных, и принять все меры, чтобы свести эти опасности к нулю.

В базах данных целостность может быть нескольких видов, причем довольно-таки разных. Разными могут быть и неприятности, которые представляют угрозу целостности. В следующих разделах рассказывается о трех видах целостности: **смысловой, доменной и ссылочной**. Кроме того, рассматриваются некоторые угрозы целостности баз данных.

## Смысловая целостность

Каждая таблица базы данных соответствует какому-либо объекту реального мира. Такой объект может быть физическим или умозрительным, но его существование в некотором смысле не зависит от базы данных. Если таблица полностью соответствует объекту, который она моделирует, то она обладает **смысловой целостностью**. Чтобы иметь такую целостность, таблица должна иметь первичный ключ. Этот ключ однозначно определяет каждую строку таблицы. Если его нет, то нет и уверенности, что вы получите нужную вам строку.

Чтобы поддержать целостность объекта, необходимо для столбца или группы столбцов, из которых состоит первичный ключ, указать NOT NULL. Кроме того, для первичного ключа еще необходимо ограничение UNIQUE. В некоторых реализациях SQL такое ограничение указывается непосредственно в определении таблицы. В других же реализациях его приходится задавать уже после того, как будет указано, каким образом данные следует добавлять в таблицу, изменять и удалять из нее. Проще всего добиться, чтобы первичный ключ был и NOT NULL, и UNIQUE, – это использовать ограничение PRIMARY KEY, как показано в следующем примере:

| CREATE TABLE CLIENT ( | | |
|---|---|---|
| ClientName | CHARACTER (30), | PRIMARY KEY, |
| Address1 | CHARACTER (30), | |
| Address2 | CHARACTER (30), | |
| City | CHARACTER (25), | |
| State | CHARACTER (2), | |
| PostalCode | CHARACTER (10), | |
| Phone | CHARACTER (13), | |
| Fax | CHARACTER (13), | |

| | | |
|---|---|---|
| ContactPerson | CHARACTER (30)); | |

Альтернативой этому способу является использование ограничения NOT NULL в сочетании с UNIQUE:

| | | |
|---|---|---|
| CREATE TABLE CLIENT ( | | |
| ClientName | CHARACTER (30), | NOT NULL,, |
| Address1 | CHARACTER (30), | |
| Address2 | CHARACTER (30), | |
| City | CHARACTER (25), | |
| State | CHARACTER (2), | |
| PostalCode | CHARACTER (10), | |
| Phone | CHARACTER (13), | |
| Fax | CHARACTER (13), | |
| ContactPerson | CHARACTER (30)); | |
| UNIQUE (ClientName)); | | |

## Доменная целостность

Обычно вы не можете гарантировать, что конкретный элемент данных из базы правильно введен, но можете хотя бы определить, разрешено ли его использование. У многих элементов данных набор возможных значений является ограниченным. Если вводится значение, которое не входит в этот набор, то такой ввод должен считаться ошибочным. Например, Соединенные Штаты состоят из 50 штатов, округа Колумбия, Пуэрто-Рико и еще нескольких владений. У каждой из этих территорий имеется код, состоящий из двух символов и признанный почтовой службой США. И если в базе данных имеется столбец State (штат), то **доменную целостность** можно обеспечить, требуя, чтобы любой ввод в этот столбец был одним из разрешенных двух-символьных кодов. Если оператор вводит код, не входящий в список принятых кодов, он тем самым нарушает доменную целостность. Проверяя соблюдение доменной целостности, вы можете отказываться принимать любую нарушающую эту целостность операцию.

Опасения за доменную целостность возникают при вводе в таблицу новых данных, выполняемом с помощью оператора INSERT или UPDATE. Домен для столбца можно установить с помощью оператора **CREATE DOMAIN** (создать домен), причем до того, как использовать этот столбец в операторе CREATE TABLE. Это показано в следующем примере, где перед созданием таблицы **TEAM** (команда) со столбцами **TeamName** (имя команды) и **League** (лига) создается домен **LeagueDom** (домен значений лиги):

| | | |
|---|---|---|
| CREATE DOMAIN LeagueDom | CHAR (8) | |
| CHECK (LEAGUE IN ('American', 'National')); | | |
| CREATE TABLE TEAM ( | | |
| TeamName | CHARACTER (20) | NOT NULL, |

| | | |
|---|---|---|
| League | CHARACTER (8) | NOT NULL |
| ); | | |

Домен для столбца League состоит из двух разрешенных значений: American (американская) и National (национальная). СУБД не позволит успешно выполнить ввод или обновление в таблице TEAM, если в столбце League вводимой или обновляемой строки появляется значение, которое отличается от American или National.

### Ссылочная целостность

Даже если в базе данных для каждой таблицы установлены целостность объекта и доменная целостность, то этой базе все равно грозят неприятности, происходящие из-за того, что связь одной таблицы с другой не согласована. В большинстве хорошо спроектированных баз данных в каждой таблице имеется как минимум один столбец, который ссылается на столбец из другой таблицы той же базы. Такие ссылки играют важную роль при поддержании общей целостности базы данных. Впрочем, те же ссылки делают возможными аномалии обновления.

**Аномалии обновления** – это неприятности, которые могут происходить после обновления значений в строке базы данных.

В целом отношения между таблицами не являются равноправными. Обычно одна таблица зависит от другой. Скажем, у вас, например, имеется база данных с таблицами CLIENT (фирма-клиент) и ORDERS (заказы). Вы можете намеренно ввести в таблицу CLIENT данные фирмы-клиента еще до того, как ею будут сделаны какие-либо заказы. Однако в таблицу ORDERS нельзя будет ввести ни одного заказа, если в первой, CLIENT, не будет записи для клиента, делающего этот заказ. Получается, что таблица ORDERS зависит от таблицы CLIENT. Такой порядок часто называют **родительско-дочерним отношением таблиц**, при котором CLIENT – это родительская, а ORDERS – дочерняя таблица. Дочерний элемент базы данных зависит от родительского. Обычно первичный ключ родительской таблицы – это столбец (или группа столбцов), который имеется и в дочерней таблице. И там он уже является внешним ключом. Во внешнем ключе могут находиться неопределенные значения, и ему не нужно быть уникальным.

Аномалии обновления возникают несколькими способами. Например, фирма-клиент не делает у вас заказов, и вы хотите удалить ее данные из базы. И если она уже сделала у вас некоторые заказы, данные о которых записаны в таблице ORDERS, то удаление ее данных из таблицы CLIENT может вызвать трудности. Дело в том, что тогда в дочерней таблице ORDERS остались бы записи, для которых не было бы соответствующих записей в главной таблице CLIENT. Аналогичные трудности могут возникнуть и тогда, когда запись в дочернюю таблицу добавляется, а соответствующее добавление в родительскую еще не сделано. Все изменения первичного ключа, происходящие в любой строке родительской таблицы, должны отражаться в соответствующих внешних ключах всех дочерних таблиц. Если этого не произойдет, образуются аномалии обновления.

Большинство трудностей, связанных со ссылочной целостностью, можно свести к минимуму, если тщательно управлять процессом обновления. В некоторых случаях необходимо **каскадное** удаление из родительской таблицы и ее дочерних таблиц. Чтобы при удалении строки из родительской таблицы произошли каскадные удаления, необходимо из всех дочерних таблиц удалить строки, значение внешнего ключа которых равно значению первичного ключа строки, удаляемой из главной таблицы. Рассмотрим следующий пример:

| CREATE TABLE CLIENT ( | | |
|---|---|---|
| ClientName | CHARACTER (30), | PRIMARY KEY, |
| Address1 | CHARACTER (30), | |

| | | |
|---|---|---|
| Address2 | CHARACTER (30), | |
| City | CHARACTER (25), | NOT NULL, |
| State | CHARACTER (2), | |
| PostalCode | CHARACTER (10), | |
| Phone | CHARACTER (13), | |
| Fax | CHARACTER (13), | |
| ContactPerson | CHARACTER (30)); | |
| CREATE TABLE TESTS ( | | |
| TestName | CHARACTER (30) | PRIMARY KEY, |
| StandardCharge | CHARACTER (30)); | |
| CREATE TABLE EMPLOYEE ( | | |
| EmployeeName | CHARACTER (30) | PRIMARY KEY, |
| Address1 | CHARACTER (30), | |
| Address2 | CHARACTER (30), | |
| City | CHARACTER (25), | |
| State | CHARACTER (2), | |
| PostalCode | CHARACTER (10), | |
| HomePhone | CHARACTER (13), | |
| OfficeExtension | CHARACTER (4), | |
| HireDate | DATE, | |
| JobClassification | CHARACTER (10), | |
| HourSalComm | CHARACTER (1)); | |
| CREATE TABLE ORDERS ( | | |
| OrderNumber | INTEGER | PRIMARY KEY, |
| ClientName | CHARACTER (30), | |
| TestOrdered | CHARACTER (30), | |
| Salesperson | CHARACTER (30), | |
| OrderDate | DATE); | |
| CONSTRAINT NameFK FOREIGN KEY (ClientName) | | |
| REFERENCES CLIENT (ClientName) | | |
| ON DELETE CASCADE, | | |
| CONSTRAINT TestFK FOREIGN KEY (TestOrdered) | | |
| REFERENCES TESTS (TestName) | | |
| ON DELETE CASCADE, | | |
| CONSTRAINT SalesFK FOREIGN KEY (Salesperson) | | |
| REFERENCES EMPLOYEE (EmployeeName) | | |
| ON DELETE CASCADE); | | |

Ограничение NameFK делает поле ClientName внешним ключом, который указывает на столбец ClientName таблицы CLIENT. Когда в таблице CLIENT удаляется строка, то в таблице ORDERS автоматически удаляются все строки, у которых в столбце ClientName имеется то же значение, что и в столбце ClientName удаляемой строки таблицы CLIENT. Происходит каскадное удаление– вначале в таблице CLIENT, а затем в таблице ORDERS. То же самое верно для внешних ключей таблицы ORDERS, которые являются первичными ключами таблиц TESTS и EMPLOYEE.

Возможно, вы не хотите производить каскадное удаление, а вместо этого хотите заменить внешний ключ подчиненной таблицы значением NULL. Проанализируйте следующий вариант предыдущего примера:

```
CREATE TABLE ORDERS (
OrderNumber     INTEGER         PRIMARY KEY,
ClientName      CHARACTER (30),
TestOrdered     CHARACTER (30),
Salesperson     CHARACTER (30),
OrderDate       DATE);
CONSTRAINT NameFK FOREIGN KEY (ClientName)
REFERENCES CLIENT (ClientName),
CONSTRAINT TestFK FOREIGN KEY (TestOrdered)
REFERENCES TESTS (TestName),
CONSTRAINT SalesFK FOREIGN KEY (Salesperson)
REFERENCES EMPLOYEE (EmployeeName),);
ON DELETE SET NULL);
```

Ограничение SalesFK определяет поле Salesperson внешним ключом, который указывает на столбец EmployeeName таблицы EMPLOYEE. Если сотрудница, работавшая вашим представителем при оформлении заказов, уходит из компании, вы удаляете ее строку из таблицы EMPLOYEE. Со временем ее место займет другой работник. А сейчас удаление строки с ее данными из таблицы EMPLOYEE приводит к заменам значений. Эти замены состоят в том, что в таблице ORDERS во всех строках с заказами, оформленными этим представителем, столбцу Salesperson присваивается неопределенное значение.

Есть и другой способ предохранить базу от несогласованных данных. Он состоит в том, чтобы отказаться от разрешения добавлять строки в дочернюю таблицу, пока в родительской таблице не появится соответствующая им строка. Если вы откажетесь разрешать добавление строк в дочерней таблице, пока не будет нужной строки в родительской таблице, то таким образом предотвратите появление "строк-сирот" в дочерней таблице. Это позволит легко поддерживать согласованность таблиц. Еще одна возможность состоит в том, чтобы запретить изменять первичный ключ таблицы. В этом случае можно не беспокоиться об обновлении внешних ключей в других таблицах, которые зависят от этого первичного ключа.

### Области возможных трудностей

Покушения на целостность данных приходится ждать с самых разных сторон. Некоторые из этих неприятностей возникают только в многотабличных базах, в то время как другие могут произойти даже в базах, в которых имеется только одна таблица. Необходимо уметь распознавать эти угрозы и сводить вероятность их появления к минимуму.

### Ввод неправильных данных

В документах или файлах с исходной информацией, которыми вы пользуетесь для заполнения своей базы, могут быть неправильные данные. Они могут быть неправильным вариантом правильных данных или быть совсем не теми данными, которые вам нужны. Проверки с помощью диапазонов допустимых значений показывают, имеют ли данные доменную целостность. Хотя такие проверки и помогают преодолеть некоторые трудности, но, откровенно говоря, далеко не все. Если в полях имеются неправильные значения, не выходящие из допустимого диапазона, то в результате проверок ничего подозрительного обнаружено не будет.

### Ошибка оператора

Ваши исходные данные могут быть правильными, но оператор при вводе поймет их неправильно. Ошибка такого рода может привести к тем же трудностям, что и ввод неправильных данных. И некоторые из средств для их преодоления такие же. Проверки с помощью диапазонов допустимых значений в этом случае полезны, но не являются панацеей. Есть еще одно решение. Оно состоит в том, чтобы все вводимые данные независимо проверялись еще и вторым оператором. Такой подход обходится дорого, потому что независимая проверка удваивает количество работников и затраченное на работу время. Хотя в некоторых случаях, когда целостность данных очень важна, дополнительные усилия и затраты себя оправдывают.

### Механическое повреждение

При механическом повреждении носителя данных, на котором была открыта таблица базы данных, данные этой таблицы могут быть испорчены. Главная защита против подобной неприятности – резервное копирование.

### Злой умысел

Не следует исключать возможности умышленной порчи данных. Ваша первая линия обороны – это запрещение доступа к базе данных потенциально опасным пользователям, а также ограничение доступа всем остальным только тем, что им нужно. Вторая линия – хранение резервных копий данных в безопасном месте. Периодически проверяйте, насколько защищена ваша база данных.

### Избыточность данных

В тех базах, которые организованы иерархически, избыточность данных – это существенная проблема. Впрочем, она имеет место и в реляционных базах. Такая избыточность не только означает ненужный расход места на диске и замедление обработки, но также может привести и к серьезному повреждению данных. Если вы храните один и тот же элемент данных в двух разных таблицах базы данных, то одна его копия может быть изменена, а другая, что находится во второй таблице, может остаться прежней. Такая ситуация приводит к расхождениям, и может случиться так, что вы не сможете сказать, какой из двух вариантов является правильным. Желательно сводить избыточность данных к минимуму. Определенная избыточность необходима, чтобы первичный ключ одной таблицы служил внешним ключом другой. Однако любой другой избыточности старайтесь избегать. Самый распространенный метод уменьшения избыточности в проекте называется**нормализацией**. Он состоит в том, что одна таблица из базы данных разбивается на две или несколько простых.

Правда, после того как вы уберете большую часть избыточности из проекта базы данных, то, возможно, обнаружите, что ее производительность стала никуда не годной. Операторы очень часто используют избыточность специально для того, чтобы ускорить обработку данных. В предыдущем примере в таблице ORDERS для определения источника каждого заказа имеется только название фирмы-клиента. При оформлении заказа, чтобы получить адрес этой фирмы, вам необходимо соединить таблицу ORDERS с таблицей CLIENT. Если в результате такого соединения таблиц программа, которая распечатывает заказы, будет работать очень медленно, то вам, возможно, захочется выполнять избыточное хранение адреса фирмы-клиента, т.е. еще и в таблице ORDERS, а не только в CLIENT. Преимущество такой избыточности состоит в более быстрой распечатке заказов, но это происходит за счет замедления и усложнения любого обновления адреса клиента.

***Совет:***

*Распространенная среди пользователей практика состоит в том, чтобы вначале проектировать базу данных с малой избыточностью и с высокой степенью нормализации, а затем, обнаружив, что важные приложения работают медленно, выборочно добавлять избыточность и уменьшать нормализацию. Главное слово здесь –*

*это "выборочно". Добавляемая избыточность предназначена для определенной цели, и так как она представляет определенный риск, предпринимайте соответствующие меры к тому, чтобы избыточность вызывала проблем не больше, чем решала*.

### Превышение технических возможностей базы данных

Система базы данных может работать безукоризненно годами, а затем периодически начать выдавать ошибки, которые постепенно становятся все более серьезными. Это означает наступление одного из пределов емкости системы. Существуют пределы количества строк таблицы, а также предел столбцов, ограничений и других характеристик. Поэтому старайтесь контролировать соответствие текущего размера и содержимого вашей базы данных техническим характеристикам вашей СУБД. Если вы обнаружили, что предел какого-либо свойства уже близок, займитесь улучшением возможностей системы. Или заархивируйте старые данные, к которым вы больше не обращаетесь, а затем удалите их из базы данных.

## Ограничения

В этой главе об ограничениях говорилось как о механизме, благодаря которому в табличный столбец могут быть введены только данные из домена этого столбца. Ограничение (**constraint**) – это правило, за исполнением которого следит СУБД. После определения базы данных можно создавать определения таблиц с заданными в этих определениях ограничениями (такими, например, как NOT NULL). Благодаря СУБД вы никогда не сможете успешно выполнить никакой транзакции, если она нарушает какое-либо ограничение.

***Помни:***

*В вашем распоряжении имеются ограничения трех разных видов*.

- **Ограничение столбца** накладывает определенное условие на столбец в таблице.
- **Ограничение таблицы** – это ограничение, относящееся ко всей таблице.
- **Утверждением** является ограничение, которое может относиться к более чем одной таблице.

### Ограничения столбцов

Пример ограничения столбца показан в следующем операторе языка определения данных DDL:

| CREATE TABLE CLIENT ( | | |
|---|---|---|
| ClientName | CHARACTER (30), | NOT NULL,, |
| Address1 | CHARACTER (30), | |
| Address2 | CHARACTER (30), | |
| City | CHARACTER (25), | |
| State | CHARACTER (2), | |
| PostalCode | CHARACTER (10), | |
| Phone | CHARACTER (13), | |
| Fax | CHARACTER (13), | |
| ContactPerson | CHARACTER (30)); | |
| ); | | |

В этом операторе ограничение NOT NULL, примененное к столбцу ClientName, указывает на то, что этот столбец не может принимать неопределенное значение. Другое ограничение, которое можно применять к столбцу, – это UNIQUE. Оно указывает на то, что каждое значение, находящееся в столбце, должно быть уникальным. Ограничение CHECK (проверка) особенно полезно тем, что может принимать в качестве аргумента любое корректное выражение.

Рассмотрим следующий пример:

| CREATE TABLE TESTS ( |
|---|

| | | |
|---|---|---|
| TestName | CHARACTER (30) | NOT NULL, |
| StandardCharge | NUMBER (6.2) | |
| CHECK (StandardCharge >= 0.0 | | |
| AND StandardCharge <= 200.0) | | |
| ); | | |

В VetLab стандартная плата за проведение анализа всегда должна быть больше или равна нулю. Кроме того, ни один из стандартных анализов не стоит больше 200 долларов. Благодаря ограничению CHECK в столбец StandardCharge не попадет никакое значение, находящееся вне диапазона 0 <= STANDARD_CHARGE <= 200. А вот еще способ установить то же самое ограничение:

```
CHECK (StandardCharge BETWEEN 0.0 AND 2 00.0)
```

### Ограничения таблиц

Ограничение PRIMARY KEY указывает на то, что столбец, к которому оно применено, является первичным ключом. Таким образом, это ограничение относится ко всей таблице и эквивалентно комбинации двух ограничений столбца: NOT NULL и UNIQUE. Это ограничение, как показано в следующем примере, можно задавать в операторе CREATE:

| | | |
|---|---|---|
| CREATE TABLE CLIENT ( | | |
| ClientName | CHARACTER (30), | PRIMARY KEY, |
| Address1 | CHARACTER (30), | |
| Address2 | CHARACTER (30), | |
| City | CHARACTER (25), | |
| State | CHARACTER (2), | |
| PostalCode | CHARACTER (10), | |
| Phone | CHARACTER (13), | |
| Fax | CHARACTER (13), | |
| ContactPerson | CHARACTER (30)); | |
| ); | | |

### Утверждения

**Утверждение** (assertion) задает ограничение для более чем одной таблицы. В следующем примере для создания утверждения применяется условие поиска, составленное для столбцов из двух таблиц.

| | | |
|---|---|---|
| CREAE TABLE ORDERS ( | | |
| OrderNumber | INTEGER | NOT NULL, |
| ClientName | CHARACTER (30), | |
| TestOrdered | CHARACTER (30), | |
| Salesperson | CHARACTER (30), | |
| OrderDate | DATE | |
| ); | | |
| CREATE TABLE RESULTS ( | | |
| ResultNumber | INTEGER | NOT NULL, |
| OrderNumber | INTEGER | |
| Result | CHARACTER(50), | |
| DateReported | DATE, | |

| | |
|---|---|
| PrelimFinal | CHARACTER (1), |
| ); | |
| CREATE ASSERTION | |
| CHECK (NOT EXISTS SELECT * FROM ORDERS, RESULTS | |
| WHERE ORDERS.OrderNumber = RESULTS. OrderNumber | |
| AND ORDERS.OrderDate > RESULTS.DateReported); | |

Благодаря этому утверждению дата анализа не будет предшествовать дате заказа теста.

## Нормализация базы данных

Среди способов организации данных есть такие, которые лучше всех остальных, а есть такие, которые более логичны, некоторые – проще. Кроме того, имеются и такие, которые, как только начинается использование базы данных, лучше других предотвращают несоответствие данных.

Если правильно не организовать структуру базы данных, то эта база станет жертвой множества разных неприятностей (которые называются **аномалиями изменения**). Чтобы их предотвратить, можно **нормализовать** структуру базы данных. Нормализация обычно влечет за собой разделение в базе данных одной таблицы на две или несколько простых.

**Аномалии изменения** так называются потому, что проявляются в таблице базы данных при добавлении в нее, изменении в ней или удалении из этой таблицы данных.

Иллюстрацией того, каким образом могут проявляться аномалии изменения, является таблица, приведенная на рис. 5.2.

SALES

| Customer_ID | Product | Price |
|---|---|---|
| 1001 | Стиральный порошок | 12 |
| 1007 | Зубная паста | 3 |
| 1010 | Отбеливатель | 4 |
| 1024 | Зубная паста | 3 |

**Рис. 5.2**. Эта таблица SALES ведет к аномалиям изменения

Ваша компания продает моющие средства для дома и предметы личной гигиены, и за один и тот же товар все покупатели платят одинаково. Все данные содержатся в таблице SALES – например, о продажах стирального порошка, зубной пасты и отбеливателя. Теперь предположим, что покупатель 1001 уехал и больше ничего у вас не приобретает. И так как он не собирается больше ничего приобретать, то вам не интересно, что же он приобретал раньше. Поэтому вы хотите удалить его строку из таблицы. Однако если вы это сделаете, то не только потеряете данные о том, что покупатель 1001 приобретал стиральный порошок, но йотом, что стиральный порошок стоит 12 долларов. Такая ситуация называется **аномалией удаления**. Удаляя одни данные (о том, что покупатель 1001 приобретал стиральный порошок), вы нечаянно удалите другие (о том, что стиральный порошок стоит 12 долларов).

В той же таблице можно наблюдать и **аномалию вставки**. Скажем, вы хотите добавить к своим товарам еще и сухой дезодорант по цене 2 доллара. Но эти данные нельзя будет поместить в таблицу SALES до тех пор, пока сухой дезодорант не потребуется какому-нибудь покупателю.

Трудность с изображенной на рисунке таблицей SALES заключается в том, что эта таблица слишком универсальна. В ней есть данные и о том, что именно приобрели у вас покупатели, и о том, сколько стоят купленные товары. Эту таблицу необходимо разбить на две другие, и каждая из них будет посвящена только одной теме (рис. 5.3).

CUT_PURCH

| Customer_ID | Product |
|---|---|
| 1001 | Стиральный порошок |
| 1007 | Зубная паста |
| 1010 | Отбеливатель |
| 1024 | Зубная паста |

PROD_PRICE

| Product | Price |
|---|---|
| Стиральный порошок | 12 |
| Зубная паста | 3 |
| Отбеливатель | 4 |

**Рис. 5.3**. Таблица SALES разбита на две другие

На рис. 5.3 изображено, что таблица SALES разделена на две новые таблицы.

- Таблица CUST_PURCH (покупки) содержит данные о сделанных у вас покупках.
- Таблица PROD_PRICE (цена товара) содержит данные о ценах ваших товаров.

Вот теперь можно удалять из таблицы CUST_PURCH строку с данными о покупателе 1001, не теряя при этом других данных, – о том, что стиральный порошок стоит 12 долларов. Данные о ценах теперь хранятся в другой таблице, PROD_PRICE. И еще, данные о сухом дезодоранте можно заносить в таблицу PROD_PRICE, независимо от того, купил кто-то этот товар или нет. Дело в том, что информация о покупках хранится не в этой таблице, а в CUST_PURCH.

**Нормализацией** называется процесс разделения одной таблицы на множество других, каждая из которых посвящена отдельной теме. Нормализация, которая решает одну проблему, может не оказывать никакого влияния на другие. И чтобы в конце концов получить такие таблицы, которые посвящены каждая единственной теме, может потребоваться несколько последовательных нормализации. В базе данных у каждой таблицы должна быть одна и только одна главная тема. Ведь если таблица посвящена хотя бы двум темам, то в такой таблице иногда бывает трудно что-то понять.

Таблицы можно классифицировать по видам тех аномалий изменения, которым эти таблицы подвержены. В своей статье, выпущенной в 1970 году (первой, где была описана реляционная модель), доктор И.Ф. Кодд (E.F. Codd) диагностирует три источника аномалий изменения и для "лечения" от этих аномалий выписывает три "лекарства". Это первая, вторая и третья **нормальные формы** (1НФ, 2НФ, 3НФ). В последующие годы И.Ф. Кодд и другие специалисты открыли как другие виды аномалий, так и средства против них – новые

нормальные формы. Нормальная форма Бойса-Кодда (НФБК) (Boyce-Codd normal form, BCNF), четвертая нормальная форма (4НФ) и пятая нормальная форма (5НФ) – каждая из них обеспечивала еще более высокую защиту от аномалий изменения, чем их предшественницы. В 1981 году появилась статья, написанная Р. Фейджином (R. Fegin), где описана доменно-ключевая нормальная форма (ДКНФ) (domain/key normal form, DKNF). Эта последняя нормальная форма гарантирует отсутствие аномалий изменения.

Нормальные формы являются **вложенными** в том смысле, что таблица, находящаяся в 2НФ, автоматически находится и в ШФ. Аналогично, таблица, которая находится в 3НФ, находится ив 2НФ и т.д. Для большинства приложений приведения базы данных в 3НФ вполне достаточно, чтобы обеспечить в этой базе высокую степень целостности. Впрочем, чтобы была абсолютная уверенность в целостности базы данных, необходимо привести ее в ДКНФ.

После проведения максимально возможной нормализации своей базы данных вам для увеличения ее производительности, вероятно, захочется выполнить выборочную денормализацию. В таком случае надо полностью отдавать себе отчет, с какими аномалиями вы, возможно, столкнетесь.

### Первая нормальная форма

Чтобы быть в первой нормальной форме (1НФ), таблица должна обладать такими качествами.

- Быть двумерной, т.е. состоять из строк и столбцов.
- В каждой строке должны находиться данные, соответствующие объекту или части объекта.
- В каждом столбце должны находиться данные, относящиеся к одному из атрибутов описываемого объекта.
- В каждой табличной ячейке (пересечении строки и столбца) должно находиться только одно значение.
- В каждом столбце должны быть только однотипные данные. Если, например, в какой-либо строке в столбце находится фамилия сотрудника, то тогда и во всех остальных строках в этом столбце также должны быть фамилии сотрудников.
- У каждого столбца должно быть уникальное имя.
- Никакие две строки не могут быть одинаковыми (т.е. каждая строка должна быть уникальной).
- Порядок расположения столбцов и строк не должны иметь значения.

Таблица (отношение), находящаяся в первой нормальной форме, хотя и имеет "иммунитет" к некоторым видам аномалий изменения, но все равно подвержена остальным. Первой нормальной форме соответствует таблица SALES (см. рис. 5.2), но, как уже говорилось, эта таблица подвержена аномалиям удаления и вставки. Так что эта нормальная форма может быть полезной в одних приложениях и ненадежной в других.

### Вторая нормальная форма

Чтобы оценить вторую нормальную форму, необходимо понимать, что такое функциональная зависимость. **Функциональная зависимость** – это связь между атрибутами. Один атрибут функционально зависит от другого, если значение второго атрибута определяет значение первого. Значение первого атрибута можно определить, зная значение второго.

Предположим, например, что у таблицы имеются атрибуты (столбцы): **StandardCharge**(стандартная плата), **NumberOfTests** (число анализов) и **TotalCharge** (общая плата), которые связаны следующей формулой:

```
TotalCharge = StandardCharge * NumberOfTests
```

В таком случае столбец TotalCharge функционально зависим от двух других: Standard-Charge и NumberOfTests. Если известны значения StandardCharge и NumberOfTests, то можно определить значение TotalCharge.

Каждая таблица в первой нормальной форме должна иметь уникальный первичный ключ. Этот ключ может состоять из одного или множества столбцов. Ключ, состоящий из множества столбцов, называется **составным**. Чтобы таблица была во второй нормальной форме (2НФ), все ее неключевые атрибуты (столбцы) должны зависеть от всего ключа. Таким образом, каждое отношение в 1НФ, которое имеет ключ, состоящий из одного атрибута, автоматически находится во второй нормальной форме. Если у отношения имеется составной ключ, то все неключевые атрибуты должны зависеть от всех компонентов ключа. Пусть у вас есть таблица с неключевыми атрибутами, которые не выполняют это условие. Тогда вам, возможно, захочется разбить таблицу на не менее чем две новые, чтобы в каждой из них все неключевые атрибуты зависели от всех компонентов первичного ключа.

Звучит достаточно запутанно? Тогда для ясности рассмотрим пример. Пусть имеется таблица SALES_TRACK (данные о продажах), аналогичная таблице SALES (продажи) (см. рис. 5.2). Правда, вместо того чтобы записывать для каждого покупателя только одну покупку, вы вводите для него строку каждый раз, когда он впервые покупает какой-либо вид товара.

Кроме того, первые покупатели товара (те, у кого значения столбца CustomerID лежат в диапазоне 1001-1009) получают скидку. Некоторые строки этой таблицы приведены на рис. 5.4.

SALES_TRACK

| Customer_ID | Product | Price |
|---|---|---|
| 1001 | Стиральный порошок | 12.11 |
| 1007 | Зубная паста | 3.50 |
| 1010 | Отбеливатель | 4.00 |
| 1024 | Зубная паста | 4.33 |

**Рис. 5.4**. В таблице SALESJRACK составной ключ состоит из столбцов CustomerID и Product

На рис. 5.4 столбец CustomerID однозначно строку не определяет. В двух строках его значения равны 1001. Еще в двух равны 1010. Однако строку однозначно определяет комбинация столбцов CustomerID и Product. Вместе эти столбцы и являются составным ключом.

Если бы не то условие, что одни покупатели имеют скидку, а другие нет, то таблица не была бы во второй нормальной форме, потому что столбец Price (цена), являющийся неключевым атрибутом, зависел бы только от столбца Product (товар). Но так как часть покупателей имеет скидку, то PRICE зависит и от CustomerID, и от Product, так что таблица все же находится во второй нормальной форме.

### Третья нормальная форма

Все-таки есть аномалии изменения, против которых таблицы во второй нормальной форме беззащитны. Эти аномалии связаны с транзитивными зависимостями.

***Помни:***
*Транзитивная зависимость имеет место тогда, когда один атрибут зависит от второго, а второй, в свою очередь, от третьего. Удаления в таблице, имеющей такие зависимости, могут вызвать ненужную потерю информации. Отношение в третьей нормальной форме – это отношение во второй нормальной форме, не имеющее транзитивных зависимостей*.

Снова посмотрим на таблицу SALES (продажи) (см. рис. 5.2), которая, как вам известно, находится в первой нормальной форме. Пока для каждого значения CustomerID (идентификатор покупателя) можно вводить только одну строку, то имеется первичный ключ, состоящий из одного атрибута, поэтому таблица находится во второй нормальной форме. Однако таблица все равно подвержена аномалиям. А что если покупателю 1010, к примеру, не повезет с отбеливателем и он вернет свою покупку, получив назад деньги? Вы собираетесь удалить из таблицы третью строку, в которой записаны данные о том, что покупатель 1010 приобрел отбеливатель. Но тут возникает проблема. Если строка будет удалена, то также будут удалены данные о том, что цена отбеливателя составляет 4 доллара. Такая ситуация является примером транзитивной зависимости. Атрибут Price (цена) зависит от атрибута Product (товар), который, в свою очередь, зависит от первичного ключа CustomerID.

Проблема транзитивной зависимости решается с помощью разделения таблицы SALES на две. Две таблицы, CUST_PURCH (покупки) и PROD_PRICE (цена товара), составляют базу данных, находящуюся в третьей нормальной форме (см. рис. 5.3).

### Доменно-ключевая нормальная форма (ДКНФ)

После того как база данных оказалась в третьей нормальной форме, большинство шансов на возникновение аномалий изменения было сведено на нет. Впрочем, большинство, но не все. Для исправления этих оставшихся неполадок как раз предназначены нормальные формы, находящиеся внутри третьей. Примерами таких форм являются нормальная форма Бойса-Кодда (НФБК), четвертая нормальная форма (4НФ) и пятая нормальная форма (5НФ). Каждая форма сводит на нет угрозу какой-либо аномалии изменения, но не дает гарантии защиты от всех таких аномалий. Такую гарантию дает только доменно-ключевая нормальная форма (ДКНФ).

***Помни:***
*Отношение находится в доменно-ключевой нормальной форме (ДКНФ), если каждое ограничение в этом отношении является логическим следствием определения ключей и доменов. Ограничением в этом определении называется любое правило, которое можно проверить. Ключ – это уникальный идентификатор табличной строки, а домен – набор разрешенных значений атрибута*.

Снова посмотрим на базу данных (см. рис. 5.2), которая находится в 1НФ. Это необходимо, чтобы увидеть, каким образом привести эту базу в ДКНФ.

```
Таблица: SALES(CustomerlD,Product,Price)
Ключ: CustomerID
Ограничения: CustomerlD определяет Product
PRODUCT определяет Price
CustomerlD должен быть целым числом больше 1000
```

Как заставить работать ограничение 3 (атрибут CustomerID должен быть целым числом больше 1000)? Можно всего лишь так определить домен CustomerID, чтобы в него входило это ограничение. Таким образом, ограничение становится логическим следствием домена столбца CustomerID. Product и зависит от CustomerID, а CustomerID – это ключ, так что

трудностей с ограничением 1 не будет, поскольку оно является логическим следствием определения ключа. Однако трудность есть с ограничением 2: Price зависит от (является логическим следствием) Product, а Product не является ключом. Справиться с трудностью можно, разделив таблицу SALES на две. В одной из них в качестве ключа используется CustomerID, а в другой – Product. Такая схема приведена на рис. 5.3. База данных на этом рисунке находится не только в 3НФ, но и в ДКНФ.

***Помни:***

*Проектируйте базы данных так, чтобы они по возможности были в ДКНФ. В таком случае ключевые и доменные ограничения определяют все требуемые ограничения, и аномалии изменений исключены. А если структура базы данных спроектирована так, чтобы ее нельзя было привести в ДКНФ, то ограничения необходимо встроить в прикладную программу, которая использует базу данных. Сама база данных не дает гарантии, что ограничения будут соблюдаться*.

### Ненормальная форма

Ненормальность иногда полезна. Возможно, вы увлеклись нормализацией, и вас занесло слишком далеко. Ведь базу данных можно разбить на такое количество таблиц, что вся она станет громоздкой и неэффективной. Ее работа может застопориться. Так что часто оптимальная структура должна быть в какой-то степени денормализованной. На самом деле базы данных, используемые в практической деятельности, никогда не нормализованы до уровня ДКНФ. Впрочем, чтобы исключить возможность повреждения данных, происходящего из-за аномалий изменений, максимально нормализуйте проектируемую вами базу данных.

После этого мосты еще не сожжены. Если производительность вас не удовлетворяет, проверьте свой проект – можно ли с помощью некоторой денормализации увеличить производительность, не жертвуя при этом целостностью. Выборочно добавляя избыточность и проводя денормализацию, вы получите базу данных, которая будет и эффективной, и защищенной от аномалий.

# Манипуляции данными из базы

В этой главе…

- Работа с данными
- Получение из таблицы нужных данных
- Вывод информации, выбранной из одной или множества таблиц
- Обновление информации, находящейся в таблицах и представлениях
- Добавление новой строки в таблицу
- Изменение всех или только некоторых данных, находящихся в строке таблицы
- Удаление строки таблицы

---

В главах 3 и 4 вы узнали, что для обеспечения целостности информации, которая хранится в базе данных, очень важно создать хорошую структуру этой базы. Впрочем, для пользователя интерес представляет не структура базы данных, а ее содержимое, т.е. сами данные. С данными можно выполнять четыре следующих действия: добавлять в таблицы, получать и выводить, изменять, а также удалять из таблиц.

В принципе, манипуляции данными выполнять достаточно просто. Легко разобраться, каким образом можно добавить в таблицу одну или сразу несколько строк данных. Изменение, удаление и получение строк из таблиц баз данных также не представляют особого труда. Главная трудность манипуляций с базами данных состоит в **выборе** строк, которые требуется изменить, удалить или получить. Иногда получение данных напоминает складывание мозаики из ее фрагментов, которые перемешаны с фрагментами сотен других таких мозаик. Нужные данные обычно перемешаны с ненужными, причем последних во много раз больше. К счастью, вам требуется лишь точно указать, что именно вам нужно сделать с помощью оператора SELECT, а весь поиск выполнит компьютер.

### SQL во встроенных инструментах

Оператор **SELECT** – это не единственное средство получения данных из базы. СУБД, как правило, имеют встроенные наглядные средства для манипуляций с данными. С помощью этих средств данные можно добавлять в базу, удалять их из нее, изменять хранящиеся в ней данные, а также отправлять запросы в базу.

В системах клиент/сервер реляционной базе данных, находящейся на сервере, обычно понятен только SQL. При разработке приложения для работы с базой данных с помощью СУБД или инструмента RAD вы обычно работаете с формами, поля которых соответствуют полям таблиц, входящих в базу данных. Поля форм ввода можно группировать по

определенному принципу, а также сопровождать пояснительным текстом. Пользователь, работая на клиентской машине, может легко проверять или изменять данные в этих полях.

Допустим, что он меняет значения некоторых полей. При этом клиентская часть СУБД принимает значения, введенные пользователем с экранной формы, создает соответствующий оператор языка SQL, UPDATE, а затем отправляет этот оператор на сервер. Серверная часть СУБД выполняет этот оператор. Так что пользователи, работающие с реляционными базами данных, непосредственно или опосредованно, т.е. с помощью процесса трансляции, пользуются языком SQL.

Во многих клиентских частях СУБД имеется возможность выбора: использовать их встроенные средства или напрямую язык SQL. В некоторых случаях из СУБД нельзя "выжать" с помощью встроенных средств все то, что можно получить с помощью SQL. Так что в любом случае полезно изучить основы SQL, даже если вы большую часть времени пользуетесь встроенными средствами. Для выполнения операции, выходящей за пределы возможностей встроенных средств, необходимо понимать, каким образом работает язык SQL и что он может делать.

## Получение данных

Задача, которую выполняют пользователи, манипулируя данными чаще всего состоит в том, чтобы получить из базы выбранную информацию. Допустим, вы хотите получить содержимое одной определенной строки, находящейся в таблице среди тысяч других. Или, возможно, требуется получить все строки, удовлетворяющие какому-либо условию или комбинации условий. А может быть, вы хотите получить из таблицы все ее строки. Для решения всех этих задач предназначен оператор SQL SELECT.

Проще всего с помощью оператора SELECT получить все данные, хранящиеся во всех строках определенной таблицы. Для этого используется следующий синтаксис:

```
SELECT * FROM CUSTOMER;
```

***Помни:***

*Звездочка (*****)– это символ-маска, который означает "все". В данном примере этот символ стоит вместо перечня всех имен столбцов из таблицы CUSTOMER. В результате выполнения этого оператора на экран выводятся все данные, находящиеся во всех строках и столбцах этой таблицы*.

Операторы SELECT могут быть намного сложнее, чем тот, который приведен в примере. Некоторые из них могут быть настолько сложными, что в них становится почти невозможно разобраться. Это связанно с тем, что есть возможность к основному оператору присоединять еще и множество уточняющих предложений. Подробно об уточняющих предложениях рассказывается в главе 9. В этой же главе кратко говорится о предложении WHERE – самом распространенном способе ограничить количество строк, возвращаемых оператором SELECT.

Оператор SELECT с предложением WHERE имеет такой общий вид:

```
SELECT список_столбцов FROM имя_таблицы
WHERE условие;
```

Список столбцов указывает, какие столбцы таблицы следует отобразить при выводе. Этот оператор отобразит только те столбцы, которые вы запросите. Предложение FROM определяет имя той таблицы, столбцы которой требуется отобразить. А предложение WHERE исключает те строки, которые не удовлетворяют указанному условию. Условие может быть простым (например, WHERE CUSTOMER_STATB= 'NH', где CUSTOMER_STATE означает "штат, где проживает клиент", а NH – "штат Нью-Хэмпшир") или составным

(например, WHERE CUSTOMER_STATE= 'NH' AND STATUS='Active', где STATUS означает "статус", а Active – "активный").

Следующий пример показывает, как выглядит составное условие внутри оператора SELECT:

```
SELECT FirstName, LastName, Phone FROM CUSTOMER
WHERE State= 'NH'
AND Status='Active';
```

Этот оператор возвращает фамилии и телефонные номера всех активных клиентов, живущих в штате Нью-Хэмпшир. Ключевое слово AND означает следующее: чтобы строка была возвращена, она должна соответствовать сразу двум условиям, State= 'NH' и Status='Active'.

### Создание представлений

Структура базы данных, спроектированной в соответствии с разумными принципами – включая и подходящую нормализацию, – обеспечивает максимальную целостность данных. Однако такая структура часто не позволяет обеспечить лучший способ их просмотра. Одни и те же данные могут использоваться разными приложениями, и у каждого из них может быть своя специализация. Одним из самых сильных качеств SQL является возможность выводить данные в виде представлений, чья структура отличается от структуры тех таблиц базы, в которых реально хранятся эти данные. Таблицы, столбцы и строки которых используются при создании представления, называются **базовыми**. В главе 3 говорилось о представлениях как о части языка определения данных (Data Definition Language, DDL). А в этом разделе представления рассматриваются как одно из средств, предназначенных для получения данных и манипуляции ими.

Оператор SELECT всегда возвращает результат в виде виртуальной таблицы.**Представление** же принадлежит к особой разновидности этих таблиц. От остальных виртуальных таблиц оно отличается тем, что в метаданных базы хранится его определение. Представление других виртуальных таблиц похоже на реальную таблицу базы данных. Работать с представлением можно так же, как и с настоящей таблицей базы данных. Разница здесь в том, что совокупность данных, находящихся в представлении, не является физически независимой частью базы данных. Представление извлекает их из одной или множества таблиц, из которых и были выбраны его столбцы. В каждом приложении могут быть свои собственные, непохожие друг на друга представления, но созданные на основе одних и тех же данных.

Обратите внимание на базу данных VetLab (см. главу 5). Эта база состоит из пяти таблиц: CLIENT (фирма-клиент), TESTS (анализы), EMPLOYEE (сотрудник), ORDERS (заказы) и RESULTS (результаты). Предположим, что главному менеджеру по маркетингу компании VetLab необходимо посмотреть, из каких штатов в эту компанию приходят заказы. Часть этой информации находится в таблице CLIENT, а часть – в ORDERS. А чиновнику из службы контроля качества требуется сравнить дату оформления заказа на один из анализов и дату получения его окончательного результата. Для этого сравнения требуются некоторые данные из таблицы ORDERS и RESULTS. В каждом конкретном случае можно создать представления, предоставляющие в точности те данные, которые требуются.

### Создание представлений из таблиц

Для менеджера по маркетингу можно создать представление ORDERS_BY_STATE (заказы по штатам), приведенное на рис. 6.1.

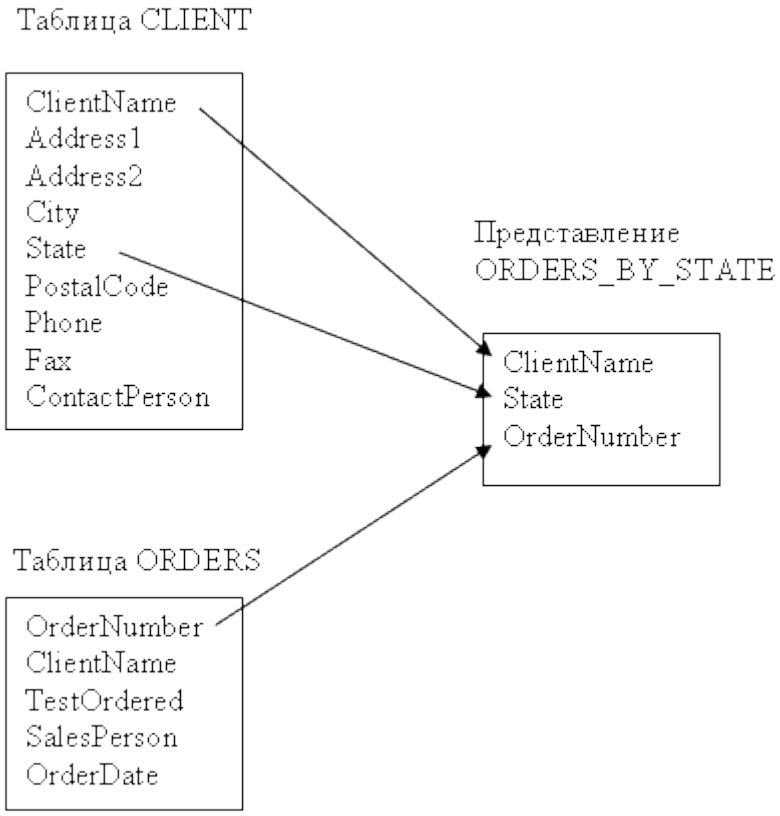


**Рис. 6.1**. Представление ORDERS_BY_STATE, предназначенное для менеджера по маркетингу

Представление для менеджера по маркетингу создается с помощью следующего оператора:

```
CREATE VIEW ORDERS_BY_STATE
(ClientName, State, OrderNumber)
AS SELECT CLIENT. ClientName, State, OrderNumber
FROM CLIENT, ORDERS
WHERE CLIENT.ClientName = ORDERS.ClientName;
```

В новом представлении имеются три столбца: **ClientName** (название фирмы-клиента),**State** (штат) и **OrderNumber** (номер заказа). ClientName находится как в CLIENT, так и в ORDERS и используется для связи между этими двумя таблицами. Новое представление получает информацию из столбца State таблицы CLIENT и берет для каждого заказа значение из столбца OrderNumber таблицы ORDERS. В приведенном примере имена столбцов нового представления объявляются явно. Впрочем, если имена точно такие же, как и у соответствующих столбцов исходных таблиц, то такое объявление не обязательно. Пример, приведенный в следующем разделе, демонстрирует похожий оператор CREATE VIEW, в котором имена столбцов для представления явно не указываются, а только подразумеваются.

### Создание представления с условием выборки

Как видно в примере на рис. 6.2, для чиновника из службы контроля качества требуется представление, отличающееся от того, которое использует менеджер по маркетингу.

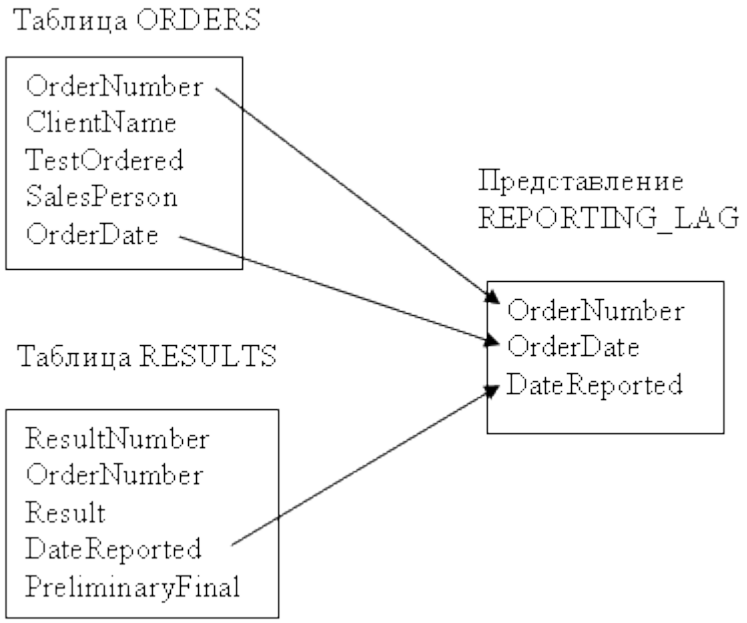


**Рис. 6.2**. Представление REPORTINGJAG (задержка результатов), предназначенное для чиновника из службы контроля качества

Ниже приведен код, с помощью которого создается представление, приведенное на рис. 6.2.

```
CREATE VIEW REPORTING_LAG
AS SELECT ORDERS.OrderNumber, OrderDate, DateReported
FROM ORDERS, RESULTS
WHERE ORDERS. OrderNumber = RESULTS. OrderNumber
AND RESULTS.PreliminaryFinal = 'F';
```

В представлении REPORTTNG_LAG содержится информация из таблицы ORDERS по датам заказов и из таблицы RESULTS по датам окончательных результатов. В этом представлении появляются только строки, у которых в столбце PRELIMJFINAL (предварительный-окончательный), взятом из таблицы RESULTS, находится значение 'F (от слова "final" – окончательный).

### Создание представления с модифицированным атрибутом

В примерах из двух предыдущих разделов предложения SELECT содержат только имена столбцов. Впрочем, в любом предложении SELECT может находиться не только имя, но и выражение. Предположим, что владелец VetLab отмечает свой день рождения и хочет в честь этого события предоставить всем своим клиентам 10-процентную скидку. Он может на основе двух таблиц, ORDERS и TESTS, создать представление BIRTHDAY (день рождения).

Вполне возможно, что оно будет создано так, как показано в следующем примере:

```
CREATE VIEW BIRTHDAY
(ClientName, Test, OrderDate, BirthdayCharge)
AS SELECT ClientName, TestOrdered, OrderDate,
StandardCharge *.9
FROM ORDERS, TESTS
WHERE TestOrdered = TestName;
```

Обратите внимание, что в представлении BIRTHDAY второй столбец – Test (анализ) – соответствует столбцу TestOrdered (заказанный анализ) из таблицы ORDERS, который также соответствует столбцу TestName (название анализа) из таблицы TESTS. Как создать это представление, можно увидеть на рис. 6.3.

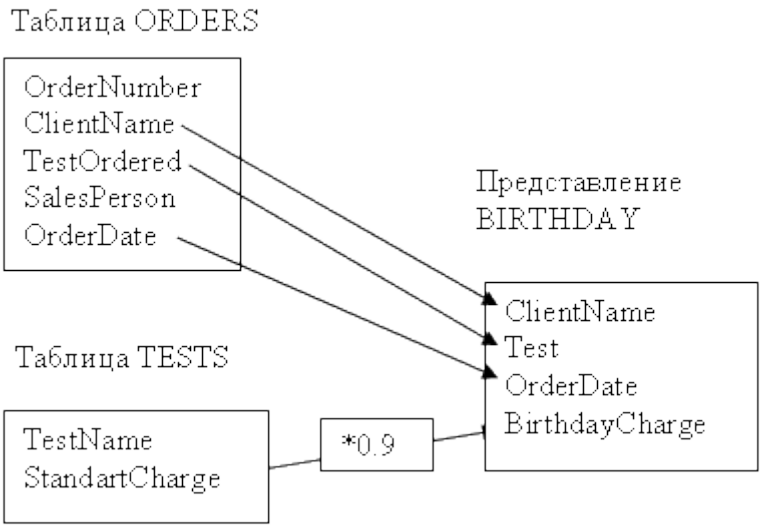


**Рис. 6.3**. Представление, созданное, чтобы показать скидки в честь дня рождения

Представления можно создавать как на основе множества таблиц, что и делалось в предыдущих примерах, так и на основе всего лишь одной таблицы. Если вам не нужны определенные столбцы и строки какой-либо таблицы, то создайте представление, в котором нет этих строк и столбцов, а затем работайте уже не с таблицей, а с представлением. Этот подход защищает пользователя от путаницы и не отвлекает внимание, которые бывают при просмотре участков таблицы, не относящихся к делу.

***Совет:***

*Еще одной причиной создания представлений является обеспечение безопасности исходных таблиц базы данных. Одни столбцы ваших таблиц следует открыть, а другие, наоборот, скрыть. В таком случае можно создать представление только с теми столбцами, которые вы хотите сделать доступными, затем предоставить к нему широкий доступ, а доступ к исходным таблицам этого представления– ограничить. В главе 13 объясняется, каким образом обеспечивается безопасность баз данных, и описывается, как предоставлять полномочия на доступ к данным и лишать их*.

### Обновление представлений

Созданные таблицы автоматически поддерживают возможности вставки, обновления И удаления данных. А вот к представлениям это относится не всегда. Обновляя представление, вы на самом деле обновляете исходную таблицу. Вот две потенциальные проблемы, возникающие при обновлении представлений.

- Некоторые представления берут свои данные не менее чем из двух таблиц, Если вы обновляете только представление, то как же узнать, какая из исходных таблиц была обновлена вместе с ним?
- В определении представления могут находиться выражения, относящиеся к предложению SELECT. Тогда каким образом это выражение можно обновлять?

Предположим, что вы создаете представление COMP из таблицы EMPLOYEE (сотрудник), используя ее поля EmpName (фамилия), SALARY (оклад) и Comm (комиссионные). Для этого вы используете следующий оператор:

```
CREATE VIEW COMP AS
SELECT EmpName, Salary+Comm AS Pay
FROM EMPLOYEE;
```

Можно ли в представлении обновить столбец PAY (оплата), используя следующий оператор?

```
UPDATE COMP SET Pay = Pay + 100;
```

Нет, этот подход не сработает потому, что в таблице EMPLOYEE нет столбца Pay, и в ней он обновлен не будет, а следовательно, не будет обновлен и в представлении. Представление не может показать того, чего нет в исходной таблице.

***Помни:***
*Когда вы думаете об обновлении представлений, не забывайте следующее правило. Столбец представления обновлять нельзя, если он не соответствует столбцу таблицы этого представления*.

**Добавление новых данных**

В базе данных каждая таблица появляется на свет пустой, т.е. сразу после своего создания (или с помощью DDL из SQL, или с помощью RAD-инструмента) такая таблица является не чем иным, как структурной оболочкой, не содержащей данных. Чтобы таблица стала полезной, в нее необходимо поместить некоторые данные. Эти данные могут быть уже в цифровом виде, т.е. введенными в компьютер, или еще нет.

- Если данные еще не в цифровом виде, то кто-то должен будет вводить их построчно с помощью клавиатуры и монитора. Кроме того, данные можно вводить и с помощью оптических сканеров и систем распознавания речи. Правда, для ввода данных такие устройства используются пока что достаточно редко.
- Если данные уже находятся в цифровом виде, но, возможно, не в формате, используемом в таблицах базы данных, тогда данные прежде всего надо преобразовать в соответствующий формат, а затем вставить их в базу.
- Даже если данные находятся уже в цифровом виде и в нужном формате, то может возникнуть необходимость перенести их в новую базу данных.

В зависимости от того состояния, в котором находятся данные, их или сразу можно перевести в базу с помощью одной операции, или, возможно, потребуется вводить в виде отдельных записей одну за другой. Каждая такая запись будет соответствовать одной строке из какой-либо таблицы базы данных.

**Добавление данных в виде отдельных записей**

В большинстве СУБД поддерживается ввод данных с помощью форм. Такая возможность позволяет создать экранную форму, в которой для каждого табличного столбца из базы данных имеется свое поле. Находящиеся в форме метки полей позволяют легко определить, какие данные следует вводить в каждое поле. Оператор вручную вводит в форму все данные, предназначенные для одной строки. После того как СУБД примет новую строку, она очищает форму, чтобы в нее можно было вводить данные для следующей строки. Таким образом можно легко добавить в таблицу данные в виде отдельных строк одну за другой.

Ввод данных с помощью форм легко использовать, кроме того, при таком вводе допускается меньше ошибок, чем при вводе списков значений, отделяемых друг от друга запятыми. Основная трудность ввода данных с помощью форм состоит в том, что он не является стандартным. В различных системах СУБД имеются собственные методы создания форм. Для оператора, занятого вводом данных, не имеет значения, каким образом создана форма ввода. Можно сделать та, что форма в любой СУБД будет выглядеть в основном одинаково. А вот разработчику приложений при переходе к новым инструментам разработки приходится учиться с самого начала. При вводе данных с помощью форм встречается и другая трудность. Она состоит в том, что в некоторых реализациях нельзя выполнять полную проверку правильности вводимых данных.

Самый лучший способ поддерживать на высоком уровне целостность данных в базе – это не вводить неправильные данные. Предотвратить неправильный ввод можно, применяя

ограничения к полям формы ввода. Это гарантирует, что в базу данных попадут только те значения, которые имеют правильный тип данных и являются частью заранее определенного диапазона. Конечно, с помощью такого ограничения нельзя предотвратить все возможные ошибки, но "отловить" некоторые все же можно.

Если в таблицу из базы данных вводятся значения, предназначенные для отдельной строки, то в команде INSERT используется следующий синтаксис:

```
INSERT INTO таблица_1 [(столбец_1, столбвц_2,…, столбец_n)]
VALUES (значение_1, значение_2,…, значение_n);
```

***Совет:***

*Бывает так, что средства проектирования форм вашей СУБД не могут осуществить все нужные проверки, гарантирующие целостность данных. В таком случае вам придется создавать собственные процедуры ввода, в которых вводимые значения присваиваются переменным и проверяются с помощью кода самой прикладной программы. Затем, убедившись, что все значения, введенные для табличной строки, являются правильными, программа может добавить эту строку в таблицу с помощью команды* ***SQL INSERT*** *(вставить)*.

Квадратные скобки ([]) означают, что список имен столбцов не является обязательным. По умолчанию порядок расположения столбцов в списке является таким же, как и в таблице. Если расположить значения, находящиеся после ключевого слова **VALUES** (значения), в том же порядке, в каком столбцы находятся в таблице, то эти элементы попадут в нужные столбцы – неважно, указаны при этом столбцы явно или нет. А если требуется расположить эти значения в порядке, который не совпадает с расположением столбцов в таблице, то тогда имена столбцов необходимо перечислить в требуемом порядке.

Чтобы ввести, например, запись в таблицу CUSTOMER (покупатель), используйте следующий синтаксис:

```
INSERT INTO CUSTOMER (CustomerlD, FirstName, LastName,
Street, City, State, Zipcode, Phone)
VALUES (:vcustid, 'David1, 'Taylor', '235 Nutley Ave.',
'Nutley', 'NJ', '07110', '(201) 555-1963');
```

После ключевого слова VALUES первым стоит vcustid – базовая переменная-счетчик, значение которой с помощью программного кода увеличивается на единицу, как только в таблицу введена новая строка. Это дает гарантию, что не будет дублирования значений в столбце **CustomerlD** (идентификатор покупателя). CustomerlD является первичным ключом для этой таблицы и поэтому должен оставаться уникальным. Остальные значения в операторе являются не переменными с элементами данных, а самими элементами данных. Конечно, элементы данных для этих столбцов также можно, если хотите, поместить в переменные. Оператор INSERT работает одинаково хорошо с аргументами ключевого слова VALUES, выраженными как в форме переменных, так и в форме значений.

### Добавление данных только в выбранные столбцы

Иногда нужно где-то отметить, что объект существует, даже если по нему еще нет всех данных. Если у вас для таких объектов есть таблица базы данных, то строку по новому объекту можно вставить в нее, не заполняя значениями все столбцы этой строки. И если вы хотите, чтобы таблица была в **первой нормальной форме**, то необходимо вставить столько данных, чтобы можно было отличать новую строку от всех остальных строк этой таблицы. (О**первой нормальной форме** см. в главе 5.) Для этого в новой строке достаточно вставить первичный ключ. Кроме этого ключа вставляйте все остальные данные, которые известны об этом объекте. В тех столбцах, куда данные не вводятся, остаются значения NULL.

Ниже приведен пример такого частичного ввода строки.

```
INSERT INTO CUSTOMER (CustomerlD, FirstName, LastName)
VALUES (:vcustid, 'Tyson1, 'Tylor');
```

Вы вставляете только уникальный идентификационный номер клиента, а также его имя и фамилию. А в остальных столбцах этой строки будут находиться значения NULL.

### Добавление в таблицу группы строк

Добавлять в таблицу строки одну за другой, используя для этого оператор INSERT, становится ужасно скучным – особенно если это растягивается на целый день. Даже вводить данные в экранную форму, тщательно продуманную с точки зрения эргономики, через некоторое время становится утомительным. Ясно, что если у вас есть надежный способ автоматического ввода данных, вы будете стараться использовать его вместо ввода вручную везде, где это возможно.

Автоматический ввод, например, можно использовать тогда, когда данные уже представлены в электронном виде, благодаря тому, что кто-то уже ввел вручную эти данные в компьютер. Незачем повторять эту рутинную работу. Перенести данные из одного файла в другой можно с минимальным участием человека. Если вам известны характеристики исходных данных и нужная форма таблицы, в которую должны быть перенесены данные, то компьютер может в принципе выполнить такой перенос данных автоматически.

### Копирование из внешнего файла данных

Предположим, что вы создаете базу данных для нового приложения. Некоторые из нужных вам данных уже имеются в каком-либо файле. Это может быть плоский файл или таблица базы данных, работающей в СУБД, отличающейся от той, которую используете вы. Данные могут быть в коде ASCII, EBCDIC или в каком-нибудь другом закрытом внутреннем формате. Так что же делать?

Прежде всего – это надеяться и молиться, чтобы нужные вам данные были представлены в каком-нибудь широко используемом формате. Если это достаточно популярный формат, то у вас имеется хороший шанс достать утилиту преобразования формата, которая может преобразовать данные в один или несколько других популярных форматов. А затем как минимум один из этих форматов можно импортировать в вашу среду разработки. А если повезет, то можно будет преобразовать текущий формат данных с помощью встроенных средств среды. Вероятно, самыми распространенными на персональных компьютерах форматами являются Access, dBASE и Paradox. Если нужные вам данные находятся в одном из этих форматов, преобразование должно пройти легко. Ну а если формат данных является не таким распространенным, то, видимо, преобразование все-таки придется провести в два этапа.

И с последней надеждой можно всегда обратиться к специальным профессиональным службам преобразования данных. Они специализируются на оказании услуг по преобразованию компьютерных данных из одного формата в другой. У них есть возможность обрабатывать сотни разных форматов – о большинстве из которых никто никогда и не слышал. Передайте одной из этих служб ленту или диск с данными в первоначальном формате, и вы получите назад те желанные, но преобразованные в любой из форматов, который только укажете.

### Перенос всех строк из одной таблицы в другую

Много проще, чем импортировать внешние данные, извлекать данные, уже находящиеся водной из таблиц вашей базы, и комбинировать их с данными из другой таблицы. В самом простом случае структура второй таблицы идентична структуре первой. Это означает, что каждый столбец первой таблицы имеет соответствующий столбец во второй, а типы данных соответствующих столбцов совпадают. В таком случае содержимое двух таблиц можно комбинировать с помощью реляционного оператора UNION (объединение). В результате

получается виртуальная таблица, в которой содержатся данные исходных таблиц. О реляционных операторах, в том числе о UNION, рассказывается в главе 10.

### Перенос выбранных столбцов и строк из одной таблицы в другую

Часто бывает так, что данные исходной таблицы не соответствуют в точности структуре той таблицы, в которую вы собираетесь их поместить. Возможно, соответствуют друг другу только некоторые из столбцов – и это как раз те столбцы, которые вы хотите перенести. Комбинируя операторы SELECT с помощью оператора UNION, можно указать, какие столбцы из исходных таблиц должны войти в полученную в результате виртуальную таблицу. Используя в операторах SELECT предложения WHERE, можно помещать в виртуальную таблицу только те строки, которые удовлетворяют определенным условиям. Предложения WHERE достаточно подробно описываются в главе 9.

Предположим, у вас имеются две таблицы, PROSPECT (потенциальный клиент) и CUSTOMER (покупатель), и вам нужно составить список всех жителей штата Мэн, данные о которых находятся в обеих таблицах. Тогда можете создать виртуальную таблицу с нужной информацией, используя следующую команду:

```
SELECT FirstName, LastName
FROM PROSPECT
WHERE State = 'ME'
UNION
SELECT FirstName, LastName
FROM CUSTOMER
WHERE State = 'ME'
```

В этом коде заключено следующее:

- Операторы SELECT говорят о том, что у созданной таблицы будут столбцы FirstName (имя) и LastName (фамилия).
- Предложения WHERE ограничивают количество строк в этой таблице, выбирая лишь те, у которых в столбце State (штат) находится значение 'ME' (штат Мэн).
- Столбца State в созданной таблице не будет, но он находится в двух исходных таблицах: в PROSPECT и CUSTOMER.
- Оператор UNION объединяет результаты, полученные при выполнении SELECT, отдельно с PROSPECT и отдельно с CUSTOMER, удаляет все дублированные строки, а затем выводит окончательный результат на экран.

Другой способ копировать данные в базе из одной ее таблицы в другую состоит в том, чтобы разместить оператор SELECT в операторе INSERT. Такой метод (подвыборка) виртуальной таблицы не создает, а просто дублирует выбранные данные. Например, вы можете взять все строки из таблицы CUSTOMER и вставить их в таблицу PROSPECT. Конечно, эта операция удастся только в том случае, если у обеих этих таблиц одинаковая структура. Далее, если нужно отобрать только тех покупателей, которые живут в штате Мэн, то достаточно простого оператора SELECT, имеющего в предложении WHERE всего лишь одно условие. Соответствующий код показан в следующем примере:

```
INSERT INTO PROSPECT
SELECT * FROM CUSTOMER
WHERE State = 'ME';
```

***Внимание:***

*Даже если эта операция и создает избыточные данные – данные о покупателях теперь хранятся в обеих таблицах, в PROSPECT и CUSTOMER, – но зато увеличивается производительность выборок. Чтобы избежать избыточности и поддерживать согласованность данных, делайте так, чтобы строки в одной таблице не вставлялись, не изменялись и не удалялись без вставки, изменения и удаления соответствующих строк в другой таблице. Может возникнуть еще одна проблема. Возможно, что оператор INCERT продублирует первичные ключи. Если существует один-единственный потенциальный клиент, имеющий ключ ProspectID, который совпадает с*

*соответствующим первичным ключом CustomerID покупателя, введенного в таблицу PROSPECT, тогда операция вставки будет неудачной*.

### Обновление имеющихся данных

Все течет, все изменяется. Если вам не нравится нынешнее положение дел, то надо немного подождать. Через некоторое время существующее положение изменится.

И так как мир постоянно преобразуется, то надо также обновлять и базы данных, с помощью которых моделируются его компоненты. Покупатель, возможно, поменял адрес. Может измениться количество какого-либо товара на складе, будем надеяться не в результате воровства, а потому что товар хорошо расходится. Это типичные примеры тех событий, из-за которых приходится обновлять базу данных.

В языке SQL для изменения данных, хранящихся в таблице, имеется оператор **UPDATE**(обновить). С помощью одного такого оператора можно изменить в таблице одну строку, несколько или все ее строки. В операторе UPDATE используется следующий синтаксис:

```
UPDATE имя_таблицы
SET столбец_1 – выражение_1, столбец_2 = выражение_2,
…, столбец_n = выражение__n
[WHERE предикаты];
```

Предложение WHERE (где) не является обязательным. Оно указывает, какие строки должны обновляться. Если это предложение не используется, то будут обновляться все строки таблицы. В свою очередь предложение SET (установить) указывает новые значения изменяемых столбцов.

Проанализируйте с помощью табл. 6.1 таблицу CUSTOMER (покупатель), имеющую столбцы Name (имя и фамилия), City (город), Area-Code (телефонный код региона) и Telephone (телефон).

**Таблица 6.1**. Таблица CUSTOMER.

| Name | City | AreaCode | Telephone |
|---|---|---|---|
| Abe Abelson | Springfield | (714) | 555-1111 |
| Bill Bailey | Decatur | (714) | 555-2222 |
| Chuck Wood | Philo | (714) | 555-3333 |
| Don Stetson | Philo | (714) | 555-4444 |
| Dolph Stetson | Philo | (714) | 555-5555 |

Время от времени списки покупателей изменяются, по мере того как эти люди переезжают, изменяются номера телефонов и т.д. Предположим, что Эйб Эйбелсон (Abe Abelson) переехал из Спрингфилда в Канкаки. Тогда запись этого покупателя, находящуюся в таблице CUSTOMER, можно обновить с помощью следующего оператора UPDATE:

```
UPDATE CUSTOMER
SET City = 'Kankakee1, Telephone = '666-6666'
WHERE Name = 'Abe Abelson1;
```

В результате его выполнения в записи произошли изменения, которые показаны в табл. 6.2.

**Таблица 6.2**. Таблица CUSTOMER после обновления одной строки оператором update.

| Name | City | AreaCode | Telephone |
|---|---|---|---|
| Abe Abelson | Kankakee | (714) | 666-6666 |

| Name | City | AreaCode | Telephone |
|---|---|---|---|
| Bill Bailey | Decatur | (714) | 555-2222 |
| Chuck Wood | Philo | (714) | 555-3333 |
| Don Stetson | Philo | (714) | 555-4444 |
| Dolph Stetson | Philo | (714) | 555-5555 |

Похожий оператор можно использовать, чтобы обновить сразу множество строк. Предположим, что город Файло переживает резкий рост населения и ему теперь требуется собственный телефонный код региона. Все строки покупателей, проживающих в этом городе, можно сразу изменить с помощью одного оператора UPDATE:

```
UPDATE CUSTOMER
SET AreaCode = '(619)'
WHERE City = 'Philo';
```

Теперь таблица CUSTOMER выглядит так, как показано в табл. 6.3.

**Таблица 6.3**. Таблица CUSTOMER после обновления нескольких строк оператором update.

| Name | City | Area-Code | Telephone |
|---|---|---|---|
| Abe Abelson | Kankakee | (714) | 666-6666 |
| Bill Bailey | Decatur | (714) | 555-2222 |
| Chuck Wood | Philo | (619) | 555-3333 |
| Don Stetson | Philo | (619) | 555-4444 |
| Dolph Stetson | Philo | (619) | 555-5555 |

Обновить в таблице все строки даже легче, чем только некоторые из них. Ведь в таком случае не надо использовать ограничивающее предложение WHERE. Представьте, что город Рантул значительно увеличился в размерах и в его состав вошли не только Канкаки, Декейтер и Файло, но и все остальные города и городки, упомянутые в базе данных. Тогда все строки можно сразу изменить с помощью одного оператора:

```
UPDATE CUSTOMER
SET City = 'Rantoul';
```

Результат показан в табл. 6.4.

**Таблица 6.4**. Таблица CUSTOMER после обновления всех строк оператором update.

| Name | City | Area-Code | Telephone |
|---|---|---|---|
| Abe Abelson | Rantoul | (714) | 666-6666 |
| Bill Bailey | Rantoul | (714) | 555-2222 |
| Chuck Wood | Rantoul | (619) | 555-3333 |
| Don Stetson | Rantoul | (619) | 555-4444 |
| Dolph Stetson | Rantoul | (619) | 555-5555 |

В предложении WHERE, используемом для ограничения тех строк, к которым применяется оператор UPDATE, может находиться **подвыборка**. Подвыборка дает возможность обновлять строки одной таблицы на основе содержимого другой.

Предположим, что вы оптовый продавец и в вашей базе данных находится таблица VENDOR (поставщик) с названиями всех фирм-производителей, у которых вы покупаете товары. У вас также есть таблица PRODUCT (товар) с названиями всех продаваемых вами товаров и ценами, которые вы за них назначаете. В таблице VENDOR имеются столбцы VendorID (идентификатор поставщика), VendorName (название поставщика), Street (улица),

City (город), State (штаг) и Zip (почтовый код). А в таблице PRODUCT имеются столбцы ProductID (идентификатор товара), ProductName (название товара), VendorlD (идентификатор поставщика) и SalePrice (цена при продаже).

Предположим, поставщик Cumulonimbus Corporation принял решение поднять цены на все виды товаров на 10%. И для того чтобы поддержать планку своей прибыли, вам также придется поднять на 10% цены продажи продуктов, получаемых от этого поставщика. Это можно сделать с помощью следующего оператора UPDATE:

```
UPDATE PRODUCT
SET SalePrice = (SalePrice * 1.1)
WHERE VendorlD IN
(SELECT VendorlD FROM VENDOR
WHERE VendorName = 'Cumulonimbus Corporation');
```

Подстрока находит то значение из столбца VendorlD, которое соответствует Cumulonimbus Corporation. Затем полученное значение можно использовать для поиска в таблице PRODUCT тех строк, которые следует обновить. Цены всех товаров, полученных от Cumulonimbus Corporation, повышаются на 10%, а цены остальных остаются прежними. О подвыборках более подробно рассказывается в главе 11.

### Перемещение данных

Помимо команд INSERT и UPDATE, можно воспользоваться командой **MERGE** (слияние), чтобы добавить данные в таблицу или представление. Команда MERGE позволяет производить "слияние" данных исходных таблиц, представления – в нужные таблицы или сами представления. Эта же команда позволяет вставить новые строки в нужную таблицу или обновить существующие строки. Таким образом, команда MERGE представляет собой весьма удобный способ копирования уже существующих данных из одного местоположения в новое, необходимое пользователю.

Возьмем, к примеру, базу данных VetLab (см. главу 5). Предположим, что некоторые работники, занесенные в таблицу EMPLOYEE, – это продавцы, которые уже приняли заказы, а другие – это работники, не связанные напрямую с продажами, или продавцы, которые еще не взяли заказы. Только что закончившийся год был прибыльным, поэтому вы решили дать премии по 100 долларов каждому, кто принял по крайней мере один заказ, и по 50 долларов всем остальным. Для начала давайте создадим таблицу BONUS (бонус) и вставим в нее записи для каждого работника, который появляется хотя бы однажды в таблице ORDERS, задавая каждой записи значение премии по умолчанию 100 долларов.

Затем воспользуемся командой MERGE, чтобы вставить новые записи для тех работников, которые не имеют заказов, давая им премии 50 долларов. Ниже приведен программный код, который позволяет создать и заполнить таблицу BONUS.

| CREARE TABLE BONUS ( | | |
|---|---|---|
| EmployeeName | CHARACTER (30) | PRIMARY KEY |
| Bonus | NUMERIC | DEFAULT 100); |

```
INSERT INTO BONUS (EmployeeName)
(SELECT EmployeeName FROM EMPLOYEE, ORDERS
WHERE EMPLOYEE.EmployeeName = ORDERS.Salesperson
GROUP BY EMPLOYEE.EmployeeName);
```

Теперь сделаем запрос для таблицы BONUS и посмотрим, что она содержит.

| SELECT * FROM BONUS; | |
|---|---|
| EmployeeName | BONUS |
| ---------- | ------- |
| Brynna Jones | 100 |
| Chris Bancroft | 100 |
| Greg Bosser | 100 |
| Kyle Weeks | 100 |

Затем выполним команду MERGE, чтобы назначить премии по 50 долларов для всех остальных работников.

```
MERGE INTO BONUS
USING EMPLOYEE
ON (BONUS.EmployeeName = EMPLOYEE.EmployeeName)
WHEN NOT MATCHED THEN INSERT
(BONUS.EmployeeName, BONUS,bonus)
VALUES (EMPLOYEE.EmployeeName, 50);
```

Записи для людей в таблице EMPLOYEE, которые не соответствуют записям для тех же людей, но уже в таблице BONUS, будут вставлены в таблицу BONUS. Теперь запрос таблицы BONUS дает следующее:

| SELECT * FROM BONUS; | |
|---|---|
| EmployeeName | BONUS |
| ---------- | ------- |
| Brynna Jones | 100 |
| Chris Bancroft | 100 |
| Greg Bosser | 100 |
| Kyle Weeks | 100 |
| Neth Doze | 50 |
| Matt Bak | 50 |
| Sam Saylor | 50 |
| Nic Foster | 50 |

Первые четыре записи, созданные с помощью команды INSERT, располагаются в алфавитном порядке по именам работников. Остальные записи, добавленные с помощью команды MERGE, располагаются в том порядке, в котором они были в таблице EMPLOYEE.

### Удаление устаревших данных

С течением времени данные могут устаревать и становиться бесполезными. Ненужные данные, находясь в таблице, только замедляют работу системы, расходуют память и путают пользователей. Возможно, лучше перенести старые данные в архивную таблицу, а затем поместить архив с этой таблицей вне системы. Таким образом, в случае такого маловероятного события, когда потребуется снова взглянуть на эти данные, их можно будет восстановить. А в остальное время они не будут замедлять ежедневную обработку данных. Впрочем, независимо от того, было ли решено архивировать устаревшие данные или нет, все же наступит время, когда их надо будет удалить. Для удаления строк из таблицы, находящейся в базе данных, в языке SQL предназначен оператор **DELETE** (удалить).

С помощью единственного оператора DELETE можно удалить как все строки таблицы, так и некоторые из них. Строки, предназначенные для удаления, можно выбрать, используя в операторе DELETE необязательное предложение WHERE. Синтаксис оператора DELETE такой же, как и в операторе SELECT, за исключением того, что столбцы указывать не надо. При удалении строки удаляются все данные, находящиеся в ее столбцах.

Предположим, например, что ваш клиент Дэвид Тейлор переехал на Таити и больше ничего у вас покупать не собирается. Вы можете удалить его данные из таблицы CUSTOMER, используя следующий оператор:

```
DELETE FROM CUSTOMER
WHERE FirstName = 'David' AND LastName = 'Taylor';
```

Если у вас только один покупатель, которого зовут Дэвид Тейлор, то этот оператор будет выполнен безупречно. А если существует вероятность, что Дэвидом Тейлором зовут как минимум двух ваших покупателей? Чтобы удалить данные именно того из них, к кому вы потеряли интерес, добавьте в предложение WHERE дополнительные условия (для таких столбцов, как, например. Street, Phone или CustomerlD).

# Определение значений

В этой главе…

- Использование переменных для уменьшения избыточного кодирования
- Получение часто запрашиваемой информации, находящейся в поле таблицы базы данных
- Комбинирование простых значений для создания составных выражений

---

В этой книге постоянно подчеркивается, насколько важной для поддержания целостности базы данных является структура этой базы. Впрочем, хотя значение структуры базы данных часто недооценивается, но не надо забывать, что наибольшую важность все же представляют сами данные. В конце концов, значения, хранящиеся на пересечении строк и столбцов в таблице базы данных, являются "сырьем", из которого можно получать ценную информацию об имеющихся взаимоотношениях и тенденциях.

Значения можно получать несколькими способами – непосредственно или с помощью функций или выражений. В этой главе описываются разные виды значений, функций и выражений.

***Помни:***
*Функция принимает данные и на их основе вычисляет значение. Выражение является комбинацией элементов данных, из которой SQL в результате вычислений получает единственное значение*.

## Значения

В SQL имеется несколько видов значений:

- значения типа записи;
- литеральные значения;
- переменные;
- специальные переменные;
- ссылки к столбцам.

## Атомы не являются неделимыми

В девятнадцатом веке ученые верили, что атом является той минимальной частью материи, какая только возможна. Поэтому они и назвали эту часть атомом – словом, происходящим от греческого "атомос", что означает "неделимый". А теперь ученым известно, что атомы не являются неделимыми и состоят из протонов, нейтронов и электронов. Протоны и нейтроны, в свою очередь, состоят из кварков, глюонов и виртуальных кварков. Кто знает, может быть, и их нельзя назвать неделимыми?

Значение поля таблицы базы данных называется атомарным, хотя многие поля совсем не являются неделимыми. У значения типа DATE имеются следующие компоненты: месяц, год и день.

А компонентами значения типа TIMESTAMP являются час, минута, секунда и т.д. Значения типов REAL и FLOAT в качестве компонентов имеют экспоненту и мантиссу. В значении типа CHAR есть компоненты, к которым можно получить доступ с помощью SUBSTRING. Поэтому, по аналогии с атомами материи, название "атомарные" для значений полей баз данных все-таки правильно. Впрочем, если исходить из первоначального

значения этого слова, то ни одно из современных применений термина "атомарный" правильным не является.

### Значения типа записи

Самыми заметными значениями в базе данных являются табличные **значения типа записи**. Это значения, которые являются содержимым каждой строки, находящейся в таблице базы данных. Значение этого типа обычно состоит из множества компонентов, ведь в каждом столбце каждой строки всегда находится какое-либо значение. **Поле** – это пересечение столбца и строки. В поле содержится **скалярное**, или **атомарное**, значение. У этого значения имеется только один компонент.

### Литеральные значения

В SQL **значение** может быть представлено или переменной, или константой. Было бы логично считать, что значение переменной время от времени меняется, а значение**константы** (т.е. постоянной величины) не меняется никогда. Важной разновидностью констант является **литеральное значение**. **Литерал** можно считать WYSIWYG-значением, потому что "то, что вы видите, то вы и получаете" (What Kou See Is What Кои Get). Представление литерального значения как раз и является этим самым значением.

Так как в языке SQL имеется много разных типов данных, то в нем имеется и много разных типов литералов. Некоторые примеры литералов разных типов данных приведены в табл. 7.1.

Обратите внимание, что литералы нечисловых типов заключены в апострофы. Эти знаки помогают избежать путаницы, хотя, впрочем, могут и привести к трудностям.

**Таблица 7.1**. Примеры литералов различных типов данных.

| Тип данных | Пример литерала |
|---|---|
| BIGINT | 8589934592 |
| INTEGER | 186282 |
| SMALLINT | 186 |
| NUMERIC | 186282.42 |
| DECIMAL | 186282.42 |
| REAL | 6.02257e23 |
| DOUBLE PRECISION | 3.1415926535897e00 |
| FLOAT | 6.02257e23 |
| CHARACTER (15) ***Примечание:*** *в строке в одинарные кавычки заключено пятнадцать символов и пробелов* | 'GREECE' |
| VARCHAR (CHARACTER VARYING) | 'lepton' |
| NATIONAL CHARACTER(15)***Примечание:****в строке в одинарные кавычки заключено пятнадцать символов и пробелов* | 'ΕΛΛΑΣ' ***Примечание:****Этот термин является словом, которым греки называют Грецию на своем языке. (Если написать его по-английски, то получится "Hellas", а по-русски – "Эллада".)* |
| NATIONAL CHARACTER VARYING | 'λεπτον' Этот термин является словом "lepton " (лептон), написанным буквами греческого алфавита. |
| CHARACTER URGE OBJECT (CLOB) | Очень длинная символьная стока |
| BINARY LARGE OBJECT (BLOB) | Очень длинная строка, состоящая из нулей и единиц (0и1) |
| DATE | DATE '1969-07-20' |
| TIME(2) | TIME '13.41.32.50' |

| Тип данных | Пример литерала |
|---|---|
| TIMESTAMP(O) | TIMESTAMP '1998-05-17-13.03.16.000000' |
| TIME WITH TIMEZONE(4) | TIME '13.41.32.5000-08.00' |
| TIMESTAMP WITH TIMEZONE(4) | TIMESTAMP '1998-05-17-13.03.16.0000+02.00' |
| INTERVAL DAY | INTERVAL '7 DAY' |

А если литерал является символьной строкой, содержащей символ одинарной кавычки? В таком случае вместо одного этого символа в литерале должны быть две одинарные кавычки подряд, чтобы показать, что кавычка является частью строки и не указывает на ее завершение. Таким образом, чтобы получился символьный литерал 'Earth's atmosphere', необходимо ввести 'Earth''s atmosphere'.

### Переменные

Прекрасно, когда при работе с базами данных можно манипулировать литералами и другими константами. Однако полезно иметь и переменные. Во многих случаях, не имея переменных, приходится делать намного больше работы. Переменная – это такая величина, значение которой может изменяться. Чтобы увидеть, почему переменные так полезны, рассмотрим следующий пример.

Предположим, что вы розничный продавец, у которого есть покупатели нескольких категорий. Тем из них, кто покупает товары в больших объемах, вы продаете эти товары по самым низким ценам. Тем же, кто покупает в средних объемах, вы продаете товары по ценам более высокого порядка. И наконец, те, кто ограничивается при покупках малыми объемами шаров, платят самую высокую цену. Вы хотите, чтобы все розничные цены имели определенные коэффициенты по отношению к той стоимости, в какую товары обошлись вам. Для своего товара F-117A вы решили, что покупатели товаров в больших объемах (покупатели класса C) будут за него платить в 1.4 раза больше, чем платите за этот товар вы. А покупатели товаров в средних объемах (покупатель класса B) будут уже платить в 1.5 раза больше. И наконец, покупатели товаров в малых объемах (покупатели класса A) – в 1.6 раза больше.

Вы храните значения стоимости товаров и назначаемых вами цен в таблице, которую вы назвали PRICING (ценообразование). Среди ее полей имеются такие: PRICE (цена), COST (стоимость), PRODUCT (продукт) и CLASS (класс). Чтобы реализовать свою новую структуру ценообразования, вы отправляете на выполнение следующие команды языка SQL:

```
UPDATE PRICING
SET Price = Cost * 1.4
WHERE Product = 'F-117A'
AND Class = 'C;
UPDATE PRICING
SET Price = Cost * 1.5
WHERE Product = 'F-117A'
AND Class = 'B';
UPDATE PRICING
SET Price = Cost * 1.6
WHERE Product = 'F-117A'
AND Class = 'A';
```

Этот код прекрасный и пока что подходит для ваших нужд. А что если энергичные усилия конкурентов начинают подрывать ваш сектор рынка? Чтобы остаться на плаву, вам, возможно, придется уменьшить установленные вами значения разницы в ценах. Тогда потребуется ввести нечто похожее на строки следующих команд:

```
UPDATE PRICING
SET Price = Cost * 1.25
WHERE Product = 'F-117A'
AND Class = 'C;
UPDATE PRICING
SET Price = Cost * 1.35
WHERE Product = 'F-117A'
AND Class = 'B';
UPDATE PRICING
SET Price = Cost * 1.45
WHERE Product = 'F-117A'
AND Class = 'A';
```

Если ваш рынок изменчив, то вам придется время от времени переписывать свой SQL-код. Это может потребовать значительных усилий с вашей стороны, особенно если цены указаны во многих местах вашего кода. Эти усилия можно свести к минимуму, если заменить литералы (например, 1.45) переменными (такими, например, как multiplierA). Тогда свои операции обновления вы можете выполнять таким образом:

```
UPDATE PRICING
SET Price = Cost *:multiplierC
WHERE Product = 'F-117A'
AND Class = 'C;
UPDATE PRICING
SET Price = Cost *:multiplierB
WHERE Product = 'F-117A'
AND Class = 'B';
UPDATE PRICING
SET Price = Cost *:multiplierA
WHERE Product = 'F-117A'
AND Class = 'A';
```

Теперь в любом случае, когда условия на рынке заставят вас менять ценообразование, остается только изменить значения переменных: **:multiplierC**, **:multiplierB** и **:multiplierA**. Эти переменные являются параметрами, передаваемыми SQL-коду, который затем использует полученные переменные, чтобы считать новые цены.

***Технические подробности:***

*Иногда переменные, используемые таким образом, называются параметрами, а иногда – базовыми переменными. Переменные называются параметрами, если они находятся в приложениях, написанных на модульном языке SQL, а базовыми переменными – если используются во встроенном SQL*.

***Помни:***

*Встроенный SQL означает, что операторы SQL встроены в код приложения, написанного на процедурном базовом языке. Кроме того, SQL-код можно поместить в модуль SQL. Модуль вызывается приложением, написанным на базовом языке. Каждый из этих двух методов имеет собственные преимущества и недостатки. Какой из них выбрать – это зависит от используемой вами конкретной реализации SQL*.

## Специальные переменные

Как только пользователь на клиентской машине соединяется с базой данных, находящейся на сервере, устанавливается **сеанс**. Если пользователь соединяется с несколькими базами данных, то сеанс, связанный с самым последним соединением, называется текущим, а предыдущие сеансы считаются **бездействующими**. Стандарт SQL:2OO3 определяет несколько специальных переменных, применяемых в многопользовательских системах. Эти переменные содержат данные о различных пользователях. Например, специальная переменная SESSION_USER (пользователь сеанса) содержит значение пользовательского идентификатора авторизации для текущего сеанса SQL. Вы можете написать программу мониторинга, определяющую, кто отправляет на выполнение операторы SQL, с помощью переменной SESSION_USER.

У модуля SQL может быть связанный с ним идентификатор авторизации, который определяется пользователем. Его значение хранится в переменной CURRENT_USER (текущий пользователь). Если такого идентификатора у модуля нет, то переменная CURRENT_USER имеет такое же значение, что и SESSION_USER.

В переменной SYSTEM_USER (системный пользователь) хранится идентификатор пользователя операционной системы. Он может отличаться от идентификатора этого пользователя, хранящегося в модуле SQL. Например, пользователь может регистрироваться в системе как LARRY (Ларри), а в модуле – уже как PLANT_MGR (директор завода). Таким образом, в переменной SESSION_USER будет храниться значение PLANT_MGR. Если этот пользователь явно не указывает идентификатор модуля, то значение PLANT_MGR будет храниться и в переменной CURRENT_USER. А значение LARRY будет храниться в переменной SYSTEM_USER.

Специальные переменные SYSTEM_USER, SESSION_USER и CURRENT_USER применяются для сбора данных о том, какие именно пользователи работают в системе. Вы можете поддерживать таблицу-журнал и периодически вставлять в нее значения, содержащиеся в этих переменных. Как это сделать, показано в следующем примере:

```
INSERT INTO USAGELOG (SNAPSHOT)
VALUES ('User1 SYSTEM_USER ||
'with ID ' || SESSION_USER ||
'active at ' || CURRENT_TIMESTAMP);
```

При выполнении этого оператора создаются примерно такие журнальные записи:

```
User LARRY with ID PLANT_MGR active at 1998-05-17-14:18:00
```

### Ссылки к столбцам

В столбцах находятся значения, по одному в каждой табличной строке. Ссылки к таким значениям часто используются в операторах SQL. Полностью определенная ссылка к столбцу состоит из имени таблицы, точки и имени столбца (например, PRICING.Product). Посмотрите на следующий оператор:

```
SELECT PRICING. Cost
FROM PRICING
WHERE PRICING. Product = 'F-117A';
```

Где PRICING. Product – это ссылка на столбец, которая содержит значение 'F-117A'. PRICING.Cost – это также ссылка на столбец, но вы не будете знать ее значения, пока не выполнится предшествующий этой ссылке оператор SELECT.

Так как имеет смысл делать ссылки только к тем столбцам, которые находятся в текущей таблице, то обычно эти ссылки полностью определять не нужно. Например, следующий оператор равнозначен предыдущему:

```
SELECT Cost
FROM PRICING
WHERE Product = 'F-117A';
```

Иногда все же приходится работать одновременно с разными таблицами. В базе данных у каких-либо двух таблиц могут быть столбцы с одинаковыми именами. В таком случае ссылки к этим столбцам приходится определять полностью. Это нужно для того, чтобы получаемый столбец был действительно тем, который вам нужен.

Предположим, например, что ваша компания имеет филиалы, расположенные в Кингстоне и Джефферсоне, и вы отдельно для каждого из этих филиалов ведете данные по работающим там сотрудникам. Ваша таблица по сотрудникам, работающим в Кингстоне, называется EMP_KINGSTON, а по работающим в Джефферсоне – EMP_JEFFERSON. Вам необходим список всех сотрудников, которые работают в обоих местах, поэтому следует найти всех тех, у кого имя вместе с фамилией находятся в обеих таблицах. То, что нужно, дает следующий оператор SELECT:

```
SELECT EMP_KINGSTON.FirstName, EMP_KINGSTON.LastName
FROM EMP_KINGSTON, EMP_JEFFERSON
WHERE EMP_KINGSTON.EmpID = EMP_JEFFERSON.EmpID;
```

Так как идентификационный номер сотрудника является уникальным и имеет одно и то же значение независимо от филиала, в котором сотрудник работает, то этот номер можно использовать для связи между таблицами (в каждой из них он находится в столбце EmpID). В результате выполнения последнего оператора возвращаются имена и фамилии только тех сотрудников, чьи данные находятся в обеих таблицах. Имена и фамилии берутся соответственно из столбцов FirstName и LastName таблицы EMP_KINGSTON.

### Выражения со значением

Выражение может быть простым или очень сложным. В нем могут находиться литеральные значения, имена столбцов, параметры, базовые переменные, подзапросы, логические связки и арифметические операторы. Впрочем, каким бы сложным выражение ни было, оно обязательно должно сводиться к одиночному значению.

Поэтому выражения SQL обычно называются **выражениями со значением**. Комбинирование множества таких выражений в одно возможно тогда, когда эти выражения-компоненты сводятся к значениям, имеющим совместимые типы данных.

В SQL определяется пять разных типов выражений со значением:

- строковые;
- числовые;
- даты-времени;
- интервальные;
- условные.

### Строковые выражения со значением

Самым простым **строковым выражением со значением** является одиночное строковое значение. В более сложных выражениях могут быть также ссылки на столбцы, итоговые функции, скалярные подзапросы, выражения с использованием ключевых слов CASE и CAST или составные строковые выражения со значением. О выражениях со значением, использующих CASE и CAST, рассказывается в главе 8. В строковых выражениях со значением можно применять только один оператор – **оператор конкатенации**. Его можно применять к любым выражениям, чтобы, соединив их вместе, получить более сложное строковое выражение со значением. Оператор конкатенации представлен двумя вертикальными линиями (||). Некоторые примеры строковых выражений со значением показаны в следующей таблице.

| Выражение | Результат |
|---|---|
| 'Хрустящий ' || 'арахис' | 'Хрустящий арахис' |
| 'Шарики' || ' ' || 'из желе' | 'Шарики из желе' |

| Выражение | Результат |
|---|---|
| FIRST_NAME \|\| ' ' \|\| LAST_NAME | 'Джо Смит' |
| B'1100111' \|\| B'01010011' | B'110011101010011' |
| ' ' \|\| 'Спаржа' | 'Спаржа' |
| 'Спаржа' \|\| ' ' | 'Спаржа' |
| 'С' \|\| ' ' \|\| 'пар' \|\| ' ' \|\| 'ж' \|\| ' ' \|\| 'а' | 'Спаржа' |

Как показывают примеры из таблицы, если объединять какую-либо строку со строкой нулевой длины, то результат будет таким же, как и первоначальная строка.

### Числовые выражения со значением

В **числовых выражениях со значением** к числовым данным можно применять операторы сложения, вычитания, умножения и деления. Такое выражение обязательно должно сводиться к числовому значению. Компоненты числового выражения со значением могут иметь разные типы данных или все могут быть числовыми. Тип данных результата зависит от типов данных компонентов, из которых получается этот результат. В стандарте SQL:2003 нет жесткого определения, каким образом тип данных результата, получаемого при выполнении выражения, должен зависеть от исходных компонентов этого выражения. Это объясняется различиями аппаратных платформ. Поэтому, если вы используете смешанные типы данных, обращайтесь к документации по той платформе, на которой работаете.

Вот некоторые примеры числовых выражений со значением.

- 72ɹ
- 49 + 83
- 5* (12-3)
- PROTEIN + FAT + CARBOHYDRATE
- FEET/5280
- COST *:multilierA

### Выражения со значением даты-времени

Выражения со значением даты-времени выполняют операции с данными, относящимися к дате и времени. Компоненты этих выражений могут иметь типы данных DATE, TIME, TIMESTAMP и INTERVAL. Результат выполнения выражения со значением даты-времени всегда относится к одному из типов даты-времени (DATE, TIME или TIMESTAMP). Например, после выполнения следующего выражения будет получена дата, которая наступит ровно через неделю:

```
CURRENT_DATE + INTERVAL '7' DAY
```

Значения времени поддерживаются в координатах Всемирного времени (**Universal Time Coordinates**, UTC), ранее известных как время по Гринвичу. Однако можно указывать и смещение, чтобы время соответствовало текущему часовому поясу. Для местного часового пояса, применяемого в вашей системе, можно использовать простой синтаксис, пример которого приведен ниже.

```
TIME '22.55.00' AT LOCAL
```

Кроме того, это значение можно указать и более развернуто:

```
TIME '22.55.00' AT TIME ZONE INTERVAL '-08.00' HOUR TO MINUTE
```

Последнее выражение определяет местное время часового пояса, в котором находится город Портленд, штат Орегон. Этот часовой пояс отстоит от Гринвича на восемь часов.

### Интервальные выражения со значением

Если взять два значения даты-времени и от одного из них отнять другое, то получится**интервал**. Но сложение таких значений друг с другом не имеет смысла, поэтому SQL эту операцию не поддерживает. Если же сложить друг с другом два интервала или вычесть один из другого, то в результате снова получится интервал. Кроме того, интервал можно умножать или делить на числовую константу.

Вспомните, что в SQL имеются два типа интервалов: **год-месяц** и **день-время**. Чтобы избежать двусмысленности, необходимо в интервальном выражении со значением указывать, какой из этих типов в нем используется. Например, в следующем выражении вычисляется интервал в годах и месяцах от текущей даты до дня, когда вы достигнете пенсионного возраста (60 лет):

```
(BIRTHDAY_60 ıCURRENT_DATE) YEAR TO MONTH
```

А это возвращает интервал в 40 дней:

```
INTERVAL '17' DAY + INTERVAL '23' DAY
```

Ниже приблизительно подсчитывается общее число месяцев, в течение которых мать пятерых детей была беременна при условии, что сейчас она не ждет шестого.

```
INTERVAL '9' MONTH * 5
```

Интервалы могут быть как положительные, так и отрицательные и состоять из любого выражения со значением или комбинации таких выражений, значением которой является интервал.

### Условные выражения со значением

Значение **условного выражения со значением** зависит от условия. Такие выражения, как CASE, NULLIF и COALESCE, значительно сложнее, чем другие выражения со значением. Эти три вида условных выражений настолько сложны, что заслуживают отдельного рассмотрения. Подробно о них речь пойдет в главе 8.

### Функции

Функция – это простая или достаточно сложная операция, которую обычные команды SQL выполнить не могут, но которая тем не менее достаточно часто встречается на практике. В SQL имеются функции, выполняющие работу, которую иначе пришлось бы выполнять приложению, написанному на базовом языке. В языке SQL есть две главные разновидности функций: **итоговые функции** и **функции значения**.

### Суммирование с помощью итоговых функций

Итоговые функции применяются к наборам строк из таблицы, а не только к ее отдельным строкам. Эти функции в текущем наборе строк "суммируют" некоторые характеристики, т.е. получают по ним определенные итоги. В такой набор могут входить все строки таблицы или только те из них, которые определяются предложением WHERE. (Подробно о предложениях WHERE рассказывается в главе 9.)

Программисты используют название **итоговые функции**, потому что те берут информацию из целого набора строк, определенным образом ее обрабатывают и выдают результат в виде единичной строки. Кроме того, эти функции еще называются **функциями наборов**.

Чтобы показать применение итоговых функций, проанализируйте табл. 7.2, в которой представлены питательные компоненты, содержащиеся в 100 граммах некоторых продуктов питания.

**Таблица 7.2**. Питательные компоненты некоторых продуктов питания (в 100 граммах).

| Продукт питания (Food) | Калории (Calories) | Белки (Protein), г | Жиры (Fat), г | Углеводы (Carbogidrate), г |
|---|---|---|---|---|

| Продукт питания (Food) | Калории (Calories) | Белки (Protein), г | Жиры (Fat), г | Углеводы (Carbogidrate), г |
|---|---|---|---|---|
| Жареные миндальные орехи | 627 | 18.6 | 57.7 | 19.6 |
| Спаржа | 20 | 2.2 | 0.2 | 3.6 |
| Сырые бананы | 85 | 1.1 | 0.2 | 22.2 |
| Гамбургер с нежирной говядиной | 219 | 27.4 | 11.3 | |
| Нежное мясо цыплят | 166 | 31.6 | 3.4 | |
| Жареный опоссум | 221 | 30.2 | 10.2 | |
| Свиной окорок | 394 | 21.9 | 33.3 | |
| Фасоль лима | 111 | 7.6 | 0.5 | 19.8 |
| Кола | 39 | | | 10.0 |
| Белый хлеб | 269 | 8.7 | 3.2 | 50.4 |
| Пшеничный хлеб | 243 | 10.5 | 3.0 | 47.7 |
| Брокколи | 26 | 3.1 | 0.3 | 4.5 |
| Сливочное масло | 716 | 0.6 | 81.0 | 0.4 |
| Шарики из желе | 367 | | 0.5 | 93.1 |
| Хрустящий арахис | 421 | 5.7 | 10.4 | 81.0 |

Информация из табл. 7.2 хранится в таблице FOODS (продукты), находящейся в базе данных. В пустых полях находится значение NULL. Сообщить важные сведения о данных из этой таблицы помогают итоговые функции COUNT, AVG, MAX, MIN и SUM.

### COUNT

Функция COUNT (счет) сообщает, сколько строк находится в таблице или сколько строк таблицы удовлетворяют некоторые условия. Вот самое простое применение этой функции:

```
SELECT COUNT (*)
FROM FOODS;
```

Функция возвращает результат, равный 15, так как она считает все строки таблицы FOODS. Тот же результат получается при выполнении следующего оператора:

```
SELECT COUNT (CALORIES)
FROM FOODS;
```

Так как значение в столбец CALORIES (калории) было введено в каждой строке, то результат подсчета получается тот же. Правда, если в некоторых строках в этом столбце находятся неопределенные значения, функция такие строки не считает.

Следующий оператор возвращает значение, равное 11, так как в 4 из 15 строк таблицы FOODS в столбце CARBOHYDRATE (углеводы) находится значение NULL:

```
SELECT COUNT (CARBOHYDRATE)
FROM FOODS;
```

***Совет:***

*В поле таблицы базы данных неопределенное значение может находиться по разным причинам. Самыми распространенными являются следующие: значение вообще не известно или пока не известно. Или значение, возможно, известно, но пока еще не введено. Иногда, если значение, предназначенное для какого-либо поля, равно нулю, оператор, вводящий данные, это поле обходит стороной и, таким образом, оставляет в нем значение NULL. Так поступать не надо, потому что*

*нуль все же является определенным значением и его можно учитывать при подсчетах. A NULL определенным значением не является, и в SQL неопределенные значения не учитываются при расчетах.*

Кроме того, чтобы узнать, сколько в столбце имеется различных значений, можно использовать функцию COUNT в сочетании с ключевым словом DISTINCT (различный). Проанализируйте следующий оператор:

```
SELECT COUNT (DISTINCT Fat)
FROM FOODS;
```

Он возвращает значение, равное 12. Как видно в таблице, в 100-граммовой порции спаржи имеется столько жиров (0.2 грамма), сколько и в 100 граммах бананов, а в 100-граммовой порции фасоли лима – ровно столько жиров (0.5 грамма), сколько в 100 граммах желе. Таким образом, в таблице находится всего 12 разных значений, имеющих отношение к содержанию жиров.

### AVG

Функция AVG (среднее) подсчитывает и возвращает среднее арифметическое всех значений, находящихся в определенном столбце. Конечно, эту функцию можно применять только к столбцам с числовыми данными, как в следующем примере:

```
SELECT AVG (Fat)
FROM FOODS;
```

В результате получается среднее содержание жиров, равное 15.37. Это число достаточно высокое. Дело в том, что весь подсчет портит информация по сливочному маслу. Возможно, вы зададите себе вопрос, а каким было бы среднее содержание жиров, если бы не учитывалось масло. Чтобы ответить на него, в оператор можно поместить выражение WHERE:

```
SELECT AVG (Fat)
FROM FOODS
WHERE FOOD <> 'Butter';
```

В этом случае содержание жиров в 100 граммах пищевых продуктов в среднем падает до 10.32 грамма.

### MAX

Функция MAX (максимум) возвращает максимальное значение, найденное в указанном столбце. Следующий оператор возвращает значение, равное 81 (количество жиров в 100 граммах сливочного масла):

```
SELECT MAX (Fat)
FROM FOODS;
```

### MIN

Функция MIN (минимум) возвращает минимальное значение, обнаруженное в указанном столбце. Следующий оператор возвращает значение, равное 0.4, потому что функция не учитывает неопределенные значения:

```
SELECT MIN (Carbohydrate)
FROM FOODS;
SUM
```

Функция SUM (сумма) возвращает сумму всех значений, обнаруженных в указанном столбце. Следующий оператор возвращает число 3924, которое является общим количеством калорий во всех 15 продуктах:

```
SELECT SUM (Calories)
FROM FOODS;
```

### Функции значения

Некоторые операции применяются для самых разных целей. Так как эти операции приходится использовать достаточно часто, то было бы более чем оправданным включить их в SQL в виде функций значения. Конечно, если сравнивать с такими системами управления базами данных, как Access или dBASE, то в SQL этих функций довольно-таки мало, но те немногие, что есть, являются, вероятно, функциями, которые нужны вам чаще всего. В SQL используются три вида таких функций:

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

### Строковые функции

**Строковые функции** принимают значение одной символьной или битовой строки и возвращают другую символьную или битовую строку. В SQL имеется шесть таких функций:

- SUBSTRING;
- UPPER;
- LOWER;
- TRIM;
- TRANSLATE;
- CONVERT.

### SUBSTRING

Функция SUBSTRING (подстрока) используется для того, чтобы из исходной строки выделить подстроку. Выделенная функцией подстрока имеет тот же тип, что и исходная. Например, если исходная строка является символьной, то и подстрока также является символьной. Вот синтаксис функции SUBSTRING:

```
SUBSTRING (строковое_значение FROM начало [FOR длина]).
```

Предложение в квадратных скобках ([]) не является обязательным. Подстрока, которую следует выделить из **строкового_значения**, начинается с символа, порядковый номер которого, если считать с самого первого символа, представлен значением начало. Кроме того, подстрока состоит из определенного количества символов, представленного значением**длина**. Если предложение FOR отсутствует, то подстрока выделяется, начиная от символа, соответствующего значению **начало**, до самого конца строки. Проанализируйте следующий пример:

```
SUBSTRING ('Полностью пшеничный хлеб' FROM 11 FOR 11)
```

Выделенной подстрокой является 'пшеничный х'. Она начинается с одиннадцатого символа исходной строки и имеет длину в одиннадцать символов. С первого взгляда SUBSTRING не представляется такой уж ценной функцией. Для литерала 'Полностью пшеничный хлеб' не требуется функция нахождения подстроки. Впрочем, функция SUBSTRING действительно представляет ценность, потому что строковое значение не обязательно должно быть литералом. Это значение может быть любым выражением, в результате выполнения которого получается символьная строка. Например, это может быть переменная fooditem, которая каждый раз может принимать разные значения. Следующее выражение может извлекать нужную подстроку независимо от того, какую символьную строку представляет переменная fooditem:

```
SUBSTRING (:fooditem FROM 11 FOR 11).
```

Все функции значения объединяет то, что они могут оперировать как со значениями, так и с выражениями, после выполнения которых получаются значения требуемого типа.

***Внимание:***

*При использовании функции SUBSTRING не следует забывать о следующем. Выбираемая вами подстрока действительно должна быть частью исходной строки. Если вам нужна подстрока, которая начинается с одиннадцатого символа, а в исходной строке всего только четыре символа, то вы получите значение NULL. Поэтому необходимо иметь некоторое представление о структуре*

*своих данных, перед тем как задавать значения для функции SUBSTRING. Кроме того, нельзя указывать отрицательную длину подстроки.*

Если у столбца тип данных VARCHAR, ширина поля конкретной строки не будет известна. Если вы укажете слишком большую длину подстроки, при которой та выйдет за правый край поля, функция SUBSTRING возвратит конец исходной строки и не будет сообщать об ошибке.

Скажем, у вас имеется оператор:

```
SELECT * FROM FOODS
WHERE SUBSTRING (Food FROM 7 FOR 7) = 'хлеб';
```

И даже если значение, находящееся в столбце FOOD таблицы FOODS, имеет длину меньше 14 символов, этот оператор все равно возвращает табличную строку с данными, относящимися к белому хлебу.

***Совет:***
*Если какой-либо оператор функции SUBSTRING имеет значение NULL, то эта функция возвращает результат NULL.*

**UPPER**

Другая функция, UPPER (верхний регистр), преобразует все символы символьной строки в верхний регистр, как показано в следующей таблице, в примерах со строками 'e.e.cummings' и Isaac Newton, Ph.D.'.

| Выражение | Результат |
|---|---|
| UPPER ('e.e.cummings') | 'E.E.CUMMINGS' |
| UPPER ('Isaac Newton, Ph.D.') | 'ISAAC NEWTON, PH.D.' |

Функция UPPER не оказывает воздействия на строку, все символы которой уже находятся в верхнем регистре.

**LOWER**

Другая функция, LOWER (нижний регистр), преобразует все символы символьной строки в нижний регистр, как показано в следующей таблице, в примерах со строками 'TAXES' и 'E.E.Cummings'.

| Выражение | Результат |
|---|---|
| LOWER ('TAXES') | 'taxes' |
| LOWER ('E. E. Cummings') | 'e. e .cummings' |

Функция LOWER не оказывает воздействия на строку, все символы которой уже находятся в нижнем регистре.

**TRIM**

Чтобы из символьной строки удалить ведущие, замыкающие или одновременно и те и другие пробелы (и не только пробелы), используйте функцию TRIM (обрезать). Следующие

примеры показывают, как ее использовать, например, применительно к строкам, где находится слово treat.

| Выражение | Результат |
|---|---|
| TRIM (LEADING ' ' FROM ' treat ') | 'treat ' |
| TRIM (TRAILING ' ' FROM ' treat ') | ' treat' |
| TRIM (BOTH ' ' FROM ' treat ') | 'treat' |
| TRIM (BOTH 't' FROM 'treat') | 'rea' |

Символом по умолчанию для этой функции является пробел, поэтому следующий синтаксис также правильный:

```
TRIM (BOTH FROM ' treat ').
```

В этом случае получается тот же результат, что и в третьем примере из таблицы, а именно 'treat'.

### TRANSLATE и CONVERT

Функции TRANSLATE (перевести) и CONVERT (преобразовать) выбирают исходную строку, составленную из символов одного набора, и переводят ее в строку, составленную из символов другого набора. Примерами могут быть переводы символов из английского набора в армянский или символов иврита во французский. Функции преобразования, выполняющие эти действия, зависят от реализации SQL. Подробности можно узнать в документации по имеющейся у вас реализации.

***Помни:***
*Если бы перевод с одного языка на другой был таким легким, как вызов в SQL функции TRANSLATE, то это было бы прекрасно. К сожалению, такая задача – не из легких. Все, что осуществляет TRANSLATE, – это перевод символа из первого символьного набора в соответствующий символ из второго набора. Она может, например, перевести 'ELLAS' (Греция) в 'Ellas'. Однако функция TRANSLATE не может перевести 'ELLAS' в 'Greece' (Греция)*.

### Числовые функции

Числовые функции значения могут принимать данные разных типов, но возвращают всегда числовое значение. В SQL имеется тринадцать таких функций.

- Положение (POSITION).
- Извлечение (EXTRACT).
- Длина (CHAR_LENGHT, CHARACTER_LENGHT, OCTET_LENGHT).
- Кардинальные числа (CARDINALITY).
- Абсолютное значение (ABS).
- Остаток от деления нацело (MOD).
- Натуральный логарифм (LN).
- Экспонента (EXP).
- Возведение в степень (POWER).
- Квадратный корень (SQRT).
- Округление "вниз" (FLOOR).
- Округление "вверх" (CEIL, CEILING).
- Интервальный номер (WIDTH_BUCKET).

### POSITION

Функция POSITION (положение) ищет указанную целевую строку внутри указанной исходной и возвращает положение в ней начального символа целевой строки. Эта функция имеет такой синтаксис:

```
POSITION (целевая_строка IN исходная_строка)
```

В следующей таблице приведено несколько примеров использования POSITION для исходной строки 'Полностью пшеничный хлеб'.

| Выражение | Результат |
|---|---|
| POSITION ('П' IN 'Полностью пшеничный хлеб') | 1 |
| POSITION ('Пол' IN 'Полностью пшеничный хлеб') | 1 |
| POSITION ('пш' IN 'Полностью пшеничный хлеб') | 11 |
| POSITION ('пшо' IN 'Полностью пшеничный хлеб') | 0 |
| POSITION ('' IN 'Полностью пшеничный хлеб') | 1 |

Если эта функция не находит целевую строку, то возвращает неопределенное значение. А если у целевой строки нулевая длина (как в последнем примере), то функция POSITION всегда возвращает единицу. Если любой из операндов этой функции имеет значение NULL, то в результате ее выполнения получится NULL.

### EXTRACT

Функция EXTRACT (извлечь) извлекает одиночное поле из значения типа даты-времени или интервала. Например, следующее выражение возвращает 08:

```
EXTRACT (MONTH FROM DATE '2000-08-20').
```

### CHARACTER_LENGTH

Функция CHARACTER_LENGTH (длина в символах) возвращает количество символов, находящихся в символьной строке. Например, следующее выражение возвращает 15:

```
CHARACTER_LENGTH (' Жареный опоссум')
```

***Помни:***

*То, что уже говорилось в этой главе по поводу функции SUBSTRING, относится и к CHARACTER_LENGTH – эта функция не особенно полезна, если ее аргументами являются литералы, например, такие как 'Жареный опоссум'. Вместо выражения CHARACTER_LENGTH ('Жареный опоссум') можно написать число 15. Действительно, написать '15' проще. Функция SUBSTRING становится более полезной, если ее аргумент является не литеральным значением, а выражением*.

### OCTET_LENGTH

Что касается мира музыки, то в нем вокальный ансамбль, состоящий из восьми певцов, называется **октетом**. В этом ансамбле обычно есть первое и второе сопрано, первый и второй альт, первый и второй тенор, а также первый и второй бас. А что касается мира информатики, то в нем ансамбль, состоящий из восьми битов данных, называется **байтом**. Слово **байт** ясно показывает, что оно имеет отношение к **биту**, но при этом подразумевается нечто большее. Прекрасная игра слов, но, к сожалению, в слове "байт" ничто не напоминает о "восьмеричности". А если позаимствовать музыкальный термин, то

тогда набор из восьми битов будет иметь более подходящее и более описательное название.

Практически во всех современных компьютерах для представления одного алфавитно-цифрового символа используется восемь битов. А в более сложных наборах символов (таких, например, как китайский) для этого требуется уже 16 битов. Функция OCTET_LENGTH (длина в октетах) подсчитывает и возвращает количество октетов (байтов), находящихся в строке. Если строка является битовой, то эта функция возвращает такое количество октетов, чтобы вместить находящееся в строке количество битов. А если строка состоит из символов англоязычного набора (с одним октетом на символ), то OCTET_LENGTH возвращает количество символов, имеющихся в строке. Если же строка состоит из символов китайского набора, то тогда число, возвращаемое функцией, в два раза превышает количество китайских символов. Например:

```
OCTET_LENGTH ('Beans, Lima')
```

Эта функция возвращает 11, потому что каждый символ помещается в октете.

В некоторых наборах символов для разных символов используется разное количество октетов. В частности, в тех из них, которые поддерживают смешанное использование символов канджи и латиницы, для перехода из одного набора символов в другой используются **управляющие последовательности**. Например, для строки, состоящий из 30 латинских символов, потребуется 30 октетов. А если все ее 30 символов взяты из канджи, то для этой строки нужно 62 октета (60 октетов плюс ведущий и замыкающий символы переключения). И наконец, если в этой строке символы латиницы и канджи попеременно чередуются друг с другом, то тогда для нее требуется 150 октетов. (Тогда для каждого символа канджи требуется два октета, а также по одному октету для ведущего и замыкающего символа переключения.) Функция OCTET_LENGTH возвращает то количество октетов, которое нужно, чтобы поместить в них имеющуюся строку.

### CARDINALITY

Эта функция работает с коллекциями элементов, такими как массивы или мультимножества, где каждый элемент является значением определенного типа данных. Кардинальное число коллекции – это количество содержащихся в ней элементов. Рассмотрим один из примеров использования функции CARDINALITY:

```
CARDINALITY (TeamRoster)
```

Например, если в списке членов команды значится двенадцать человек, то эта функция возвращает значение 12. Столбец TeamRoster таблицы TEAM может быть как массивом, так и мультимножеством. В свою очередь, массив – это упорядоченная коллекция элементов, а мультимножество – неупорядоченная коллекция. Для списка команды, который может изменяться достаточно часто, разумней использовать мультимножество.

### ABS

Функция ABS возвращает абсолютное значение числового выражения.

```
ABS (-273)
```

Функция возвращает 273.

### MOD

Функция MOD возвращает остаток от деления нацело первого числового выражения на второе числовое выражение.

```
MOD (3.2)
```

Данная функция возвращает числовое значение 1 – остаток, получаемый при делении нацело числа 3 на число 2.

### LN

Функция LN возвращает натуральный логарифм от числового.

```
LN (9)
```

Возвращаемое значение будет приближенно равно 2.197224577. Количество знаков числа после запятой зависит от вашей реализации.

### EXP

Данная функция возводит основание натурального логарифма e в степень, указанную числовым выражением.

```
EXP (2)
```

Функция возвращает значение, которое приближенно равно 7.389056. Количество знаков числа после запятой зависит от вашей реализации.

### POWER

Функция POWER возводит первое числовое выражение в степень, указанную вторым числовым выражением.

```
POWER (2.8)
```

В этом примере функция возвращает значение 256, т.е. два в восьмой степени.

### SQRT

Эта функция возвращает квадратный корень числового выражения.

```
SQRT (4)
```

Функция возвращает значение 2 – квадратный корень из четырех.

### FLOOR

Функция FLOOR округляет числовое выражение до наибольшего целого числа, не превышающего данное выражение.

```
FLOOR (3.141592)
```

Функция возвращает значение 3.0.

### CEIL, или CEILING

Данная функция округляет числовое выражение до наименьшего целого числа, которое не меньше, чем данное выражение.

```
CEIL (3.141592)
```

Функция возвращает значение 4.0.

### WIDTH_BUCKET

Функция WTDTH_BUCKET используется при выполнении процессов в режиме реального времени (online application processing, OLAP). Эта функция имеет четыре аргумента и возвращает целое число между 0 (нулем) и значением последнего аргумента плюс 1 (один). Для первого аргумента она назначает область в разделенном на равновеликие части диапазоне чисел между вторым и третьим аргументами функции. Для значений, находящихся за пределами заданного диапазона, функция возвращает значение 0 (нуль) либо значение последнего аргумента плюс 1 (один).

Например:

```
WIDTH_BUCKET (PI, 0, 9, 5)
```

Предположим, что PI– числовое выражение со значением– это 3.141592. Интервал значений между нулем и девятью (0 и 9 – второй и третий аргументы функции

соответственно) нужно разделить на пять равных отрезков (5 – четвертый аргумент функции), каждый шириной в две единицы. В этом случае функция возвращает значение 2, поскольку число 3.141592 находится во втором отрезке, который является диапазоном значений от двух до четырех.

### Введение операторов языка SQL в базу данных приложения Microsoft Access

Access не позволяет ввести операторы SQL в базу данных, поэтому все операторы должны быть введены как запросы. Многие программные продукты, например SQL Server, Oracle, MySQL или PostgreSQL, имеют редакторы, предназначенные для ввода операторов языка SQL. Так, в приложении SQL Server это редактор Query Analyzer. Для других приложений такие редакторы описаны в документации.

Для Access также существует возможность ввода операторов SQL, однако этот путь весьма сложен и зачастую запутан. Подробные пошаговые инструкции относительно ввода операторов SQL в приложение Access см. в главе 4.

### Функции значения даты-времени

В языке SQL имеются три функции, которые возвращают информацию о текущей дате, текущем времени или о том и другом вместе. CURRENT_DATE возвращает текущую дату, CURRENT_TIME – текущее время, а CURRENT_TIMESTAMP – текущую дату и текущее время. Первая из этих функций не принимает аргументов, а вторые две – только один. Этот аргумент указывает точность для секундной части возвращаемого функцией значения времени. О типах данных **даты-времени** и о том, что такое точность, см. в главе 2.

Некоторые примеры функций значения даты-времени приведены в следующей таблице.

| Выражение | Результат |
|---|---|
| CURRENT_DATE | 2000-12-31 |
| CURRENT_TIME (1) | 08:36:57.3 |
| CURRENT_TIMESTAMP (2) | 2000 12 31 08:36:57.38 |

Дата, возвращаемая функцией CURRENT_DATE, имеет тип данных не CHARACTER, а DATE. Время, возвращаемое функцией CURRENT_TIME (p), имеет, в свою очередь, тип данных TIME, значение даты и времени, возвращаемое функцией CURRENT_TIMESTAMP (p), имеет тип данных TIMESTAMP. Так как информацию о дате и времени средства SQL получают из системных часов компьютера, то эта информация является правильной для того часового пояса, в котором находится компьютер.

В некоторых приложениях значения даты и времени требуется представлять в виде символьных строк. Преобразование типа данных можно выполнять с помощью выражения CAST (приведение), которое описывается в главе 8.

## Сложные выражения со значением

В этой главе…

- Использование условных выражений case
- Преобразование элемента данных из одного типа данных в другой
- Экономия времени ввода данных с помощью выражений со значением типа запись

---

В главе 2 SQL был назван **подъязыком данных**. Фактически единственная задача этого языка состоит в том, чтобы работать с информацией из базы данных. В SQL нет многих возможностей обычного процедурного языка. В результате, если разработчики используют SQL, то для организации выполнения программ им приходится думать о совмещении кода базового языка с кодом SQL. Это усложняет разработку программы и отрицательно сказывается на ее производительности.

Ухудшение производительности, которое происходит из-за ограниченных возможностей SQL, привело к тому, что новые версии его международных спецификаций включают новые возможности. Одной из таких возможностей является выражение CASE. С его помощью наконец-то можно построить долгожданную условную структуру. Вторым из нововведений является выражение CAST. Оно помогает преобразовывать табличную информацию из одного типа данных в другой. И наконец, третье нововведение – выражение со значением типа запись – дает возможность работать со списком значений там, где без него можно было работать только с одним значением. Например, если список значений является списком столбцов из таблицы, то теперь можно выполнять какую-либо операцию со всеми этими столбцами, используя для этого очень простой синтаксис.

## Условные выражения GASC

В каждом полноценном компьютерном языке имеется какой-либо условный оператор или условная команда. На самом же деле у большинства языков таких операторов или команд имеется несколько. Вероятно, самой распространенной среди них является структура IF…THEN…ELSE…ENDIF. Если вычисленным значением условия, следующего за ключевым словом IF (если), является True (истина), то выполняется блок команд, который следует за ключевым словом THEN (тогда). А если вычисленное значение условия не равно True, то выполняется блок команд, следующий за ключевым словом ELSE (иначе). О завершении структуры свидетельствует ключевое слово ENDIF (конец IF). Эта структура является прекрасным средством представления в виде кода любой операции, которая может выполняться одним из двух способов. С другой стороны, структура IF…THEN…ELSE…ENDIF меньше подходит для операций, имеющих больше двух вариантов выполнения.

***Помни:***
*В большинстве компьютерных языков имеется оператор CASE, предназначенный для ситуаций, когда требуется в зависимости от некоторых условий направлять выполнение программы по одному из многих имеющихся вариантов*.

В SQL:2003 CASE является выражением, а не оператором. Поэтому CASE в этом языке является только частью оператора, а не оператором как таковым. В SQL выражение CASE можно поместить почти в любом месте, где только может находиться какое-либо значение. Во время выполнения программы для этого выражения вычисляется значение. А для операторов CASE из других языков никакое значение не вычисляется; вместо этого они управляют выполнением программы.

Выражение CASE можно использовать двумя способами.

- **Использовать вместе с условиями поиска**. Выражение CASE отыскивает в таблице такие строки, для которых эти условия выполняются, т.е. являются истинными. Если для какой-либо строки условия являются истинными, то оператор, в состав которого входит выражение CASE, работает с данной строкой.

- **Использовать, чтобы сравнить значение, указанное в этом выражении, с содержимым табличного поля**. Действия, выполняемые оператором, содержащим выражение CASE, зависят от того, какому из нескольких указанных значений соответствует содержимое поля в очередной табличной строке.

Эти понятия станут более ясными после изучения разделов "Использование выражения CASE вместе с условиями поиска" и "Использование выражения CASE вместе со значениями". В первом из этих разделов приводятся два примера использования выражения CASE вместе с условиями поиска. В одном из этих примеров проводится поиск по всей таблице, и на основе имеющихся условий делаются разные изменения табличных значений. Во втором разделе, "Использование выражения CASE вместе со значениями", есть два примера использования этого выражения вместе со значениями.

### Использование выражения CASE с условиями поиска

Эффективным способом использования выражения CASE является проводимый по всей таблице поиск тех строк, в которых выполняется определенное условие поиска. Если использовать выражение CASE таким способом, то у него должен быть следующий синтаксис:

```
CASE
WHEN условие 1 THEN результат1
WHEN условие2 THEN результат2
…
WHEN условие_n THEN результат_n
ELSE результат_x
END
```

Выражение CASE проверяет, является ли истинным **условие1** в первой **оцениваемой строке** (т.е. в первой из тех строк, которые соответствуют условиям предложения WHERE, если только оно имеется). Если да, то выражение CASE принимает значение **результата1**. А если **условие1** не выполняется, строка проверяется на выполнение **условия2**. Если оно выполняется, то выражение CASE принимает значение **результата1** и т.д. А если ни одно из имеющихся условий не выполнено, то CASE принимает значение **результат_х**. Предложение ELSE не является обязательным. В том случае, если этого предложения нет и не выполняется ни одно из указанных условий, выражение принимает значение NULL. После того как оператор SQL, в котором находится выражение CASE, выполнится по отношению к первой оцениваемой строке таблицы и выполнит соответствующее действие, он приступает к следующей строке. Такая последовательность действий продолжается до тех пор, пока не будет закончена обработка всей таблицы.

### Обновление значений на основе условия

Выражение CASE можно поместить почти в любом месте оператора SQL, где только может находиться значение. Поэтому использование этого выражения раскрывает перед вами огромные возможности. Можно использовать CASE внутри оператора UPDATE (обновить), чтобы, например, на основе определенного условия по-разному изменять табличные значения. Проанализируйте следующий пример:

```
UPDATE FOODS
SET RATING = CASE
WHEN FAT < 1
THEN 'очень мало жиров'
WHEN FAT < 5
THEN 'мало жиров'
WHEN FAT < 20
THEN 'среднее количество жиров'
WHEN FAT < 50
THEN 'высокое количество жиров'
ELSE 'сплошные жиры'
```

```
END;
```

Этот оператор проверяет по порядку условия WHEN, пока не встретится первое истинное значение, после чего он игнорирует оставшиеся условия.

В табл. 7.2 было показано содержимое жиров в 100 граммах некоторых продуктов питания. Таблица из базы данных, содержащая эту информацию, может также иметь столбец RATING (оценка), который дает быструю оценку величины содержания жиров. Если запустить предшествующий оператор UPDATE в таблице FOODS (продукты питания) из главы 7, то у спаржи будет оценка "очень мало жиров", у цыплят – "мало жиров", а жареные миндальные орехи попадут в категорию "сплошные жиры".

### Обход условий, вызывающих ошибки

Другим ценным применением выражения CASE является **обход исключений** – проверка условий, которые вызывают ошибки. Проанализируйте выражение CASE, которое определяет размер "компенсации" для продавцов. В компаниях, где работникам компенсируют недополученные комиссионные, часто к комиссионным своих новых работников дают еще и "компенсацию". В следующем примере такую "компенсацию" к своим комиссионным получают новые продавцы, причем по мере роста комиссионных ее размер довольно сильно уменьшается:

```
UPDATE SALES_COMP
SET COMP = COMMISION + CASE
WHEN COMMISSION <> 0
THEN DRAW/COMMISSION
WHEN COMMISSION = 0
THEN DRAW
END;
```

Если у продавца комиссионных нет, то структура этого примера позволяет избежать операции деления на нуль, которая, как известно, приводит к ошибке. А если продавец все же заработал какие-то комиссионные, то ему выплатят эти комиссионные плюс "компенсацию", которая уменьшается пропорционально их размеру.

Все выражения THEN внутри общего выражения CASE должны быть одного и того же типа – или все числовые, или символьные, или даты-времени. Результат выражения CASE имеет тот же тип.

### Использование выражения CASE со значениями

При сравнении проверяемого значения с набором других можно использовать более компактную форму выражения CASE. Эту форму полезно использовать внутри оператора SELECT или UPDATE, когда в столбце таблицы содержится ограниченное число разных значений и нужно связать с каждым из них соответствующее значение результата CASE. Если использовать выражение CASE таким образом, то оно будет иметь следующий синтаксис:

```
CASE значение_n
WHEN значенив1 THEN результат1
WHEN значение2 THEN результат2
…
WHEN значение_n THEN результат_n
ELSE результат_x
END
```

Если проверяемое значение **(значение_n)** равно **значению1**, то выражение принимает значение **результат1**. А если **значение_n** не равно **значению1**, а **значению2**, то выражение принимает значение **результат2**. Все значения, предназначенные для сравнения, проверяются сверху вниз, по направлению к **значению_n**, пока не будет найдено то из них, которое равно **значению_n.** Если же такое значение найдено не будет, то выражение принимает значение **результат_x**. И опять, если необязательное предложение ELSE отсутствует и ни одно из значений, предназначенных для сравнения, не равно проверяемому, то выражение принимает неопределенное значение.

Чтобы понять, как работает форма CASE со значениями, проанализируйте пример с таблицей, в которой находятся фамилии и звания офицеров. Требуется получить их список, в котором перед фамилиями офицеров стояли бы аббревиатуры их званий. Для сравнения будут использоваться такие звания: генерал (general), полковник (colonel), подполковник (lieutenant colonel), майор (major), капитан (captain), старший лейтенант (first lieutenant), лейтенант (second lieutenant). И наконец, тот, у кого какое-либо другое звание, в списке будет просто назван "господином" (Mr.). Список создается с помощью следующего оператора:

```
SELECT CASE RANK
WHEN ' general' THEN 'Gen.'
WHEN 'colonel' THEN 'Col.'
WHEN 'lieutenant colonel' THEN 'Lt. Col.
WHEN 'major' THEN 'Maj.'
WHEN ' captain' THEN 'Capt. '
WHEN 'first lieutenant' THEN '1st. Lt.
WHEN 'second lieutenant' THEN '2nd. Lt.
ELSE 'Mr. '
END,
LAST_ NAME
FROM OFFICERS;
Результат должен быть примерно такой:
Capt. Midnight
Col. Sanders
Gen. Schwarzkopf
Maj. Disaster
Mr. Nimitz
```

Честер Нимиц был адмиралом во флоте Соединенных Штатов во время Второй мировой войны. Так как в выражении CASE его звания нет, то оно определяется предложением ELSE.

Вот еще пример. Предположим, что капитан Миднайт получает повышение в звании и становится майором. Требуется сделать соответствующие изменения в базе данных OFFICERS (офицеры). Предположим, что в переменной officerjastjiame (фамилия офицера) находится значение 'Midnight', а в переменной new_rank (новое звание) – целое значение (4), которое, согласно следующей таблице, соответствует новому званию Миднайта.

Тогда ввести данные о повышении можно с помощью следующего кода SQL:

```
UPDATE OFFICERS
SET RANK = CASE:new_rank
WHEN 1 THEN 'general'
WHEN 2 THEN 'colonel'
WHEN 3 THEN ' lieutenant colonel'
WHEN 4 THEN 'major'
WHEN 5 THEN 'captain'
WHEN 6 THEN 'first lieutenant'
WHEN 7 THEN 'second lieutenant'
WHEN 8 THEN'Mr. '
END
WHERE LAST_NAME =:officer_last_name;
```

| new_rank | Звание |
|---|---|
| 1 | general |
| 2 | colonel |
| 3 | lieutenant colonel |
| 4 | major |
| 5 | captain |
| 6 | first lieutenant |
| 7 | second lieutenant |
| 8 | Mr. |

Для выражения CASE со значениями есть еще один синтаксис:

```
CASE
WHEN значение_n = значение1 THEN результат1
WHEN значвние_n = значение2 THEN результат2
…
WHEN значение_n = значвние_n THEN результат_n
ELSE результат_x
END
```

### Специальное выражение CASE – NULLIF

Время от времени предметы переходят из одного известного состояния в другое. Иногда вам кажется, что вы что-то знаете, но в конце концов выясняется, что это не так. Классическая термодинамика, как и современная теория хаоса, утверждает, что системы легко переходят из хорошо известного, упорядоченного состояния в состояние хаоса, которое никто не может предсказать. Кто когда-либо видел состояние комнаты подростка через неделю после того, как в ней была генеральная уборка, тот может поручиться за точность этих теорий.

В таблицах из базы данных точно определенные значения находятся в тех полях, в которых имеются известные данные. А если значение поля неизвестно, то в этом поле обычно находится неопределенное значение (NULL). SQL дает возможность с помощью выражения CASE менять определенное значение табличного поля на неопределенное. Значение NULL означает, что значение поля вам больше не известно.

Представьте, что вы владеете небольшой авиакомпанией, которая выполняет рейсы между Южной Калифорнией и штатом Вашингтон. До недавних пор во время некоторых из рейсов делалась промежуточная посадка в международном аэропорту Сан-Хосе для дозаправки. Затем вы лишились разрешения на полеты в Сан-Хосе. Теперь дозаправку приходится проводить в одном из двух международных аэропортов: или Сан-Франциско, или Окленда. Так что сейчас вам точно не известно, во время какого рейса в каком из аэропортов будут садиться ваши самолеты, ясно лишь, что не в Сан-Хосе. У вас имеется база данных FLIGHT (полет), в которой находится важная информация о каждом из ваших рейсов (в том числе данные в столбце RefuelStop (остановка для дозаправки)). Теперь эту базу нужно обновить, чтобы удалить из нее все упоминания о Сан-Хосе. Один из способов это сделать показан в следующем примере:

```
UPDATE FLIGHT
SET RefuelStop = CASE
WHEN RefuelStop = 'San Jose1
THEN NULL
ELSE RefuelStop
END;
```

Так как подобные ситуации, когда нужно заменить известное значение неопределенным (NULL), бывают часто, то для выполнения этой задачи в SQL имеется специальный упрощенный синтаксис. Вот как выглядит предыдущий пример, переписанный в таком упрощенном виде:

```
UPDATE FLIGHT
SET RefuelStop = NULLIF(RefuelStop, 'San Jose');
```

Это выражение можно прочитать следующим образом: "Обновить базу данных FLIGHT, заменяя в столбце RefuelStop значение San Jose на NULL. Другие значения менять не надо".

Синтаксис NULLIF еще более удобен, если нужно преобразовать данные, накопленные в таком виде, который позволял с ними работать с помощью программы, написанной на стандартном языке программирования, например на COBOL или Fortran. В стандартных

языках программирования значение NULL не используется, поэтому очень часто для выражения понятия "не известен" или "не применяется" используются специальные значения. Например, значение "не известен" в поле SALARY (зарплата) может быть представлено числом -1, а значение "не известен" или "не применим" в поле JOBCODE (код задания) – символьной строкой "***". Если необходимо в SQL-совместимой базе данных представить состояния "не известен" или "не применим" с помощью значения NULL, то специальные значения придется преобразовать в неопределенные. В следующем примере эта операция выполняется для таблицы с данными о сотрудниках, в которой некоторые значения окладов неизвестны:

```
UPDATE EMP
SET Salary = CASE Salary
WHEN -1 THEN NULL
ELSE Salary
END;
```

Это преобразование удобнее выполнять, используя выражение NULLIF:

```
UPDATE EMP
SET Salary = NULLIF(Salary, -1);
```

### Еще одно специальное выражение CASE – COALESCE

COALESCE (объединять), как и NULLIF, является упрощенной формой специального выражения CASE. COALESCE работает со списками значений, которые могут быть как определенными, так и неопределенными. Если в списке только одно из значений не является NULL, то оно и становится значением выражения COALESCE. Если же в списке таких значений больше, чем одно, то значением выражения становится первое из них. Когда все значения списка – NULL, то значением выражения COALESCE также становится NULL.

Выражение CASE, выполняющее те же действия, имеет следующий вид:

```
CASE
WHEN значение 1 IS NOT NULL
THEN значение1
WHEN значение2 IS NOT NULL
THEN значение2
…
WHEN значение_n IS NOT NULL
THEN значение_n
ELSE NULL
END
```

А соответствующий упрощенный синтаксис COALESCE выглядит так:

```
COALESCE(значение1, значение2,…, значение_n)
```

Возможно, вам придется использовать выражение COALESCE после выполнения операции OUTER JOIN (о которой рассказывается в главе 10). Этот оператор позволяет сократить объем вводимого кода.

### Преобразование типов данных с помощью выражения CAST

В главе 2 рассказывалось о различных типах данных, используемых при работе с SQL. В идеальном случае каждый столбец таблицы должен иметь подходящий тип. Однако в действительности не всегда ясно, каким же он должен быть. Предположим, определяя для базы данных таблицу, вы присваиваете столбцу тип данных, который замечательно подходит для вашего нынешнего приложения. Однако позднее вам, возможно, потребуется расширить поле деятельности вашего приложения или написать полностью новое приложение, в котором данные используются по-другому. Для этого нового использования может потребоваться тип данных, который отличается от выбранного вами ранее.

Возможно, вам потребуется сравнить столбец одного типа, находящийся в одной таблице, со столбцом другого типа из другой таблицы. Например, в одной таблице даты могут храниться в виде символьных данных, а в другой – в виде значений типа DATE. Даже если в обоих столбцах находятся одни и те же элементы данных, например даты, их разные типы могут не позволить сделать сравнение. Для SQL-86 и SQL-89 несовместимость типов данных представляет большую проблему. Однако с появлением SQL-92 появилось и удобное ее решение – выражение CAST (приведение).

Выражение CAST преобразует табличные данные или базовые переменные одного типа в другой. После такого преобразования можно выполнять необходимые операцию или анализ.

Используя выражение CAST, вы, естественно, столкнетесь с некоторыми ограничениями. Нельзя без разбора преобразовать данные одного типа в любой другой. Преобразуемые данные должны быть совместимы с новым типом. Например, можно использовать выражение CAST для преобразования в тип DATE символьной строки '1998-04-26', имеющей тип данных CHAR(IO). Однако символьную строку 'rhinoceros' (носорог), также имеющую тип данных CHAR(IO), преобразовывать с помощью CAST в тип DATE уже нельзя. Нельзя преобразовать значение типа INTEGER в значение типа SMALLINT, если размер первого из них превышает размер, максимально допустимый для SMALLINT.

Элемент данных любого из символьных типов можно преобразовать в любой другой тип (например, числовой или даты) при условии, что значение этого элемента имеет вид литерала нового типа. И наоборот, элемент данных любого типа можно преобразовать в любой из символьных типов в виде литерала исходного типа.

Другие возможные преобразования перечислены ниже.

- Любой числовой тип – в любой другой числовой. При преобразовании в тип с меньшей дробной частью система округляет результат или отбрасывает в нем лишние цифры.
- Любой точный числовой тип – в интервал, состоящий из одного компонента, такой, например, как INTERVAL DAY или INTERVAL SECOND.
- Любой тип DATE – в TMESTAMP. В полученном значении типа TTMESTAMP та часть, которая предназначена для времени, будет заполнена нулями.
- Любой тип TIME – в тип TIME с другой точностью дробной части или в TIMESTAMP. Часть, предназначенная для даты в TIMESTAMP, заполняется значением текущей даты.
- Любой тип TIMESTAMP – в DATE, TIME или TIMESTAMP с другой точностью дробной части.
- Любой тип INTERVAL года-месяца – в точный числовой тип или в INTERVAL года-месяца с другой точностью ведущего поля.
- Любой тип INTERVAL дня-времени – в точный числовой тип или в INTERVAL дня-времени с другой точностью ведущего поля.

### Использование выражения CAST внутри SQL-кода

Предположим, что вы работаете для торговой компании, которая собирает данные о своих предполагаемых сотрудниках, а также о сотрудниках, которых вы уже наняли. Данные о предполагаемых сотрудниках вы разместили в таблице PROSPECT (потенциальный сотрудник), а различаете каждого из них по его номеру социального страхования (Social Security Number, SSN), который вы храните в виде данных типа CHAR(9). Данные о работающих сотрудниках вы разместили в другой таблице, EMPLOYEE (сотрудник), и различаются они также по своему номеру социального страхования, но имеющему уже тип INTEGER. Теперь вам требуется получить список людей, данные о которых содержатся в обеих таблицах. Чтобы выполнить эту задачу, используйте выражение CAST:

```
SELECT * FROM EMPLOYEE
```

```
WHERE EMPLOYEE.SSN =
CAST(PROSPECT.SSN AS INTEGER);
```

### Использование выражения CAST при взаимодействии SQL и базового языка

Главное предназначение выражения CAST состоит в том, чтобы работать с такими типами данных, которые есть в SQL, но отсутствуют в базовом языке. Вот некоторые примеры таких типов.

- Fortran и Pascal не имеют типов данных DECIMAL и NUMERIC.
- Стандартный COBOL не содержит типов данных FLOAT и REAL.
- Ни в каком языке, кроме SQL, нет типа данных DATETIME.

Предположим, что для доступа к таблицам, у которых есть столбцы с типом данных DECIMAL (5.3), вам нужен язык Fortran или Pascal. При этом требуется избежать неточности, которая может случиться, если переводить значения из этих столбцов в тип данных REAL, используемый в этих двух языках. Эту задачу можно выполнить, используя CAST для перевода данных в базовые переменные (и из базовых переменных), которые имеют тип символьной строки. Например, числовое значение 198.37 переводится в значение '0000198.37' типа CHAR(IO). Вначале в тип CHAR(IO) с помощью CAST преобразуются те данные типа DECIMAL (5.3), которые относятся к сотруднику с идентификационным номером, находящимся в базовой переменной:emp_id_var:

```
SELECT CAST(Salary AS CHAR(10)) INTO:salary_var
FROM EMP
WHERE EmpID =:emp_id_var;
```

Затем приложение проверяет появившуюся в переменной:salary_var символьную строку и присваивает этой переменной новое значение, а затем с помощью следующего кода SQL обновляет базу данных:

```
UPDATE EMP
SET Salary = CAST(:salary_var AS DECIMAL(5.3))
WHERE EmpID =:emp_id_var;
```

С символьными строками, такими как '000198.37', работать в языках Fortran и Pascal очень трудно, но для выполнения в этих языках нужных операций можно написать набор специальных процедур. В принципе, в любом базовом языке можно получать и обновлять любые SQL-данные, а также получать и задавать точные значения.

Вообще говоря, выражение CAST лучше всего подходит для преобразования типов данных базового языка в типы данных базы и наоборот, а не для преобразования одних типов данных базы в другие.

### Выражения со значением, типа записи

В SQL-86 и SQL-89 большинство операций выполнялось с одиночным значением или с одиночным столбцом из табличной строки. Чтобы работать с множеством значений, необходимо с помощью логических связок создать сложные выражения. (О логических связках вы узнаете в главе 9.)

С появлением SQL-92 в языке SQL появились **выражения со значением типа записи**, которые работают не с одиночными значениями или столбцами, а со списками значений или столбцов. Такое выражение является списком выражений со значением, заключенных в апострофы и отделяемых друг от друга запятыми. Можно работать сразу со всем значением типа записи или только с нужным его подмножеством.

В главе 6 рассказывалось, как с помощью оператора INSERT добавлять в таблицу новую строку. Чтобы это сделать, используется выражение со значением типа записи. Рассмотрите следующий пример. Данные сыра чеддер вводятся в поля FOODNAME

(название продукта питания), CALORIES (калории), PROTEIN (белки), FAT (жиры), CARBOHYDRATE (углеводы) из таблицы FOODS (продукты питания).

```
INSERT INTO FOODS
(FOODNAME, CALORIES, PROTEIN, FAT, CARBOHYDRATE)
VALUES
('Сыр чеддер', 398, 25, 32.2, 2.1);
```

В этом примере выражением со значением типа записи является ('Сыр чеддер', 398, 25, 32.2, 2.1). Если таким образом в операторе INSERT использовать выражение со значением типа записи, в этом выражении могут быть значения NULL и значения по умолчанию. (**Значением по умолчанию** называется то, которое должно быть в табличном столбце, если явно не указано другое значение.) Вот, например, вполне соответствующее правилам выражение со значением типа записи:

```
('Сыр чеддер', 398, NULL, 32.2, DEFAULT).
```

Можно добавить в таблицу сразу множество строк, вставив в предложение VALUES множество выражений со значением типа записи. Например, так выглядит ввод в таблицу FOODS данных о таких продуктах: салат-латук, маргарин, горчица и спагетти:

```
INSERT INTO FOODS
(FOODNAME, CALORIES, PROTEIN, FAT, CARBOHYDRATE)
VALUES
('Салат-латук', 14, 1.2, 0.2, 2.5),
('Маргарин', 720, 0.6, 81.0, 0.4),
('Горчица', 75, 4.7, 4.4, 6.4),
('Спагетти', 148, 5.0, 0.5, 30.1);
```

Выражения со значением типа записи можно использовать, чтобы сэкономить место при вводе кода, предназначенного для сравнения. Предположим, у вас имеются две таблицы сданными о продуктах питания, одна из которых составлена на английском языке, а другая– на испанском. Требуется найти те строки первой таблицы, которые точно соответствуют строкам второй. Если не использовать выражения со значением типа записи, то, возможно, придется создавать такой код:

```
SELECT * FROM FOODS
WHERE FOODS.CALORIES = COMIDA.CALORIA
AND FOODS.PROTEIN = COMIDA.PROTEINA
AND FOODS.FAT = COMIDA.GORDO
AND FOODS.CARBOHYDRATE = COMIDA.CARBOHIDRATO;
```

А вот код, выполняющий те же действия, но составленный с помощью выражения со значением типа записи:

```
SELECT * FROM FOODS
WHERE (FOODS.CALORIES, FOODS.PROTEIN, FOODS.FAT,
FOODS.CARBOHYDRATE)
=
(COMIDA.CALORIA, COMIDA.PROTEINA, COMIDA.GORDO,
COMIDA.CARBOHIDRATO);
```

В этом примере при вводе кода было сэкономлено не слишком много места. Больший выигрыш будет, если число сравниваемых столбцов будет больше. В крайнем случае (как, в частности, в этом примере), видимо, лучше придерживаться старого синтаксиса, так как он более понятен.

Впрочем, использование выражения со значением типа записи имеет дополнительное преимущество перед его аналогом с развернутым синтаксисом. Дело в том, что это выражение работает намного быстрее. В принципе, какая-либо очень умная реализация может анализировать развернутый синтаксис и выполнять его как выражение со значением типа записи. Однако на практике на такую хитроумную оптимизацию пока что не способна ни одна из имеющихся СУБД.

# "Пристрелка" к нужным данным

В этой главе…

- Указание требуемых таблиц
- Отделение нужных строк от всех остальных
- Создание эффективных предложений where
- Как работать со значениями null
- Создание составных выражений с логическими связками
- Группирование вывода результата запроса по столбцу
- Упорядочение результата запроса

---

У системы управления базой данных есть две главные обязанности: хранение данных и обеспечение к ним легкого доступа. В хранении данных нет ничего особенного, ту же работу может выполнять и шкаф. А что действительно трудно – так это обеспечить удобный доступ к данным. Для этого необходимо иметь возможность выловить небольшое количество нужных данных из океана ненужных.

SQL позволяет использовать некоторые характеристики самих данных для определения, представляют ли они для вас интерес. Операторы SELECT (выбрать), DELETE (удалить) и UPDATE (обновить) сообщают **ядру** (**engine**) базы данных, т.е. той части СУБД, которая как раз и взаимодействует с данными, какие именно строки необходимо выбрать, удалить или обновить. Чтобы обрабатывались требуемые строки, в операторы SELECT, DELETE и UPDATE добавляются уточняющие предложения.

### Уточняющие предложения

В SQL имеются следующие уточняющие предложения: FROM, WHERE, HAVING, GROUP BY и ORDER BY. Предложение FROM (из) сообщает ядру базы данных, с какой таблицей (или таблицами) он должен работать. Что касается WHERE (где) и HAVING (при условии), то эти предложения указывают характеристику, определяющую необходимость выполнения текущей операции над конкретной строкой. И наконец, предложения GROUP BY (группировать по) и ORDER BY (упорядочивать по) указывают, каким образом следует выводить строки, полученные из базы данных. Основные сведения по уточняющим предложениям приведены в табл. 9.1.

**Таблица 9.1**. Уточняющие предложения и их назначение.

| Уточняющее предложение | Для чего предназначено |
|---|---|
| **from** | Указывает, из каких таблиц брать данные |
| **where** | Фильтрует строки, которые не соответствуют условию поиска |
| **group by** | Группирует строки в соответствии со значениями в столбцах группирования |
| **having** | Фильтрует группы, которые не соответствуют условию поиска |
| **order by** | Сортирует результаты предыдущих предложений перед получением окончательного вывода |

Если используется больше чем одно из этих предложений, то они должны располагаться в следующем порядке:

```
SELECT список_столбцов
FROM список_таблиц
[WHERE условие_поиска]
[GROUP BY столбец_группирования]
[HAVING условие_поиска]
[ORDER BY условие_упорядочивания];
```

Ниже описывается, как работают уточняющие предложения.

- Предложение WHERE – это фильтр, который выбирает строки, удовлетворяющие условию поиска, и отбрасывает все остальные.
- Предложение GROUP BY создает группы из строк, отобранных с помощью предложения WHERE, каждая из которых соответствует какому-либо значению из столбца группирования.
- Предложение HAVING – это другой фильтр, который обрабатывает каждую из групп, созданных с помощью предложения GROUP BY, и выбирает те из них, которые удовлетворяют условию поиска, отбрасывая все остальные.
- Предложение ORDER BY сортирует все, что остается после того, как все предыдущие предложения проведут обработку таблицы (или таблиц).

Квадратные скобки (**[ ]**) означают, что предложения WHERE, GROUP BY, HAVING и ORDER BY не являются обязательными.

SQL выполняет эти предложения в следующем порядке: FROM, WHERE, GROUP BY, HAVING и SELECT. Предложения работают по принципу конвейера, когда каждое из них получает результат выполнения предыдущего предложения, обрабатывает этот результат и передает то, что получилось, следующему предложению. Если этот порядок выполнения переписать в виде функций, то он будет выглядеть следующим образом:

```
SELECT (HAVING (GROUP BY (WHERE (FROM…))))
```

Предложение ORDER BY выполняется уже после SELECT. Оно может обращаться только к тем столбцам, которые перечислены в списке, находящемся после SELECT. К другим же столбцам из таблиц, перечисленных в предложении FROM, предложение ORDER BY обращаться не может.

### Предложения From

Предложение FROM легко понять, если в нем указана только одна таблица, как, например, в следующем примере:

```
SELECT * FROM SALES;
```

Этот оператор возвращает все данные, находящиеся во всех строках каждого столбца таблицы SALES (продажи). Впрочем, в предложении FROM можно указывать больше, чем одну таблицу. Например:

```
SELECT *
```

```
FROM CUSTOMER, SALES;
```

Этот оператор создает виртуальную таблицу, в которой данные из таблицы CUSTOMER (покупатель) объединены с данными из таблицы SALES. Для создания новой таблицы каждая строка из CUSTOMER объединяется с каждой строкой из SALES. Поэтому в создаваемой таким способом новой виртуальной таблице количество строк равно количеству строк первой таблицы, умноженному на количество строк второй. И если в CUSTOMER десять строк, а в SALES – сто, то в новой таблице их будет тысяча.

Такая операция называется **декартовым произведением** двух исходных таблиц. Декартово произведение на самом деле является разновидностью операции объединения (JOIN). Об операциях объединения подробно рассказывается в главе 10.

В большей части приложений большинство тех строк, которые созданы в результате применения к двум таблицам декартова произведения, не имеют никакого смысла. Что касается виртуальной таблицы, созданной из CUSTOMER и SALES, то в ней представляют интерес только строки, в которых значение CustomerlD (идентификатор покупателя) из таблицы CUSTOMER равно значению CustomerlD из таблицы SALES. Все остальные строки можно отфильтровать с помощью предложения WHERE.

### Предложения WHERE

В этой книге предложение WHERE использовалось много раз без всякого объяснения, потому что его значение и способ использования очевидны. Оператор выполняет операцию (такую как SELECT, DELETE или UPDATE) только с теми табличными строками, для которых определенное условие истинно. У предложения WHERE такой синтаксис:

```
SELECT список_столбцов
FROM имя_таблицы
WHERE условие;
DELETE FROM имя_таблицы
WHERE условие;
UPDATE имя_таблицы
SET столбец1=значение1, столбец2=значение2,…,
столбец_n=значение_n
WHERE условие;
```

Во всех случаях условие в предложении WHERE может быть или простым, или сколь угодно сложным. Чтобы из множества условий создать одно, их можно соединить друг с другом при помощи логических связок AND, OR и NOT. В этой главе мы еще вернемся к ним.

Вот некоторые типичные примеры предложений WHERE.

```
WHERE CustomerlD = SALES.CustomerID
WHERE FOODS.Calories = COMIDA.Caloria
WHERE FOODS.Calories < 219
WHERE FOODS.Calories > 3 * base_value
WHERE FOODS.Calories < 219 AND FOODS.Protein > 27.4
```

Условия, выражаемые предложениями WHERE, называются предикатами. **Предикат** – это выражение, которое утверждает факт, относящийся к значениям из этого выражения.

Например, предикат FOODS.Calories < 219 является истинным, если в текущей строке значение столбца FOODS.Calories меньше 219. Если утверждение является истинным, то оно удовлетворяет условию. Утверждение может быть истинным (т.е. его значение равно True), ложным (его значение равно False) или с неопределенным логическим значением. Последний случай бывает тогда, когда в утверждении какие-либо элементы имеют значение NULL. Наиболее распространенными являются предикаты сравнения (=, <, >, <>, <= и >=), но в SQL имеются и некоторые другие, которые значительно увеличивают возможности "отфильтровывать" требуемые данные от всех остальных. Ниже приведен список с предикатами, предоставляющими такую возможность.

- Предикаты сравнения.
- BETWEEN.
- IN [NOT IN].

- LIKE [NOT LIKE].
- NULL.
- ALL, SOME, ANY.
- EXISTS.
- UNIQUE.
- OVERLAPS.
- MATCH.
- SIMILAR.
- DISTINCT.

### Предикаты сравнения

Примеры, приведенные в предыдущем разделе, демонстрируют обычное использование предикатов сравнения, в которых одно значение сравнивается с другим. Каждая строка, где в результате сравнения получается значение True, выполняет условие предложения WHERE, и с ней выполняется определенная операция (SELECT, UPDATE, DELECT и т.д.). Строки, где в результате сравнения получается значение False, пропускаются. Проанализируйте, например, следующий оператор SQL:

```
SELECT * FROM FOODS
WHERE Calories < 219;
```

Этот оператор выводит все строки таблицы FOODS, в которых значение, хранящееся в столбце Calories, меньше 219.

В табл. 9.2 приведены шесть предикатов сравнения.

**Таблица 9.2**. Предикаты сравнения языка SQL.

| Сравнение | Символ |
|---|---|
| Равно | = |
| Не равно | <> |
| Меньше | < |
| Меньше или равно | <= |
| Больше | > |
| Больше или равно | >= |

### BETWEEN

Иногда нужно выбрать ту строку, в которой значение какого-либо столбца входит в определенный диапазон. Один из способов это сделать – использовать предикаты сравнения. Можно, например, составить предложение WHERE, предназначенное для выбора всех строк таблицы FOODS, в которых значение, хранящееся в столбце CALORIES, больше 100 и меньше 300:

```
WHERE FOODS.Calories > 100 AND FOODS.Calories < 300
```

В это сравнение не включены продукты питания, содержащие в точности 100 или 300 калорий, – в нем имеются только те значения, которые находятся в промежутке между этими числами. Чтобы в сравнение попали и эти два значения, можно написать следующий оператор:

```
WHERE FOODS.Calories >= 100 AND FOODS.Calories <= 300
```

Другой способ указать диапазон, включая его границы, – использовать предикат BETWEEN (между):

```
WHERE FOODS.Calories BETWEEN 100 AND 3 00
```

Это предложение выполняет абсолютно те же функции, что и оператор в предыдущем примере, в котором используются предикаты сравнения. Как видите, эта формулировка позволяет ввести меньшее количество кода, а также она чуть более наглядная, чем та, в которой используются два предиката сравнения, соединенные логической связкой AND.

***Внимание:***
*Ключевое слово BETWEEN может привести к путанице, потому что неочевидно, включены ли в предложение границы диапазона. На самом деле границы в предложение включены. Кроме того, первая граница обязательно должна быть не больше второй. Если, например, в FOODS.Calories содержится значение 200, то следующее предложение возвращает значение True:*

```
WHERE FOODS.Calories BETWEEN 100 AND 300
```

Однако предложение, казалось бы, эквивалентное предыдущему примеру, на самом деле возвращает противоположный результат False:

```
WHERE FOODS.Calories BETWEEN 300 AND 100
```

***Помни:***
*Для ключевого слова BETWEEN первая граница обязательно должна быть не больше второй*.

Предикат BETWEEN можно использовать со следующими типами данных: символьными, битовыми, даты-времени, а также с числовыми. Вам могут встретиться примеры, похожие на следующий:

```
SELECT FirstName, LastName
FROM CUSTOMER
WHERE CUSTOMER.LastName BETWEEN 'A' AND 'Mzzz';
```

При его выполнении возвращаются данные обо всех покупателях, фамилии которых находятся в верхней половине списка по алфавиту.

## IN и NOT IN

Предикаты IN (в) и NOT IN (не в) используются для работы с любыми указанными значениями, такими, например, как OR (Орегон), WA (Вашингтон) и ID (Айдахо), из определенного набора значений, а именно штатов США. Например, у вас имеется таблица, в которой перечислены поставщики товаров, регулярно закупаемых вашей компанией. Вам нужно узнать телефонные номера тех поставщиков, которые размещаются в северной части Тихоокеанского побережья. Эти номера можно найти с помощью предикатов сравнения, например таких, которые показаны в следующем примере:

```
SELECT Company, Phone
FROM SUPPLIER
WHERE State = 'OR' OR State = 'WA' OR State = 'ID';
```

Впрочем, для выполнения той же самой задачи можно также использовать предикат IN:

```
SELECT Company, Phone
FROM SUPPLIER
WHERE State IN ('OR1, 'WA1, 'ID');
```

Это чуть более компактная форма записи, чем та, в которой используются предикаты сравнения и логические OR.

Таким же образом работает и второй вариант этого предиката – NOT IN. Скажем, у вас есть представительства в штатах Калифорния, Аризона и Нью-Мексико. Чтобы избежать уплаты налога с продаж, вы обдумываете возможность работы только с теми поставщиками, чьи представительства находятся за пределами этих трех штатов. Используйте следующую конструкцию:

```
SELECT Company, Phone
```

```
FROM SUPPLIER
WHERE State NOT IN CCA1, 'AZ', 'NM');
```

Используя таким образом ключевое слово IN, можно вводить чуть меньше кода. Впрочем, это не такое уж и большое преимущество. Как показано в первом примере этого раздела, примерно столько же придется потрудиться над кодом, если использовать предикаты сравнения.

***Совет:***

*Даже если предикат IN дает возможность вводить не намного меньше кода, то у вас все равно есть еще одна веская причина, чтобы использовать именно IN, а не предикаты сравнения. Ваша СУБД, скорее всего, поддерживает оба этих метода, и один из них может работать значительно быстрее другого. Эти два метода включения или исключения можно испытать в работе, а затем использовать тот из них, который покажет самый быстрый результат. А если в СУБД имеется хороший оптимизатор, то, независимо от вида используемых вами предикатов, во время работы, скорее всего, будет выбран самый оптимальный метод. Испытание этих двух методов как раз покажет, насколько хорошо оптимизирует код используемая СУБД. Если между временем выполнения двух операторов имеется значительная разница, то качество оптимизации под вопросом*.

Кроме того, ключевое слово IN представляет ценность и в другой области. Если оно является частью подзапроса, то для получения результатов, которые нельзя добыть в одной таблице, ключевое слово IN позволяет выбрать информацию уже из двух таблиц. О подзапросах рассказывается в главе 11, а сейчас мы рассмотрим пример того, как в них используется ключевое слово IN.

Предположим, что вам нужно вывести имена всех тех, кто за последние 30 дней купил товар F-117A. Имена и фамилии покупателей находятся в таблице CUSTOMER (покупатель), а данные о сделках– в таблице TRANSACT (заключение сделок). Для этого вы сможете использовать следующий запрос:

```
SELECT FirstName, LastName
FROM CUSTOMER
WHERE CustomerlD IN
(SELECT CustomerlD
FROM TRANSACT
WHERE ProductID = 'F-117A'
AND TransDate >= (CurrentDate -30));
```

В первой из этих таблиц используются поля CustomerlD (идентификатор покупателя), FirstName (имя), LastName (фамилия), а во второй – CustomerlD, ProductID (идентификатор товара) и TransDate (дата сделки). Кроме того, используется переменная CurrentDate (текущая дата). Внутренний оператор SELECT, работающий с таблицей TRANSACT, вложен во внешний оператор SELECT, работающий с таблицей CUSTOMER. Первый из них ищет в столбце CustomerlD номера всех тех, кто за последние 30 дней купил товар F-l 17A. А внешний оператор SELECT выводит имена и фамилии всех покупателей, номера которых получены с помощью внутреннего оператора SELECT.

## LIKE и NOT LIKE

Для сравнения двух символьных строк, чтобы выяснить их частичное соответствие друг другу, можно использовать предикат LIKE (похожий). Частичное соответствие представляет ценность тогда, когда о разыскиваемой строке что-то все-таки известно, но не известно, как она в точности выглядит. Кроме того, частичные соответствия можно использовать, чтобы получить из таблицы множество строк, в которых один из столбцов содержит похожие друг на друга символьные строки.

Чтобы указать частичные соответствия, в SQL используются два символа-маски (**wildcard character**). Знак процента (**%**) означает любую строку, состоящую из любого количества символов (в том числе и равного нулю). Символ подчеркивания (_) означает любой одиночный символ. Некоторые примеры того, как можно использовать предикат LIKE, показаны в табл. 9.3.

**Таблица 9.3**. Предикат like, используемый в SQL.

| Выражение | Возвращаемые значения |
|---|---|
| WHERE WORD LIKE 'intern%' | intern |
| | internal |
| | international |
| | internet |
| | interns |
| WHERE WORD LIKE '%Peace%' | Justice of the Peace |
| | Peaceful Warrior |
| WHERE WORD LIKE 't_p_' | tape |
| | taps |
| | tipi |
| | tips |
| | tops |
| | type |

Предикат NOT LIKE (не похожий) дает возможность получить все строки, которые не удовлетворяют частичному соответствию, имеющему, как в следующем примере, не менее одного символа-маски:

```
WHERE PHONE NOT LIKE '503%'
```

В этом случае будут возвращены все строки таблицы, в которых телефонный номер, содержащийся в столбце PHONE (телефон), не начинается с 503.

***Совет:***

*Возможно, вам потребуется выполнить поиск строки, в которой находится знак процента или символ подчеркивания. В таком случае необходимо, чтобы SQL интерпретировал, например, знак процента как знак процента, а не как символ-маску. Проводить такой поиск можно, если перед символом, который при поиске должен восприниматься буквально, ввести управляющий символ. В качестве такого символа можно назначить любой символ, лишь бы его не было в проверяемой строке. Как это сделать, показано в следующем примере:*

```
SELECT Quote
FROM BARTLETTS
WHERE Quote LIKE '20#%'
ESCAPE '#';
```

Символ **%** превращается в обычный из символа-маски с помощью стоящего перед ним символа #. Точно таким же способом можно отключить и символ подчеркивания, а также сам управляющий символ. Например, предшествующий запрос должен найти такую цитату из "Bartlett's Familiar Quotations" (Известные цитаты Бартлетта):

```
20% of the salespeople produce 80% of the results.
```

Данный запрос найдет также следующее:

```
20%
```

### SIMILAR

Вместе с SQL: 1999 появился и предикат SIMILAR (подобный), который позволяет находить частичное соответствие более эффективно, чем это делает предикат LIKE. С помощью предиката SIMILAR можно сравнить символьную строку с регулярным выражением. Скажем, например, вы просматриваете в таблице программной совместимости столбец OperatingSystem (операционная система), чтобы проверить совместимость с Microsoft Windows. Можно составить примерно такое предложение WHERE:

```
WHERE OperatingSystem SIMILAR TO
'(Windows (3.1195|98|Millenium Edition|CE|NT|2 000|XP)) '
```

В результате выполнения этого предиката будут возвращены все строки, у которых в столбце OperatingSystem содержится любая из указанных операционных систем Microsoft.

### NULL

С помощью предиката NULL выполняется поиск всех тех строк, в которых выбранный столбец содержит неопределенное значение. Именно такие значения имелись в столбце Carbohydrate (углеводы) в нескольких строках таблицы FOODS (продукты питания) (см. главу7). Названия продуктов из этих строк можно получить с помощью такого оператора:

```
SELECT (FOOD)
FROM FOODS
WHERE Carbohydrate IS NULL;
```

Этот запрос возвращает следующие значения:

```
Гамбургер с нежирной говядиной
Нежное мясо цыплят
Жареный опоссум
Свиной окорок
```

Как вы, возможно, и предполагаете, если вставить ключевое слово NOT (нет), то получится совершенно противоположный результат:

```
SELECT (FOOD)
FROM FOODS
WHERE Carbohydrate IS NOT NULL;
```

Этот запрос возвращает все строки таблицы FOODS, за исключением тех четырех, которые были выведены предыдущим запросом.

***Внимание:***
*Выражение Carbohydrate IS NULL– это не то же самое, что CARBOHYDRATE = NULL. Для иллюстрации этого утверждения предположим, что в текущей строке таблицы FOODS значения в столбцах Carbohydrate и Protein (белки) являются неопределенными. Из этого можно сделать несколько выводов*.

- Carbohydrate IS NULL истинно.
- Protein IS NULL истинно.
- Carbohydrate IS NULL AND Protein IS NULL истинно.
- Истинность Carbohydrate = Protein не может быть определена.
- Carbohydrate = NULL является недопустимым выражением.

Использовать ключевое слов NULL в сравнениях бессмысленно, так как всегда возвращается ответ NULL.

Почему же истинность выражения Carbohydrate = Protein определяется как неизвестная, даже если Carbohydrate и Protein имеют одно и то же неопределенное значение? Да потому, что NULL просто означает "я не знаю". Вы же не знаете, каким должно быть значение в Carbohydrate, и не знаете, каким – в Protein. Следовательно, вам не известно, являются ли эти неизвестные значения одинаковыми. Возможно, в Carbohydrate следует ввести 37, а в

Protein – 14, а может, у каждого из них должно быть значение 93. Если вам неизвестно количество углеводов и белков, то нельзя сказать, являются ли эти величины одинаковыми.

## ALL, SOME, ANY

Тысячелетия назад греческий философ Аристотель сформулировал систему логики, ставшей основой значительной части западной мысли. Сущность этой логики состоит в следующем. Следует начать с набора посылок, о которых известно, что они истинные, затем применить к ним операции и вывести, таким образом, новые истины. Вот пример такой процедуры.

```
Посылка 1. Все греки – люди.
Посылка 2. Все люди смертны.
Заключение. Все греки смертны.
```

А вот еще пример.

```
Посылка 1. Некоторые греки – женщины.
Посылка 2. Все женщины – люди.
Заключение. Некоторые греки – люди.
```

Другой способ выражения идеи второго примера состоит в следующем.

```
Если какие-либо греки – женщины и все женщины – люди, то некоторые греки люди.
```

В первом примере в обеих посылках используется квантор всеобщности ALL (все), который дает возможность сделать в заключении разумный вывод обо всех греках. Во втором примере в одной из посылок используется квантор существования SOME (некоторые), который также позволяет сделать в заключении вывод обо всех греках. А в третьем примере, чтобы сделать то же заключение, что и во втором примере, используется квантор существования ANY (какие-либо) – синоним квантора SOME.

Посмотрите, как кванторы SOME, ANY и ALL применяются в SQL.

Рассмотрим пример с бейсбольной статистикой. Бейсбол – это вид спорта, требующий значительных физических нагрузок, особенно у питчера, т.е. игрока, подающего мяч. Ему за время игры приходится 90-150 раз бросать мяч со своей возвышенности до основной базы – места, где находится игрок с битой. Такие нагрузки очень утомляют, и часто получается так, что к концу игры питчера на подаче приходится заменять. Бессменно подавать мячи в течение всей игры – это уже выдающееся достижение, причем неважно, привели такие попытки к победе или нет.

Предположим, что вы собираете данные о количестве тех игр, в которых питчеры высшей лиги бессменно подавали мяч. В одной из ваших таблиц перечислены все питчеры Американской лиги, а в другой – Национальной лиги. В каждой из этих таблиц обо всех игроках хранятся следующие данные: имя игрока, его фамилия и количество игр, бессменно проведенных им на подаче.

### Возможная двусмысленность квантора any

Первоначально в SQL в качестве квантора существования использовалось слово ANY. Это использование оказалось достаточно запутанным и приводило к ошибкам, так как в английском языке слово any иногда означает всеобщность, а иногда – существование."Do any of you know where Baker Street is?" (Кто-нибудь из вас знает, где находится улица Бейкер-Стрит?) "I can eat more eggs than any of you." (Я могу съесть больше яиц, чем любой из вас.) В первом предложении, скорее всего, задается вопрос, есть ли хотя бы один человек, который знает, где находится улица Бейкер-Стрит. Any используется как квантор существования. Второе предложение – это хвастливое заявление о том, что я могу съесть больше яиц, чем самый большой едок из окружающих. В этом случае any используется как квантор всеобщности. Поэтому разработчики стандарта SQL-92 хотя и оставили в нем от предыдущих версий SQL слово ANY, чтобы была совместимость с предшествующими

продуктами, но в то же время добавили его менее запутанный синоним – слово SOME. SQL:2OO3 также поддерживает оба квантора существования.

В Американской лиге разрешается, чтобы назначенный хиттер (designated hitter, DH) мог бить битой по мячу вместо одного из девяти игроков, играющих в обороне. (Кто такой хиттер? Это игрок с битой. А назначенный хиттер – это тот хиттер, которому не требуется играть в оборонительной позиции.) Обычно DH делают это вместо питчеров, потому что те, как известно, плохие хиттеры. Питчерам приходится тратить слишком много времени и усилий на совершенствование своего броска. Поэтому у них остается мало времени, чтобы тренироваться с битой, как это делают остальные игроки.

Скажем, вам теоретически известно, что в среднем у стартовых питчеров (питчеров, играющих с самого начала игры) Американской лиги насчитывается больше игр, в которых они бессменно подавали мяч, чем у стартовых питчеров Национальной лиги. Такой вывод основан на тех ваших наблюдениях, что назначенные хиттеры дают возможность хорошо бросающим, но слабым в обращении с битой питчерам Американской лиги оставаться на подаче до завершения игры. Так как DH уже работают с битой вместо них, то становится не важным, что питчеры – слабые хиттеры. Однако в Национальной лиге питчер, как правило, заменяется в конце игры хиттером, у которого обычно больше шансов подготовить лучший удар. Чтобы проверить свою теорию, вы составляете следующий запрос:

```
SELECT FirstName, LastName
FROM AMERICAN_LEAGUER
WHERE CompleteGames > ALL
(SELECT CompleteGames
FROM NATIONAL_LEAGUER);
```

Подзапрос (внутренний оператор SELECT) возвращает список, показывающий для каждого питчера Американской лиги количество тех игр, в которых он бессменно подавал мяч. Внешний же запрос возвращает имена и фамилии всех питчеров Американской лиги, которые, бессменно подавая мяч, сыграли больше игр, чем каждый питчер Национальной лиги. В этом запросе используется квантор всеобщности ALL. Таким образом, возвращаются имена и фамилии тех питчеров Американской лиги, которые имеют больше игр с бессменной подачей мяча, чем самый лучший по этому показателю питчер Национальной лиги.

Посмотрите на следующий похожий оператор:

```
SELECT FirstName, LastName
FROM AMERICAN_LEAGUER
WHERE CompleteGames > ANY
(SELECT CompleteGames
FROM NATIONAL_LEAGUER);
```

В этом случае вместо квантора всеобщности ALL используется квантор существования ANY. Подзапрос (внутренний, вложенный запрос) здесь такой же, как и в предыдущем примере. В результате выполнения этого подзапроса получается полный список игр, в течение которых питчеры Национальной лиги бессменно подавали мяч. А внешний подзапрос возвращает имена и фамилии всех тех питчеров Американской лиги, которые, бессменно подавая мяч, сыграли больше игр, чем какой-либо из питчеров Национальной лиги. Вот в этом-то запросе и используется квантор существования ANY. Вы можете быть уверены, что хотя бы один из питчеров Национальной лиги ни одной игры не провел бессменно на подаче. Поэтому результат, скорее всего, будет включать в себя всех питчеров Американской лиги, которые бессменно провели хотя бы одну игру.

Если заменить ключевое слово ANY эквивалентным ему SOME (какой-то), то результат будет точно такой же. И если истинно утверждение, что "хотя бы один питчер Национальной лиги ни

одной игры бессменно не провел на подаче", то тогда можно сказать, что "какой-то питчер Национальной лиги ни одной игры не провел на подаче бессменно".

### EXISTS

Для определения того, возвращает ли подзапрос какие-либо строки, вместе с ним можно использовать предикат EXISTS (существует). Если подзапрос возвращает хотя бы одну строку, то этот результат удовлетворяет условию EXISTS и выполняется внешний запрос. Ниже приведен пример использования предиката EXISTS.

```
SELECT FirstName, LastName
FROM CUSTOMER
WHERE EXISTS
(SELECT DISTINCT CustomerlD
FROM SALES
WHERE SALES.CustomerID = CUSTOMER.CustomerlD)
```

В таблице SALES (продажи) хранятся данные обо всех продажах, выполненных компанией. В этой таблице в поле CustomerID находятся идентификаторы покупателей, которые участвовали в какой-нибудь из сделок. В таблице CUSTOMER (покупатель) хранятся имя и фамилия каждого покупателя, но нет никакой информации о конкретных сделках.

Имеющийся в последнем примере подзапрос возвращает строку для каждого покупателя который сделал хотя бы одну покупку. А внешний запрос возвращает имена и фамилии тех кто участвовал в сделках, указанных в таблице SALES.

Предикат EXISTS, как показано в следующем запросе, эквивалентен сравнению COUNT с нулем:

```
SELECT FirstName, LastName
FROM CUSTOMER
WHERE 0 <>
(SELECT COUNT(*)
FROM SALES
WHERE SALES.CustomerID = CUSTOMER.CustomerlD);
```

Для каждой строки таблицы SALES, в которой значение CustomerlD равно какому-либо значению CustomerlD из таблицы CUSTOMER, этот оператор выводит столбцы FirstName (имя) и LastName (фамилия) из таблицы CUSTOMER. Поэтому для каждой сделки, отмеченной в таблице SALES, этот оператор выводит имя и фамилию того покупателя, который в ней участвовал.

### UNIQUE

Вместе с подзапросом, как и предикат EXISTS, можно также использовать предикат UNIQUE (уникальный). Если первый из этих предикатов является истинным тогда, когда подзапрос возвращает хотя бы одну строку, то второй из них будет истинным тогда, когда среди возвращенных подзапросом строк нет двух одинаковых. Другими словами, предикат UNIQUE будет истинным, если все возвращаемые подзапросом строки будут уникальными. Проанализируйте следующий пример:

```
SELECT FirstName, LastName
FROM CUSTOMER
WHERE UNIQUE
(SELECT CustomerID FROM SALES
WHERE SALES.CustomerID = CUSTOMER.CustomerID);
```

Этот оператор возвращает только имена и фамилии всех новых покупателей, которые участвовали лишь в одной из сделок, указанных в таблице SALES. Два значения NULL считаются не равными друг другу и, следовательно, уникальными. И когда ключевое слово UNIQUE применяется к таблице, полученной в результате выполнения подзапроса, а в этой таблице никаких строк, кроме двух неопределенных, больше нет, то и тогда предикат UNIQUE является истинным.

### DISTINCT

Предикат DISTINCT (отличающийся) похож на UNIQUE, за исключением отношения к значениям NULL. Если в таблице, полученной в результате выполнения подзапроса, все значения являются уникальными, тогда они отличаются друг от друга. Однако, в отличие от результата предиката UNIQUE, если к такой таблице применить ключевое слово DISTINCT, а в ней, кроме двух неопределенных, больше никаких строк нет, то предикат DISTINCT является ложным. Два значения NULL не считаются **отличающимися** друг от друга, хотя и считаются уникальными. Такая странная ситуация выглядит противоречиво, но этому есть свое объяснение. Дело в том, что в некоторых ситуациях два значения NULL должны считаться отличными друг от друга, а в некоторых – одинаковыми. Тогда в первом случае надо использовать UNIQUE, а во втором – DISTINCT.

### OVERLAPS

Предикат OVERLAPS (перекрывает) применяется для того, чтобы определить, не перекрывают ли друг друга два промежутка времени. Он полезен тогда, когда нужно избежать "накладок" в расписании. Когда два промежутка времени перекрываются, то этот предикат возвращает значение True. Если они не перекрываются, то будет возвращено значение False.

Промежуток времени можно указать двумя способами: в виде начального и конечного времени или в виде начального времени и длительности. Вот несколько примеров:

```
(TIME '2:55:00', INTERVAL '1' HOUR)
OVERLAPS
(TIME '3:30:00', INTERVAL '2' HOUR)
```

В только что приведенном примере будет возвращено значение True, так как 3:30 наступает после 2:55 меньше чем через час.

```
(TIME '9:00:00', TIME '9:30:00')
OVERLAPS
(TIME '9:29:00', TIME '9:31:00')
```

Во втором примере будет возвращено значение True, потому что два промежутка времени перекрываются в течение одной минуты.

```
(TIME '9:00:00', TIME '10:00:00')
OVERLAPS
(TIME '10:15:00', INTERVAL '3' HOUR)
```

В третьем примере будет возвращено значение False, так как два промежутка времени не перекрываются.

```
(TIME '9:00:00', TIME '9:30:00')
OVERLAPS
(TIME '9:30:00', TIME '9:35:00')
```

И наконец, в последнем примере будет возвращено значение False – хотя два промежутка времени и являются смежными, но они не перекрываются.

### MATCH

В главе 5 шла речь о ссылочной целостности, которая включает в себя поддержание согласованности в многотабличной базе данных. Целостность может быть нарушена, если в дочернюю таблицу добавить строку, у которой нет соответствующей строки в родительской таблице. Можно вызвать похожие сложности, удалив из родительской таблицы строку и оставив в дочерней те строки, которые соответствуют удаленной.

Скажем, вы, ведя свой бизнес, собираете данные о своих покупателях в таблицу CUSTOMER (покупатель), а данные о продажах заносите в таблицу SALES (продажи). Вам не хочется добавлять строку в SALES до тех пор, пока данные о покупателе, участвующем в соответствующей сделке, не появятся в таблице CUSTOMER. Вам также не хочется удалять из CUSTOMER данные о покупателе, если он участвовал в сделках, информация о которых все еще хранится в таблице SALES. Перед тем как выполнять вставку или

удаление, вам, возможно, захочется проверить, не приведет ли к нарушениям целостности выполнение со строкой какой-либо из этих операций. Такую проверку может выполнить предикат MATCH (соответствие).

Как используется предикат MATCH, можно узнать с помощью примера, где применяются опять же таблицы CUSTOMER и SALES. CustomerID (идентификатор покупателя) – это первичный ключ таблицы CUSTOMER, и работает он как внешний ключ таблицы SALES. В каждой строке таблицы CUSTOMER должно быть уникальное значение CustomerID, не равное NULL. А в таблице SALES ключ CustomerID не является уникальным, потому что в ней повторяются его значения, относящиеся к тем, кто покупал больше одного раза. Это нормальная ситуация, которая не угрожает целостности, потому что в этой таблице CustomerID является не первичным, а внешним ключом.

***Совет:***
*По-видимому, в столбце CustomerID таблицы SALES могут быть и значения NULL, потому что кто-то может зайти к вам с улицы, купить что-то и выйти еще до того, как вы сможете ввести его или ее имя, фамилию и адрес в таблицу CUSTOMER. Тогда в дочерней таблице может появиться строка, у которой нет соответствующей строки в родительской таблице. Чтобы справиться с этой трудностью, можно включить в таблицу CUSTOMER строку для "общего" пользователя и заносить все эти анонимные продажи в базу на его идентификатор*.

Скажем, к кассиру подходит покупательница и утверждает, что 18 мая 2003 года она купила истребитель-невидимку F-117A "Стелс". Теперь же она хочет вернуть самолет, потому что его, словно авианосец, видно на вражеских радарах. Ее заявление может быть подтверждено с помощью проверки вашей базы SALES с помощью MATCH. Прежде всего необходимо найти в столбце CustomerID идентификатор покупательницы и присвоить его значение переменной.vcustid, а затем можно использовать следующий синтаксис, в котором применяются столбцы CustomerID, ProductID (идентификатор товара), SaleDate (дата продажи):

```
… WHERE (-.vcustid, 'F-117-A1, '2003-05-18')
MATCH
(SELECT CustomerID, ProductID, SaleDate
FROM SALES).
```

Если есть запись о продаже с этим идентификатором пользователя, товаром и датой, то предикат MATCH возвращает значение True. А вы возвращаете покупательнице деньги. (***Примечание****: если какое-либо значение в первом аргументе предиката MATCH будет неопределенным, то всегда будет возвращаться значение True*.)

***Технические подробности:***
*Разработчики языка SQL добавили в него предикаты MATCH и UNIQUE по одной и той же причине – эти предикаты дают возможность явно выполнять проверки, которые определены для неявных ограничений, связанных со ссылочной целостностью и уникальностью*.

Предикат MATCH имеет такой общий вид:

```
Значение_типа_записи ROW MATCH [UNIQUE] [SIMPLE| PARTIAL | FULL ] Подзапрос
```

Ключи UNIQUE (уникальный), SIMPLE (простой), PARTIAL (частичный) и FULL (полный) связаны с правилами обработки выражения типа записи, имеющего столбцы с неопределенными значениями. Правила для предиката MATCH являются точной копией соответствующих правил ссылочной целостности.

## Правила ссылочной целостности

Правила ссылочной целостности требуют, чтобы значения в столбце (или столбцах) одной таблицы соответствовали значениям в столбце (или столбцах) другой. Столбцы в

первой таблице называются **внешним ключом**, а во второй – **первичным**, или**уникальным**, **ключом**. Например, столбец EmpDeptNo (номер отдела, где работает сотрудник) из таблицы EMPLOYEE (сотрудник) можно объявить внешним ключом, который ссылается на столбец DeptNo (номер отдела) из таблицы DEPT (отдел). Это соответствие дает гарантию, что когда в таблицу EMPLOYEE о сотруднике заносится информация, что он работает в отделе 123, то в таблице DEPT появляется запись, в которой значением столбца DeptNo является 123.

Такая ситуация является довольно простой, если внешний и первичный ключи состоят из одного столбца каждый. Однако оба эти ключа могут состоять также из множества столбцов. Например, значение в столбце DeptNo может быть уникальным только для одного и того же значения в столбце Location (представительство). Поэтому, чтобы однозначно определить строку из таблицы DEPT, необходимо указать значение и для столбца Location, и для столбца DeptNo. Если, например, отдел 123 имеется в двух представительствах, расположенных соответственно в Бостоне и в Тампе, то отделы необходимо указывать как ('Boston', '123') и ('Tampa', '123'). В таком случае для указания в таблице EMPLOYEE строки из таблицы DEPT необходимо использовать два столбца. Их можно назвать EmpLoc (представительство, где работает сотрудник) и EmpDeptNo. Если сотрудник работает в каком-либо отделе, расположенном в Бостоне, то значениями столбцов EmpLoc и EmpDeptNo будут соответственно Boston' и '123'. Таким образом, объявление внешнего ключа в EMPLOYEE будет следующим:

```
FOREIGN KEY (EmpLoc, EmpDeptNo)
REFERENCES DEPT (Location, DeptNo)
```

Вывод правильных заключений из ваших данных в громадной степени усложняется, если в этих данных содержатся неопределенные значения. Иногда данные с такими значениями надо интерпретировать одним способом, а иногда – другим. Разные интерпретации данных, в которых встречаются значения NULL, можно задавать с помощью ключевых слов UNIQUE, SIMPLE, PARTIAL и FULL. Если в ваших данных нет неопределенных значений, то вы в значительной степени избавитесь от необходимости ломать голову, просто возьмете и перейдете к следующему разделу "Логические связки". Ну а если в ваших данных такие значения есть, то тогда от режима скорочтения сейчас лучше отказаться и начать медленно и внимательно читать последующие абзацы. В каждом из них описана отдельная ситуация, связанная со значениями NULL, и рассказывается, как с ней справляется предикат MATCH.

Если значения EmpLoc и EmpDeptNo вместе являются или не являются неопределенными, то правила ссылочной целостности будут такие же, как и для ключей, состоящих из одного столбца с неопределенными или определенными значениями. Но если значение EmpLoc неопределенное, а EmpDeptNo – нет или наоборот, то тогда нужны новые правила. И какие же правила нужны, когда в таблицу EMPLOYEE при вставке или обновлении ее строк вводятся значения EmpLoc и EmpDeptNo как (NULL, '123') или ('Boston', NULL)? Существует шесть вариантов, при которых используются SIMPLE, PARTIAL и FULL вместе с ключевым словом UNIQUE или без него. Присутствие этого ключевого слова означает следующее. Чтобы предикат был истинным, значение типа записи, найденное с помощью MATCH в таблице-результате выполнения подзапроса, должно быть уникальным. И если в значении выражения R, имеющем тип записи, оба компонента являются неопределенными, то предикат MATCH возвращает значение True, каким бы ни было содержимое сравниваемой таблицы, полученной при выполнении подзапроса.

Если в значении выражения R типа записи с ключевым словом SIMPLE, но без UNIQUE, ни один из компонентов не является неопределенным и при этом хотя бы одна строка в таблице, полученной при выполнении подзапроса, соответствует R, то предикат MATCH возвращает значение True. В противном случае он возвращает значение False.

Если в значении выражения R типа записи с ключевыми словами SIMPLE и UNIQUE ни один из компонентов не является неопределенным и при этом хотя бы одна строка в

таблице, полученной при выполнении подзапроса, уникальна и соответствует R, то тогда предикат MATCH возвращает значение True. В противном случае он возвращает значение False.

Если в значении выражения R типа записи с ключевым словом SIMPLE какой-нибудь из компонентов является неопределенным, то предикат MATCH возвращает значение True.

Если в значении выражения R типа записи с ключевым словом PARTIAL, но без UNIQUE, какой-нибудь из компонентов не является неопределенным и при этом определенные значения хотя бы одной строки в таблице, полученной при выполнении подзапроса, соответствуют R, то тогда предикат MATCH возвращает значение True. В противном случае он возвращает значение False.

Если в значении выражения R типа записи с ключевыми словами PARTIAL и UNIQUE какой-нибудь из компонентов не является неопределенным и при этом определенные части R соответствуют определенным частям хотя бы одной уникальной строки в таблице, полученной при выполнении подзапроса, то тогда предикат MATCH возвращает значение True. В противном случае он возвращает значение False.

Если ни один из компонентов значения выражения R типа записи с ключом FULL, но без UNIQUE, не является неопределенным и при этом хотя бы одна строка в таблице, полученной при выполнении подзапроса, соответствует R, то тогда предикат MATCH возвращает значение True. В противном случае он возвращает значение False.

Если ни один из компонентов значения выражения R типа записи с ключами FULL и UNIQUE не является неопределенным и при этом хотя бы одна строка в таблице, полученной при выполнении подзапроса, уникальна и соответствует R, то тогда предикат MATCH возвращает значение True. В противном случае он возвращает значение False.

Если какой-либо из компонентов выражения R является неопределенным и указано ключевое слово FULL, то предикат MATCH возвращает значение False.

### Правила комитета по стандартам

С появлением SQL-89 стало подразумеваться, что по умолчанию используется правило UNIQUE. Это случилось еще до того, как кто-либо успел предложить или обсудить другие варианты. Но такие предложения появились уже во время разработки SQL-92. Кто-то упорно предпочитал правила PARTIAL и считал, что они должны быть единственными. С этой точки зрения правила SQL-89 (UNIQUE) были настолько нежелательны, что рассматривались как ошибка, которую необходимо исправить с помощью правил PARTIAL. Были и те, кому больше нравились правила UNIQUE, а правила PARTIAL для них были непонятными, двусмысленными и неэффективными. Впрочем, были и те, кто предпочитал еще более строгие правила FULL. В конце концов этот спор был решен следующим образом: пользователи получили в свое распоряжение все три ключевых слова и теперь могли выбирать тот подход, который им нужен. А потом, с появлением SQL: 1999, добавились еще и правила SIMPLE. Впрочем, рост числа правил делает работу с неопределенными значениями какой угодно, но только не простой (simple). Итак, если не указаны ключевые слова SIMPLE, PARTIAL или FULL, то будут выполняться правила SIMPLE.

### Логические связки

Как видно из массы предыдущих примеров, чтобы из таблицы получить нужные строки, одного условия в запросе часто бывает недостаточно. В некоторых случаях условий, которым должны удовлетворять строки, должно быть не меньше двух. В других же случаях, чтобы быть выбранной, строка должна удовлетворять одному из нескольких условий. А иногда нужно получить только те строки, которые указанному условию не удовлетворяют. Для выполнения этих требований в SQL имеются логические связки AND, OR и NOT.

### AND

Если для получения строки необходимо, чтобы все условия из какого-либо их набора имели значение True, используйте логическую связку AND (и). Проанализируйте следующий пример, в нем используются поля InvoiceNo (номер счета-фактуры), SaleDate (дата продажи), Salesperson (продавец), TotalSale (всего продано) из таблицы SALES (продажи):

```
SELECT InvoiceNo, SaleDate, Salesperson, TotalSale
FROM SALES
WHERE SaleDate >= '2003-05-18'
AND SaleDate <= '2003-05-24';
```

Предложение WHERE (где) должно соответствовать следующим двум условиям.

- Дата SaleDate должна была наступить не раньше 18 мая 2003 года.
- Дата SaleDate должна была наступить не позже 24 мая 2003 года.

Таким образом, обоим условиям будут одновременно соответствовать только те строки, в которых записаны данные о продажах в течение недели, прошедшей с 18 мая. Запрос возвратит именно эти строки.

***Внимание:***

*Обратите внимание, что связка AND (и) имеет чисто логическое значение. Такое ограничение иногда может привести к путанице, потому что союз "и" люди обычно используют в более широком смысле. Предположим, например, что ваш босс говорит: "Мне нужны данные о продажах, проведенных Фергюсоном и Фордом". А раз он сказал о "Фергюсоне и Форде", то вы, возможно, напишете следующий запрос SQL:*

```
SELECT *
FROM SALES
WHERE Salesperson = 'Ferguson'
AND Salesperson = 'Ford';
```

Только не несите его результаты своему боссу. Ладно? Тому, что он имел в виду, больше соответствует другой запрос:

```
SELECT *
FROM SALES
WHERE Salesperson IN ('Ferguson', 'Ford');
```

Первый запрос будет безрезультатным, потому что ни одну из продаж, отмеченных в таблице SALES, Фергюсон и Форд не провели вместе. Второй же запрос вернет информацию обо всех продажах, сделанных или Фергюсоном, или Фордом. Скорее всего, она-то и требовалась вашему боссу.

### OR

Если для возвращения строки необходимо, чтобы из нескольких условий для этой строки было верно хотя бы одно, используйте логическую связку OR (или):

```
SELECT invoiceNo, SaleDate, Salesperson, TotalSale
FROM SALES
WHERE Salesperson = 'Ford'
OR TotalSale > '200';
```

В результате выполнения этого запроса будут получены данные обо всех продажах, которые были сделаны на любую сумму, но только Фордом, или были сделаны кем угодно, но при этом превышали 200 долларов.

### NOT

Для отрицания условия служит связка NOT (не). Если к условию, которое возвращает значение True, добавить NOT, то после этого условие будет возвращать значение False. А

если до изменения условие возвращало False, то после добавления к нему NOT оно будет возвращать True. Посмотрите на следующий пример:

```
SELECT InvoiceNo, SaleDate, Salesperson, TotalSale
FROM SALES
WHERE NOT (Salesperson = 'Ford');
```

Этот запрос возвращает строки для всех сделок по продажам, совершенных всеми продавцами, кроме Форда.

***Внимание:***
*Иногда при использовании связки (AND, OR или NOT) бывает неясно, какая у нее область действия. Чтобы гарантировать применение связки именно к нужному предикату, заключите его в круглые скобки. В последнем примере связка NOT применяется как раз к целому предикату (Salesperson = Ford'), а не к какой-либо его части*.

## Предложения GROUP BY

Иногда вместо того, чтобы получить отдельные записи, вам может понадобиться узнать что-либо о группе записей. В этом случае вам поможет предложение GROUP BY (группировать по).

Предположим, что вы менеджер по продажам и хотите посмотреть эффективность ваших продаж. Для этого вы можете воспользоваться оператором SELECT, как показано в следующем примере:

```
SELECT InvoiceNo, SaleDate, Salesperson, TotalSale
FROM SALES;
```

Полученный результат приведен на рис. 9.1.

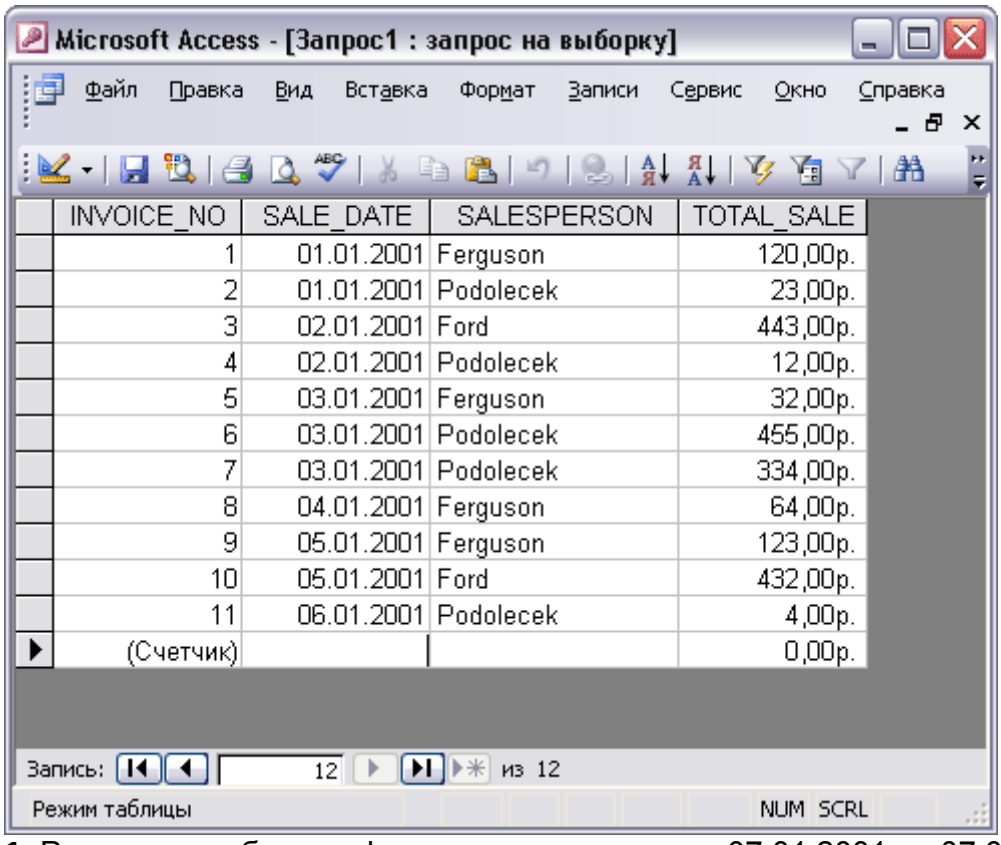


**Рис. 9.1**. Результат выбора информации о продажах с 07.01.2001 по 07.07.2001

Этот результат дает вам лишь некоторое представление о том, как работают ваши продавцы, поскольку здесь выводится информация о небольшом количестве продаж. Однако в реальной жизни продажи компаний очень велики, и в этом случае уже непросто

будет определить, как были достигнуты результаты. Чтобы проверить, была ли коммерчески эффективной работа продавцов, вы можете скомбинировать предложение GROUP BY с одной из функций агрегирования (также называемыми итоговыми функциями), чтобы получить количественную картину о выполнении продаж. К примеру, вы можете просмотреть, кто из продавцов работает с более дорогостоящими и прибыльными позициями, используя функцию среднего значения (AVG):

```
SELECT Salesperson, AVG(TotalSale)
FROM SALES
GROUP BY Salesperson;
```

Полученный результат приведен на рис. 9.2.

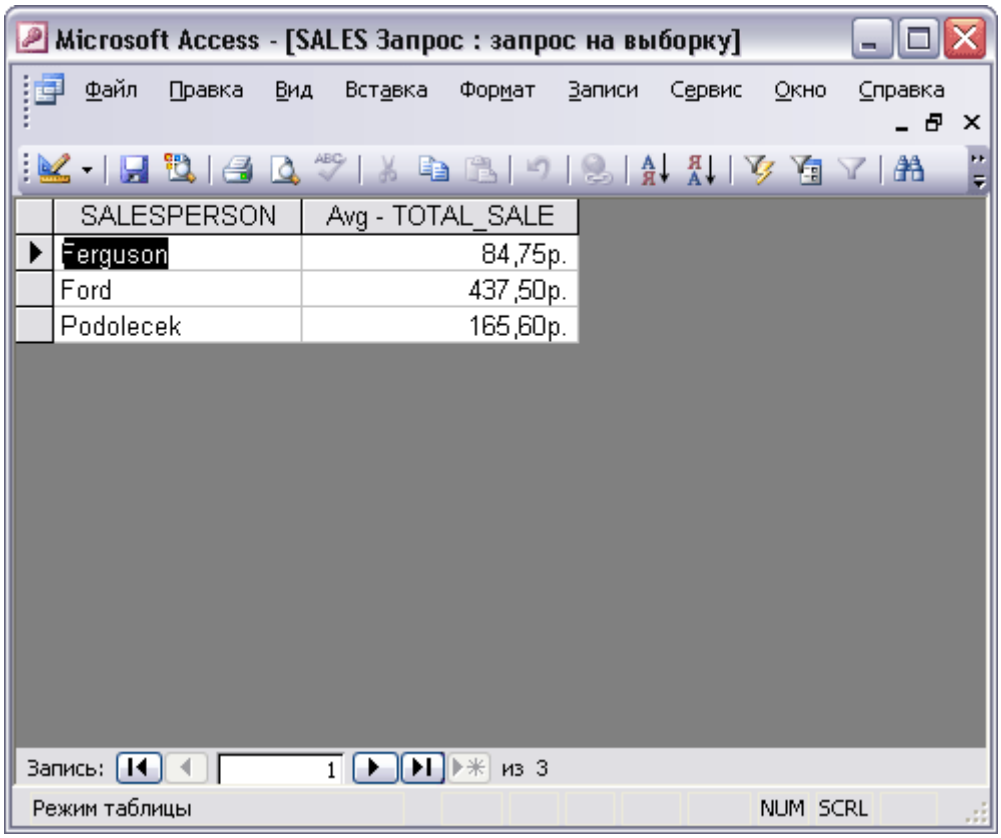


**Рис. 9.2**. Средний уровень продаж по каждому продавцу

Как видно из рис. 9.2, средние продажи Фергюсона значительно выше, чем у двух других продавцов. Чтобы сравнить общие продажи по каждому продавцу, выполните следующий запрос:

```
SELECT Salesperson, SUM(TotalSale)
FROM SALES
GROUP BY Salesperson;
```

Результат этого запроса приведен на рис. 9.3.

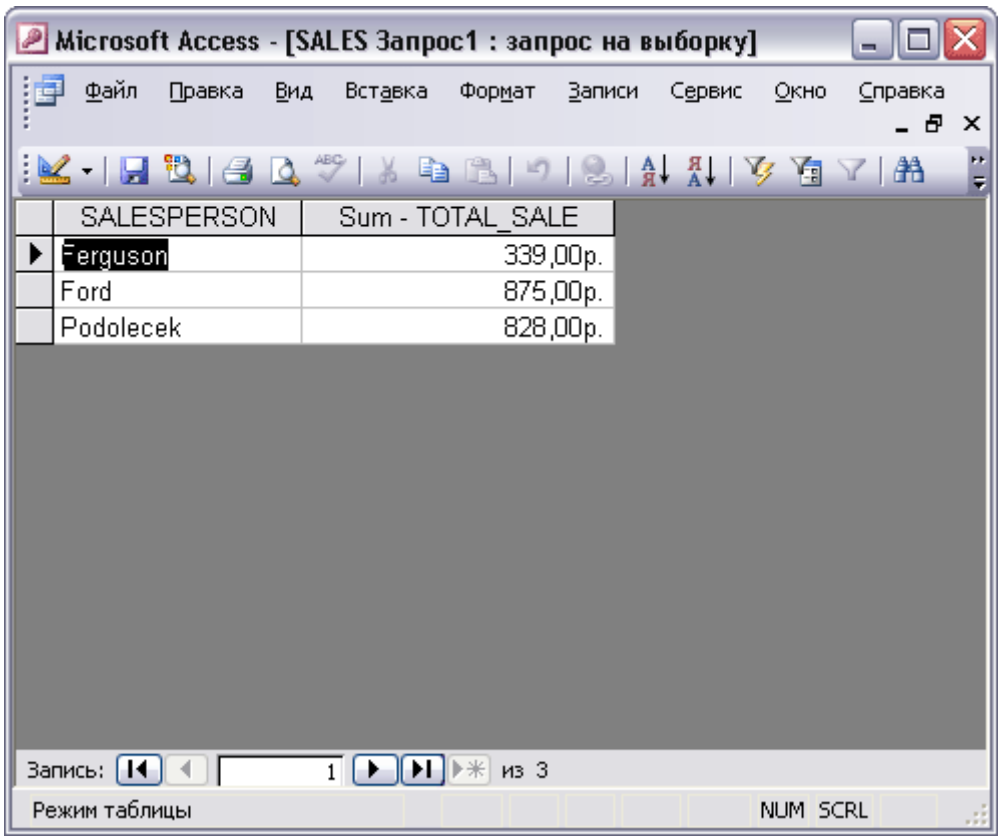


**Рис. 9.3**. Общие продажи по каждому продавцу

Как и в случае среднего уровня продаж, Фергюсон имеет также самый высокий уровень общих продаж.

### Предложение HAVING

Предложение HAVING позволяет еще лучше анализировать сгруппированные данные. Предложение HAVING (при условии) – это фильтр, который по своему действию похож на предложение WHERE, но, в отличие от предложения WHERE, HAVING работает не с отдельными строками, а с их группами. Проиллюстрируем действие предложения HAVING, используя следующий пример. Предположим, что менеджеру по продажам нужно сосредоточиться на работе других продавцов. Для этого ему необходимо исключить из общих данных количество продаж Фергюсона, поскольку его продажи находятся "в другой весовой категории". Чтобы сделать это, воспользуемся предложением HAVING:

```
SELECT Salesperson, SUM(TotalSale)
FROM SALES
GROUP BY Salesperson
HAVING Salesperson <> 'Ferguson';
```

Результат этого запроса приведен на рис. 9.4. Строки, которые относятся к продавцу по фамилии Фергюсон, на экран не выводятся.

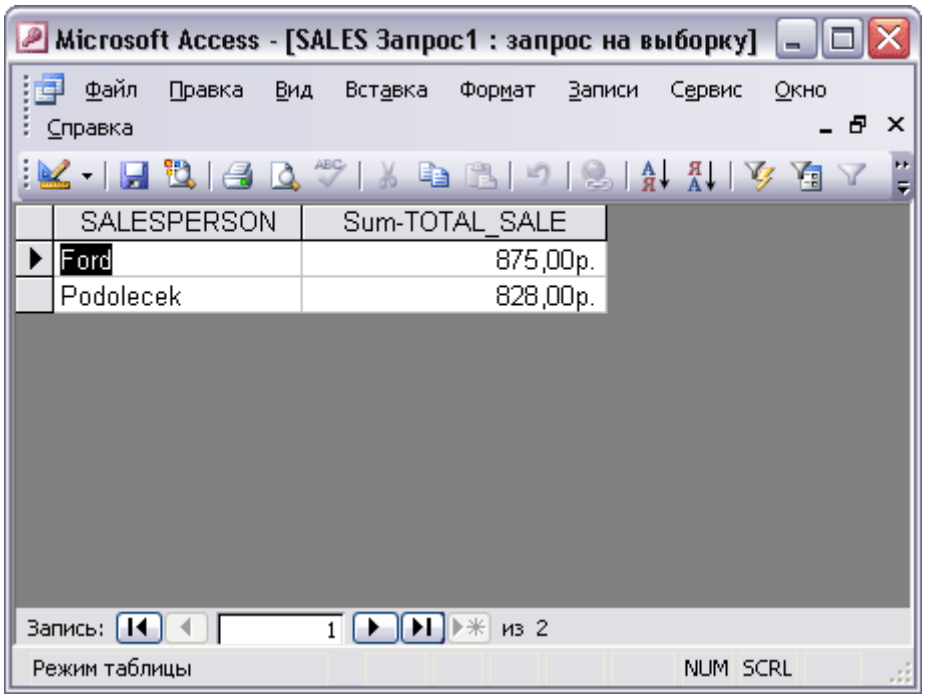


**Рис. 9.4**. Общие продажи по каждому продавцу за исключением Фергюсона

Чтобы показать таблицу, выводимую запросом, в алфавитном порядке или в обратном алфавитном порядке, используйте предложение ORDER BY (по порядку). В то время как предложение GROUP BY собирает строки в группы и сортирует группы по алфавиту, ORDER BY сортирует отдельные строки. ORDER BY должно быть последним предложением в запросе. Если в запросе также имеется и предложение GROUP BY, то оно вначале собирает строки вывода в группы. Затем предложение ORDER BY сортирует строки, находящиеся внутри каждой группы. А если предложения GROUP BY нет, то оператор рассматривает всю таблицу как группу и предложение ORDER BY сортирует все ее строки таким образом, чтобы были упорядочены значения в столбцах, указанных в этом предложении.

Это можно проиллюстрировать с помощью данных из таблицы SALES (продажи). В ней имеются столбцы InvoiceNo (номер счета-фактуры), SaleDate (дата продажи), Salesperson (продавец), TotalSale (всего продано). Все данные SALES можно увидеть, но в произвольном порядке, если воспользоваться следующим простым примером:

```
SELECT * FROM SALES;
```

В одних реализациях это порядок, в котором строки вставлялись в таблицу, а в других реализациях строки могут быть выстроены в зависимости от времени самого последнего обновления каждой из них. Кроме того, порядок может неожиданно измениться, если кто-то

физически реорганизует базу данных. В большинстве случаев порядок расположения получаемых строк приходится указывать. Возможно, вам нужно увидеть строки, которые расположены по порядку, задаваемому значениями из столбца SaleDate:

```
SELECT * FROM SALES ORDER BY SaleDate;
```

При выполнении этого примера все строки таблицы SALES будут возвращены именно втом порядке, который задан значениями SaleDate.

А порядок расположения тех строк, у которых в столбце SaleDate одинаковые значения, зависит от используемой реализации. Впрочем, и для строк с одинаковыми значениями Sale-Date можно также указать порядок сортировки. Возможно, вам, например, нужно для каждого значения SaleDate видеть строки таблицы SALES, расположенные по порядку, которые заданы значениями InvoiceNo:

```
SELECT * FROM SALES ORDER BY SaleDate, InvoiceNo;
```

В этом примере таблица SALES вначале упорядочивается по значениям SaleDate; затем для каждого такого значения строки этой таблицы располагаются по порядку, задаваемому InvoiceNo. Однако не путайте этот пример со следующим запросом:

```
SELECT * FROM SALES ORDER BY InvoiceNo, SaleDate;
```

При выполнении этого запроса SALES упорядочивается по столбцу InvoiceNo. Затем для каждого значения InvoiceNo строки таблицы SALES располагаются по порядку, задаваемому столбцом SaleDate. Скорее всего, нужный вам результат вы не получите, потому что мало вероятно, чтобы для одного номера счета-фактуры было множество дат продажи.

Следующий запрос является очередным примером того, как SQL может возвращать данные:

```
SELECT * FROM SALES ORDER BY Salesperson, SaleDate;
```

В этом примере упорядочивание сначала идет по столбцу Salesperson, а затем – по SaleDate. Просмотрев данные, расположенные в таком порядке, вы, возможно, захотите его изменить:

```
SELECT * FROM SALES ORDER BY SaleDate, Salesperson;
```

Теперь упорядочивание сначала идет по столбцу SaleDate, а затем – по Salesperson.

Во всех этих примерах упорядочивание идет в порядке возрастания (ASC), который является порядком сортировки по умолчанию. Последний оператор SELECT вначале показывает самые ранние продажи (строки таблицы SALES), а в пределах определенной даты ставит продажи, проведенные Адамсом, перед продажами, проведенными Бейкером. А если вы предпочитаете порядок убывания (DESC), можете указать его для одного или множества столбцов из предложения ORDER BY:

```
SELECT * FROM SALES
ORDER BY SaleDate DESC, Salesperson ASC;
```

В этом примере для данных продаж указывается порядок убывания дат, в результате чего записи о самых недавних продажах будут показаны первыми, но для продавцов указывается порядок возрастания, т.е. их фамилии будут располагаться в обычном алфавитном порядке.

# Реляционные операторы

В этой главе…

- Объединение таблиц, имеющих похожую структуру
- Объединение таблиц, имеющих разную структуру
- Получение нужных данных из множества таблиц

---

SQL – это язык запросов, используемый в реляционных базах данных. Почти во всех примерах предыдущих глав рассматривались простые базы данных с одной таблицей. Теперь настало время показать, в чем же состоит **реляционность** реляционной базы. Вообще говоря, эти базы называются "реляционными" потому, что состоят из множества связанных друг с другом таблиц (а "связанные друг с другом" – это по-английски "related").

Так как данные, хранящиеся в реляционной базе, распределены по множеству таблиц, то запрос обычно извлекает данные из более чем одной таблицы. В SQL:2003 имеются операторы, которые объединяют данные из множества исходных таблиц в одну. Это операторы UNION, INTERSECTION и EXCEPT, а также семейство операторов объединения JOIN. Причем каждый из них объединяет данные своим особым способом.

## UNION

Оператор **UNION** (объединение) – это реализация в языке SQL оператора объединения из реляционной алгебры. Оператор UNION дает возможность получать информацию из нескольких таблиц, имеющих одинаковую структуру. **Одинаковая структура** означает следующее.

- Во всех таблицах имеется одинаковое количество столбцов.
- У всех соответствующих столбцов должны быть идентичный тип данных и одинаковая длина.

При соблюдении этих критериев таблицы являются совместимыми для объединения. В результате объединения двух таблиц возвращаются все строки, имеющиеся в каждой из них. Не возвращаются только повторяющиеся строки.

Скажем, вы создаете базу данных по бейсбольной статистике (см. главу 9). Она состоит из двух таблиц, совместимых для объединения, которые называются AMERICAN (Американская лига) и NATIONAL (Национальная лига). В каждой из них имеются по три столбца, и типы у всех соответствующих столбцов совпадают. На самом деле даже имена у таких столбцов одинаковые, хотя для объединения это условие не является обязательным.

В таблице NATIONAL перечислены имена, фамилии питчеров Национальной лиги и количество тех игр, в которых они все время были на подаче. Эти данные находятся в столбцах FirstName (имя), LastName (фамилия) и CompleteGames (полностью сыгранные игры). Та же информация, но только о питчерах Американской лиги, содержится в таблице AMERICAN. Если объединить таблицы

NATIONAL и AMERICAN с помощью оператора UNION, то в результате получится виртуальная таблица со всеми строками из первой и второй таблиц. В этом примере, чтобы показать работу оператора UNION, я вставил в каждую таблицу всего лишь по несколько строк:

```
SELECT * FROM NATIONAL;
FirstName LastName  CompleteGames
----------- ----------- ------------------

Sal         Maglie                  11
Don         Newcombe                 9
Sandy       Koufax                  13
Don         Drysdale                12
SELECT * FROM AMERICAN;
FirstName LastName  CompleteGames
----------- ----------- ------------------
Whitey      Ford                    12
Don         Larson                  10
Bob         Turley                   8
Allie       Reynolds                14
SELECT * FROM NATIONAL
UNION
SELECT * FROM AMERICAN;
FirstName LastName  CompleteGames
----------- ----------- ------------------
Allie       Reynolds                14
Bob         Turley                   8
Don         Drysdale                12
Don         Larson                  10
Don         Newcombe                 9
Sal         Maglie                  11
Sandy       Koufax                  13
Whitey      Ford                    12
```

Оператор UNION DISTINCT работает так же, как и оператор UNION без ключевого слова DISTINCT. В обоих случаях дублирующие строки удаляются из конечной совокупности.

***Внимание:***
*Звездочка ('*') используется для обозначения всех столбцов, имеющихся в таблице. Это сокращенное обозначение работает в большинстве случаев прекрасно, но если реляционные операторы используются во встроенном или модульном коде SQL, то это обозначение может доставить массу неприятностей. Что если в одну из таблиц или сразу во все будут добавлены дополнительные столбцы? Тогда эти таблицы больше не будут совместимыми для объединения, и программа перестанет работать. И даже если во все таблицы, для обеспечения совместимости по*

*операции объединения, будут добавлены одни и те же столбцы, то программа скорее всего не будет готова работать с этими дополнительными данными. Таким образом, лучше явно перечислять нужные столбцы, а не полагаться на сокращение '*'. Но при вводе с консоли "одноразовой" команды SQL звездочка работает прекрасно. Если вдруг запрос не сработает, всегда можно быстро вывести структуру таблицы*.

Операция UNION обычно убирает любые повторяющиеся строки, которые появляются в результате ее выполнения. В большинстве случаев это желаемый результат. Впрочем, иногда повторяющиеся строки требуется сохранять. В таких случаях используйте UNION ALL (объединение всех).

Обратимся к нашему примеру и предположим, что Боб Тарли по кличке "Буллит" ("пуля") был "продан" в середине сезона из команды "Нью-Йорк Янкиз", входящей в Американскую лигу, в "Бруклин Доджерс" из Национальной лиги. А теперь предположим, что в каждой команде за сезон у этого питчера было по восемь игр, в течение которых он бессменно подавал мяч. Обычный оператор UNION, показанный в примере, отбросит одну из двух строк с данными об этом игроке. И хотя будет казаться, что за сезон он полностью провел на подаче мяча только восемь игр, но ведь на самом деле таких игр – замечательный результат – было 16. Корректные данные можно получить с помощью следующего запроса:

```
SELECT * FROM NATIONAL
UNION ALL
SELECT * FROM AMERICAN;
```

Иногда оператор UNION можно применять и к двум таблицам, которые не являются совместимыми для объединения. Если в таблицу, которая должна получиться, войдут столбцы, имеющиеся в обеих исходных таблицах и являющиеся совместимыми, то можно использовать оператор UNION CORRESPONDING (объединение соответствующих). В этом случае учитываются только указанные столбцы, и только они войдут в получившуюся таблицу.

Полностью отличаются друг от друга данные, которые бейсбольные статистики собирают по питчерам и игрокам в дальней части поля. Однако каждый раз и в том и в другом случае записываются имя (first name), фамилия (last name) игрока, выходы на поле (putouts), ошибки (errors) и доля принятых мячей (fielding percentage). Конечно, по игрокам в дальней части поля не собирают данные о выигрышах/проигрышах (won/lost record), остановленных мячах (saves) или другие сведения, относящиеся только к подаче мяча. Но все равно, чтобы получить некоторую общую информацию об умении играть в защите, можно выполнять оператор UNION, который будет брать данные из двух таблиц – OUTFIELDER (игрок в дальней части поля) и PITCHER (питчер):

```
SELECT *
FROM OUTFIELDER
UNION CORRESPONDING
(FirstName, LastName, Putouts, Errors, FieldPct)
SELECT *
FROM PITCHER;
```

В результате получается таблица, где для каждого питчера или игрока в дальней части поля указаны имя и фамилия, а также количество выходов на поле, ошибок и доля принятых мячей. Здесь, как и при использовании простого оператора UNION, повторяющиеся строки удалены. Таким образом, если игрок некоторое время играл в дальней части поля, а также был питчером, то при выполнении оператора UNION CORRESPONDING часть статистики этого игрока будет потеряна. Чтобы этого не случилось, используйте UNION ALL CORRESPONDING (объединение всех соответствующих).

***Совет:***
*В списке, находящемся сразу за ключевым словом CORRESPONDING (соответствующие), должны быть только те имена столбцов, которые имеются во всех объединяемых таблицах. Если этот*

*список имен будет пропущен, то будет неявно использован полный список имен. Однако, если в одну или несколько таблиц будут добавлены новые столбцы, этот неявный список может измениться. Так что имена столбцов лучше не пропускать, а указывать явно*.

## INTERSECT

В результате выполнения оператора UNION создается таблица, где появляются все строки, которые могут находиться в какой-либо из исходных таблиц. А если нужны только те строки, каждая из которых находится одновременно во всех исходных таблицах, то можно использовать оператор INTERSECT (пересечь). Он является реализацией в SQL оператора пересечения из реляционной алгебры. Выполнение INTERSECT будет показано на примере из воображаемого мира, в котором Боб Тарли был в середине сезона "продан" команде "Доджерс".

SELECT * FROM NATIONAL;

| FirstName | LastName | CompleteGames |
|---|---|---|
| Sal | Maglie | 11 |
| Don | Newcombe | 9 |
| Sandy | Koufax | 13 |
| Don | Drysdale | 12 |
| Bob | Turley | 8 |

SELECT * FROM AMERICAN;

| FirstName | LastName | CompleteGames |
|---|---|---|
| Whitey | Ford | 12 |
| Don | Larson | 10 |
| Bob | Turley | 8 |
| Allie | Reynolds | 14 |

В таблице, полученной в результате выполнения оператора INTERSECT, будут показаны только те строки, которые находятся одновременно во всех исходных таблицах:

```
SELECT *
FROM NATIONAL
INTERSECT
SELECT *
FROM AMERICAN;
```

| FirstName | LastName | CompleteGames |
|---|---|---|
| Bob | Turley | 8 |

В полученной таким образом таблице сообщается, что Боб Тарли был единственным питчером, который и в той и в другой лиге бессменно подавал мяч в течение одного и того же количества игр. Обратите внимание, что, как и в случае с UNION, INTERSECT DISTINCT выдает тот же результат, что и оператор INTERSECT, используемый без ключевого слова. В этом примере возвращается только одна строка с именем Боба Тарли.

Роль ключевых слов ALL и CORRESPONDING в операторе INTERSECT такая же, как и в операторе UNION. Если используется ALL, то получится таблица, в которой повторяющиеся строки остаются. А когда используется CORRESPONDING, то исходные таблицы не обязательно должны быть совместимыми для объединения, хотя у соответствующих столбцов должны быть одинаковые тип и длина.

Проанализируем следующий пример. В муниципалитете хранят данные о пейджерах, используемых полицейскими, пожарниками, уборщиками улиц и другими работниками городского хозяйства. Данные обо всех ныне используемых пейджерах находятся в таблице PAGERS (пейджеры) из базы данных. А данные обо всех пейджерах, которыми по той или иной причине не пользуются, находятся в другой таблице, OUT (вышедший из строя), имеющей такую же структуру, что и PAGERS. Информация ни по одному из пейджеров не может одновременно быть в двух таблицах. Выполнив оператор INTERSECT, можно проверить, не произошло ли такое ненужное дублирование строк:

```
SELECT *
FROM PAGERS
INTERSECT CORRESPONDING (PagerlD)
SELECT *
FROM OUT;
```

В результате появляется таблица, и если в ней будут находиться какие-либо строки, то это будет означать, что база данных обновлена некорректно. Необходимо проверить все значения, которые появляются в этой таблице в столбце PagerlD (идентификатор пейджера). Ведь пейджер, соответствующий этому идентификатору, либо используется, либо не работает. Одновременно и того и другого не бывает. Обнаружив противоречивые данные, можно провести работы по восстановлению целостности базы данных – выполнить в одной из двух таблиц операцию DELETE (удалить).

## EXCEPT

Оператор UNION выполняется с двумя таблицами и возвращает все строки, которые имеются как минимум в одной из них. Другой же оператор, INTERSECT, возвращает все те строки, которые имеются одновременно в первой и второй таблицах. А оператор EXCEPT (за исключением), наоборот, возвращает все строки, которые имеются в первой таблице, но не имеются во второй.

Теперь вернемся к примеру с базой данных, в которой находится информация о муниципальных пейджерах. Скажем, группа пейджеров, объявленных неработающими, была возвращена поставщику для ремонта, но к настоящему времени эти пейджеры уже исправлены и используются снова. В таблицу PAGERS данные о возвращенных пейджерах уже занесены, но из таблицы OUT их данные по некоторой причине не удалены, хотя это надо было сделать. С помощью оператора EXCEPT можно вывести все номера пейджеров, находящиеся в столбце PagerlD таблицы OUT, за исключением тех номеров, которые принадлежат уже исправленным пейджерам:

```
SELECT *
FROM OUT
EXCEPT CORRESPONDING (PagerlD)
SELECT *
FROM PAGERS;
```

При выполнении этого запроса возвращаются все строки из таблицы OUT, у которых значения PageiTD не находятся в таблице PAGERS.

## Операторы объединения

Операторы UNION, INTERSECT и EXCEPT представляют ценность в тех многотабличных базах данных, таблицы которых являются совместимыми. Однако во многих случаях приходится брать данные из наборов таблиц, имеющих между собой мало общего. Мощными реляционными операторами являются операторы объединения JOIN, в результате выполнения которых данные, взятые из множества таблиц, объединяются в одну. Таблицы из этого множества могут иметь мало общего друг с другом.

Стандарт SQL:2003 поддерживает разные типы операторов объединения. Какой из них лучше всего подходит в конкретной ситуации – это зависит от того результата, который требуется получить.

## Простой оператор объединения

Любой многотабличный запрос является оператором объединения. Исходные таблицы объединяются в том смысле, что в таблице, полученной в результате этого объединения, будет находиться информация из всех исходных таблиц. Самым простым оператором объединения является оператор SELECT, выполняемый с двумя таблицами и не имеющий никаких ограничителей из предложения WHERE. Так что каждая строка из первой таблицы объединяется с каждой строкой из второй. В результате получается таблица, которая является декартовым произведением двух исходных таблиц. Понятие декартова произведения обсуждалось в главе 9 в связи с использованием предложения FROM. Количество строк в полученной таблице равно произведению числа строк в первой исходной таблице и числа строк во второй.

Представьте, что вы, например, являетесь в какой-либо компании менеджером по персоналу и что часть вашей работы состоит в том, чтобы вести учет сотрудников. Большинство данных о сотруднике, таких, например, как домашний адрес и номер телефона, не являются особо засекреченными. Однако доступ к некоторым данным, таким, например, как зарплата, должен быть только утех, кто имеет соответствующее разрешение. Чтобы защитить секретную информацию, держите ее в отдельной таблице, имеющей парольную защиту.

Посмотрите на следующие две таблицы:

| **EMPLOYEE (сотрудник)** | **COMPENSATION (компенсация)** |
|---|---|
| ---------- | ---------- |
| EmpID (идентификатор сотрудника) | Employ (сотрудник) |
| FName (имя) | Salary (зарплата) |
| LName (фамилия) | Bonus (премиальные) |
| City (город) | |
| Phone (телефон) | |

Заполните таблицы какими-либо взятыми для примера данными.

| EmpID | FName | LName | City | Phone | Employ | Salary | BONUS | |
|---|---|---|---|---|---|---|---|---|
| 1 | Whitey | Ford | Orange | 555-1001 | 1 | 33000 | 10000 | |
| 2 | Don | Larson | Newark | 555-3221 | 2 | 18000 | 2000 | |
| 3 | Sal | Maglie | Nutley | 555-6905 | 3 | 24000 | 5000 | |
| 4 | Bob | Turley | Passaic | 555-8908 | 4 | 22000 | 7000 | |

Создайте виртуальную таблицу с помощью следующего запроса:

```
SELECT *
FROM EMPLOYEE, COMPENSATION;
```

Вот что вышло:

| EmpID | FName | LName | City | Phone | Employ | Salary | Bonus |
|---|---|---|---|---|---|---|---|
| 1 | Whitey | Ford | Orange | 555-1001 | 1 | 33000 | 10000 |
| 1 | Whitey | Ford | Orange | 555-1001 | 2 | 18000 | 2000 |
| 1 | Whitey | Ford | Orange | 555-1001 | 3 | 24000 | 5000 |
| 1 | Whitey | Ford | Orange | 555-1001 | 4 | 22000 | 7000 |
| 2 | Don | Larson | Newark | 555-3221 | 1 | 33000 | 10000 |
| 2 | Don | Larson | Newark | 555-3221 | 2 | 18000 | 2000 |
| 2 | Don | Larson | Newark | 555-3221 | 3 | 24000 | 5000 |
| 2 | Don | Larson | Newark | 555-3221 | 4 | 22000 | 7000 |
| 3 | Sal | Maglie | Nutley | 555-6905 | 1 | 33000 | 10000 |
| 3 | Sal | Maglie | Nutley | 555-6905 | 2 | 18000 | 2000 |
| 3 | Sal | Maglie | Nutley | 555-6905 | 3 | 24000 | 5000 |
| 3 | Sal | Maglie | Nutley | 555-6905 | 4 | 22000 | 7000 |
| 4 | Bob | Turley | Passaic | 555-8908 | 1 | 33000 | 10000 |
| 4 | Bob | Turley | Passaic | 555-8908 | 2 | 18000 | 2000 |
| 4 | Bob | Turley | Passaic | 555-8908 | 3 | 24000 | 5000 |
| 4 | Bob | Turley | Passaic | 555-8908 | 4 | 22000 | 7000 |

В получившейся таблице, представляющей собой декартово произведение таблиц EMPLOYEE и COMPENSATION, имеется значительный излишек данных. Кроме того, эта таблица не имеет большого смысла. В ней каждая строка из таблицы EMPLOYEE добавляется к каждой строке из таблицы COMPENSATION. Единственными строками в этой таблице, передающими содержательную информацию, являются те, в которых число из столбца EmpID, взятого из таблицы EMPLOYEE, равняется числу из столбца Employ, взятого из таблицы COMPENSATION. В этих строках имя, фамилия и адрес сотрудника объединены с выплатами того же сотрудника.

Когда вы пытаетесь получить из множества таблиц полезную информацию, то декартово произведение, созданное с помощью простого объединения, почти никогда не бывает тем, что вам нужно. Впрочем, почти всегда первым шагом к тому, что вам нужно, бывает декартово произведение. Отфильтровывать из объединения ненужные строки можно с помощью ограничений, указываемых в предложении WHERE. Самым распространенным объединением, использующим фильтрующее предложение WHERE, является объединение, основанное на равенстве.

## Объединение, основанное на равенстве

**Объединение, основанное на равенстве**, – это простое объединение с предложением WHERE, в котором находится условие, определяющее, что значение из одного столбца первой таблицы должно быть равно значению из соответствующего столбца второй таблицы. Если применить такое объединение к таблицам, имеющимся в примере из предыдущего раздела, то можно получить намного более содержательный результат:

```
SELECT *
FROM EMPLOYEE, COMPENSATION
WHERE EMPLOYEE.EmpID = COMPENSATION.Employ;
```

И вот что вышло:

| EmpID | FName | LName | City | Phone | Employ | Salary | Bonus |
|---|---|---|---|---|---|---|---|
| 1 | Whitey | Ford | Orange | 555-1001 | 1 | 33000 | 10000 |
| 2 | Don | Larson | Newark | 555-3221 | 2 | 18000 | 2000 |
| 3 | Sal | Maglie | Nutley | 555-6905 | 3 | 24000 | 5000 |
| 4 | Bob | Turley | Passaic | 555-8908 | 4 | 22000 | 7000 |

В этой таблице зарплаты и премии, расположенные справа, прилагаются к данным о сотрудниках, находящимся слева. Впрочем, лишние данные есть и в этой таблице, так как столбец EmpID повторяет столбец Employ. Исправить этот недостаток можно, сформулировав запрос немного по-другому:

```
SELECT EMPLOYEE.*, COMPENSATION.SALARY, COMPENSATION.Bonus
FROM EMPLOYEE, COMPENSATION
WHERE EMPLOYEE.EmpID = COMPENSATION.Employ;
```

В результате получилось следующее:

| EmpID | FName | LName | City | Phone | Salary | Bonus |
|---|---|---|---|---|---|---|
| 1 | Whitey | Ford | Orange | 555-1001 | 33000 | 10000 |
| 2 | Don | Larson | Newark | 555-3221 | 18000 | 2000 |
| 3 | Sal | Maglie | Nutley | 555-6905 | 24000 | 5000 |
| 4 | Bob | Turley | Passaic | 555-8908 | 22000 | 7000 |

Эта таблица сообщает вам то, что вы хотите знать, при этом не "нагружая" вас никакими лишними данными. Впрочем, писать сам запрос было несколько утомительно. Чтобы избежать двусмысленности, в именах столбцов приходилось явно указывать имена таблиц. Единственная выгода от этого – тренировка пальцев.

Можно облегчить труд по вводу кода SQL, если использовать псевдонимы (или **имена корреляции**). **Псевдоним** – это другое, более короткое имя таблицы. Если переделать предыдущий запрос с помощью псевдонимов, то получится примерно следующее:

```
SELECT E.*, C.Salary, C.Bonus
FROM EMPLOYEE E, COMPENSATION C
WHERE E.EmpID = C.Employ;
```

В этом примере E – это псевдоним для EMPLOYEE, а C – для COMPENSATION. Действие псевдонима ограничено только тем оператором, в котором он определен. После объявления псевдонима в предложении FROM его необходимо использовать в пределах оператора. При этом нельзя одновременно использовать и длинную форму имени таблицы, и псевдоним.

Смешение полных имен с псевдонимами приводит к путанице. Проанализируйте следующий пример:

```
SELECT T1.C, T2.C
FROM T1 T2, T2 T1
WHERE T1.C > T2.C;
```

В этом примере псевдонимом для T1 является T2, а для T2 – T1. Конечно, такой выбор неразумен, однако формально он не противоречит никаким правилам. Если допустить возможность совместного использования полных имен и псевдонимов, невозможно определить, о какой таблице идет речь.

Предыдущий пример с псевдонимами эквивалентен следующему оператору SELECT без них:

```
SELECT T2.C, T1.C
FROM T1, T2
WHERE T2.C > T1.C;
```

Стандарт SQL:2003 позволяет объединять больше двух таблиц. Их максимальное количество зависит от конкретной реализации. Синтаксис, используемый при таких объединениях, аналогичен тому, который применяется в случае двух таблиц:

```
SELECT E.*, C.Salary, C.Bonus, Y.TotalSales
FROM EMPLOYEE E, COMPENSATION C, YTD_SALES Y
WHERE E.EmpID = C Employ
AND C.Employ = Y.EmpNo;
```

Этот оператор проводит с тремя таблицами объединение, основанное на равенстве. При выполнении этого оператора извлекаются данные, хранящиеся в соответствующих столбцах каждой из таблиц. Это делается для того, чтобы можно было получить таблицу, в которой будут имена и фамилии продавцов, число проведенных каждым из них продаж и полученная ими компенсация. Менеджер по продажам сможет быстро увидеть, заслужил ли продавец свое вознаграждение.

***Совет:***
*Если данные о продажах, проведенных продавцами за последние 12 месяцев до текущей даты, будут храниться в отдельной таблице YTD_SALES (продажи за предшествующий год), то производительность и надежность будут выше, чем при хранении этих данных в таблице EMPLOYEE. Данные в EMPLOYEE относительно стабильные. Имя и фамилия человека, его адрес и номер телефона меняются не слишком часто. А данные о продажах за год меняются, наоборот, достаточно часто. Так как в таблице YTD_SALES столбцов меньше, чем в EMPLOYEE, то таблица YTD_SALES, скорее всего, сможет обновляться быстрее. И если при обновлении итогов продаж можно не трогать таблицу EMPLOYEE, то уменьшается риск случайного изменения хранящихся в ней данных.*

## Перекрестное объединение

**CROSS JOIN** (перекрестное объединение) – это ключевое слово для простого объединения, не имеющего предложение WHERE. Поэтому оператор:

```
SELECT *
FROM EMPLOYEE, COMPENSATION;
```

…также может быть записан как:

```
SELECT *
FROM EMPLOYEE CROSS JOIN COMPENSATION;
```

В результате получается декартово произведение (также известное как **перекрестное произведение**) двух исходных таблиц. CROSS JOIN редко дает тот окончательный результат, который вам нужен, но его применение может быть полезно в качестве первого шага в той цепочке манипуляций данными, которая в конце концов приведет к нужному результату.

## Естественное объединение

Частным случаем объединения, основанного на равенстве, является **естественное объединение** (natural join). В предложении WHERE из объединения, основанного на равенстве, проверяется равенство значения из столбца первой исходной таблицы значению из столбца второй. У двух столбцов должны быть одинаковые тип и длина, как, впрочем, у сравниваемых столбцов должно быть одно и то же имя. На самом же деле при естественном объединении равенство проверяется для всех столбцов из первой таблицы, имеющих те же имена, что и соответствующие им столбцы из второй.

Представьте, что в таблице COMPENSATION из предыдущего примера также имеются столбцы Salary и Bonus, но Emploу заменен на ЕтрШ. В таком случае можно выполнить естественное объединение таблиц COMPENSATION и EMPLOYEE. Традиционный синтаксис объединения должен выглядеть примерно так:

```
SELECT E.*, C.Salary, C.Bonus
FROM EMPLOYEE E, COMPENSATION C
WHERE E.EmpID = C.EmpID;
```

Этот запрос является естественным произведением. Для той же самой операции есть и альтернативный синтаксис:

```
SELECT E.*, C.Salary, C.Bonus
FROM EMPLOYEE E NATURAL JOIN COMPENSATION C;
```

## Условное объединение

**Условное объединение** похоже на объединение, основанное на равенстве, но в проверяемом условии присутствие равенства не обязательно (хотя и не исключается). Проверяемым условием может быть любой правильно составленный предикат. Если условие в проверяемой строке выполняется, то эта строка станет частью полученной таблицы. Синтаксис условного объединения немного отличается от того, который вы видели до сих пор. Это отличие состоит в том, что условие содержится в предложении ON (в), а не в WHERE (где).

Скажем, бейсбольному статисту надо знать, какие питчеры из Национальной лиги провели полностью на подаче столько игр, сколько это сделал хотя бы один питчер Американской лиги. Этот вопрос предназначен для объединения, основанного на равенстве, а также может быть выражен с помощью синтаксиса условного объединения:

```
SELECT *
FROM NATIONAL JOIN AMERICAN
ON NATIONAL.CompleteGames = AMERICAN.CompleteGames;
```

## Объединение по именам столбцов

**Объединение по именам столбцов** похоже на естественное, но является более гибким. При естественном объединении проверяется равенство значений из всех одноименных столбцов, которые только имеются в исходных таблицах. А что касается объединения по именам столбцов, то в нем можно выбирать, какие одноименные столбцы должны проверяться, а какие – нет. Если хотите, то можете выбрать их все, фактически превращая объединение по именам столбцов в естественное. Можете также выбрать и меньшее количество этих столбцов. Таким образом, есть возможность в большей степени определить, какие строки из перекрестного произведения должны оказаться в полученной вами итоговой таблице.

Скажем, вы изготовитель шахмат и имеете инвентарную таблицу, в которой хранятся данные о белых фигурах, а также другую такую же таблицу, но с данными о черных фигурах. Эти таблицы

называются WHITE (белая) и BLACK (черная), и в каждой из них имеются следующие поля: Piece (фигура), Quant (количество), Wood (дерево). В таблицах хранятся такие данные:

| WHITE | | | BLACK | | |
|---|---|---|---|---|---|
| **Piece** | **Quant** | **Wood** | **Piece** | **Quant** | **Wood** |
| King | 502 | Oak | King | 502 | Ebony |
| Queen | 398 | Oak | Queen | 397 | Ebony |
| Rook | 1020 | Oak | Rook | 1020 | Ebony |
| Bishop | 985 | Oak | Bishop | 985 | Ebony |
| Knight | 950 | Oak | Knight | 950 | Ebony |
| Pawn | 431 | Oak | Pawn | 431 | Ebony |

Для каждой разновидности фигур – короля (King), королевы (Queen), ладьи (Rook), слона (Bishop), коня (Knight), пешки (Pawn), – изготавливаемых из дуба (Oak) или из черного дерева (Ebony), количество белых и черных фигур должно быть равным. Если же равенство нарушено, то это означает, что некоторые фигуры или потеряны, или украдены, и, следовательно, вам надо улучшить условия хранения товара.

При естественном объединении проверяется равенство значений во всех одноименных столбцах. В таком случае получится пустая таблица, потому что в таблице WHITE нет таких строк, где значение в столбце Wood было бы равно какому-либо значению из столбца Wood таблицы BLACK. Таблица, полученная в результате естественного объединения, не позволит определить, пропало что-нибудь или нет. Поэтому надо использовать объединение по именам столбцов, в котором столбец Wood исключается из рассмотрения. Это объединение может быть представлено в таком виде:

```
SELECT *
FROM WHITE JOIN BLACK
USING (Piese, Quant);
```

В результате объединения получается таблица только с теми строками, в которых количество белых и черных фигур, имеющихся на складе, совпадает:

| Piece | Quant | Wood | Piece | Quant | Wood |
|---|---|---|---|---|---|
| King | 502 | Oak | King | 502 | Ebony |
| Rook | 1020 | Oak | Rook | 1020 | Ebony |
| Bishop | 985 | Oak | Bishop | 985 | Ebony |
| Knight | 950 | Oak | Knight | 950 | Ebony |

Внимательный читатель может заметить, что из списка пропали королева и пешка, – признак того, что каких-либо из этих фигур не хватает.

## Внутреннее объединение

Объединения – мистические операторы, и для правильного обращения с ними требуется недюжинная внутренняя сила. Возможно, вы даже слышали о **внутреннем объединении** (inner join), – оно-то и является квинтэссенцией реляционных операций. Я вас разыграл! Во внутренних объединениях вовсе нет ничего таинственного. На самом деле внутренними являются все объединения, о которых уже говорилось в этой главе. Объединение по именам столбцов из последнего примера можно сформулировать и как внутреннее, если воспользоваться следующим синтаксисом:

```
SELECT *
```

```
FROM WHITE INNER JOIN BLACK
USING (Piese, Quant);
```

Результат при этом получится тот же самый.

Внутреннее объединение называется "внутренним", чтобы его можно было отличить от внешнего объединения. Из таблицы, получаемой при внутреннем объединении, выбрасываются все строки, у которых нет соответствующих строк одновременно в обеих исходных таблицах. А при внешнем объединении несоответствующие строки сохраняются. Такая вот между ними разница, и нет в ней ничего метафизического.

## Внешнее объединение

При объединении двух таблиц в первой из них (назовем ее левой) могут быть строки, которых нет во второй (правой) таблице. И наоборот, в правой таблице могут быть строки, которых нет в левой. При выполнении внутреннего объединения этих таблиц все несоответствующие строки из вывода удаляются. Однако при **внешнем объединении** (outer join) такие строки остаются. На самом деле любое внешнее объединение бывает трех видов: левое, правое и полное.

## Левое внешнее объединение

В запросе, имеющем объединение, левая таблица – это та, которая в операторе запроса предшествует ключевому слову JOIN, а правая – та, которая следует за ним. При **левом внешнем объединении** (left outer join) несоответствующие строки, имеющиеся в левой таблице, в выводе сохраняются, а имеющиеся в правой – из него, наоборот, удаляются.

Чтобы понять работу внешних объединений, представьте себе корпоративную базу данных, в которой хранятся записи о сотрудниках компании, ее отделах и представительствах. Примеры данных этой компании приведены в табл. 10.1-10.3.

**Таблица 10.1**. LOCATION (представительство).

| LOCATION_ID (идентификатор представительства) | CITY (город) |
|---|---|
| 1 | Boston |
| 3 | Tampa |
| 5 | Chicago |

**Таблица 10.2**. DEPT (отдел).

| DEPT_ID (идентификатор отдела) | LOCATION_ID | NAME (название) |
|---|---|---|
| 21 | 1 | Sales |
| 24 | 1 | Admin |
| 27 | 5 | Repair |
| 29 | 5 | Stock |

**Таблица 10.3**. EMPLOYEE (сотрудник).

| EMP_ID (идентификатор сотрудника) | DEPT_ID | NAME (фамилия) |
|---|---|---|
| 61 | 24 | Kirk |
| 63 | 27 | McCoy |

Теперь предположим, что вам нужно просмотреть все данные обо всех сотрудниках, в том числе, в каком отделе и представительстве сотрудник работает. Такую задачу можно выполнить с помощью объединения, основанного на равенстве:

```
SELECT *
FROM LOCATION L, DEPT D, EMPLOYEE E
WHERE L.LocationlD a D.LocationID
AND D.DeptID = E.DeptID;
```

Результат выполнения этого оператора следующий:

1 Boston 24 Admin 61 24 Kirk
5 Chicago 27 Repair 63 27 McCoy

Полученная в результате таблица содержит все данные обо всех сотрудниках, в том числе, в каком отделе и представительстве сотрудник работает. Так как каждый сотрудник компании работает в каком-либо представительстве и в одном из отделов, то для этого примера как раз и подходит объединение, основанное на равенстве.

А теперь предположим, что вам требуются данные как о представительствах, так и связанные с представительствами данные об отделах и сотрудниках. Это "уже совсем другая история", потому что в представительстве может не быть никаких отделов. Поэтому для получения нужных данных используйте, как показано в следующем примере, внешнее объединение:

```
SELECT *
FROM LOCATION L LEFT OUTER JOIN DEPT D
ON (L.LocationID = D.LocationID)
LEFT OUTER JOIN EMPLOYEE E
ON (D.DeptID = E.DeptID);
```

В этом объединении данные берутся из трех таблиц. Сначала объединяются таблицы LOCATION и DEPT. Затем получившаяся таблица объединяется с таблицей EMPLOYEE. Даже если строки из таблицы, расположенной левее оператора LEFT OUTER JOIN, и не имеют соответствующих строк в таблице, расположенной правее этого оператора, они все равно входят в результат. Таким образом, при первом объединении в результат войдут все представительства, даже без отделов. А при втором объединении – войдут все отделы, даже без персонала. И вот какой получается результат:

| | | | | |
|---|---|---|---|---|
| 1 Boston | 24 | 1 Admin | 61 | 24 Kirk |
| 5 Chicago | 27 | 5 Repair | 63 | 27 McCoy |
| 3 Tampa | NULL | NULL | NULL | NULL NULL NULL |
| 5 Chicago | 29 | 5 Stock | NULL | NULL NULL |
| 1 Boston | 21 | 1 Sales | NULL | NULL NULL |

В нем первые две строки такие же, как и строки из предыдущего примера. А в третьей строке в столбцах, относящихся к отделам и сотрудникам, находятся неопределенные значения, потому что в Тампе нет никаких отделов и никто из сотрудников там постоянно не работает. В четвертой и пятой строках находятся данные о складе и об отделе продаж, но в столбцах этих строк, относящихся к сотрудникам, находятся неопределенные значения, так как в этих двух отделах персонала нет. Это внешнее объединение сообщает все то же, что и объединение, основанное на равенстве, а также предоставляет следующую информацию:

- обо всех представительствах компании, с отделами или без таковых;
- обо всех отделах компании, с персоналом или без него.

Нет никакой гарантии, что строки из последнего примера выведены в нужном вам порядке. Этот порядок в различных реализациях может быть разным. Чтобы выводить строки в том порядке, который вам нужен, вставьте предложение ORDER BY в оператор SELECT, как, например, в этот:

```
SELECT *
FROM LOCATION L LEFT OUTER JOIN DEPT D
ON (L.LocationID = D.LocationID)
LEFT OUTER JOIN EMPLOYEE E
ON (D.DeptID = E.DeptID) ORDER BY L.LocationID, D.DeptID, E.EmpID;
```

***Совет:***
*Так как левого внутреннего объединения не существует, то левое внешнее объединение можно назвать покороче – левое объединение (в коде SQL это ключевые слова LEFT JOIN).*

## Правое внешнее объединение

Готов поклясться, что вы уже знаете, как ведет себя правое внешнее объединение. И вы правы! **Правое внешнее объединение** (right outer join) сохраняет в выводе несоответствующие строки, взятые из правой таблицы, но удаляет из него несоответствующие строки, взятые из левой. Это внешнее объединение можно использовать с теми же таблицами, что в левом внешнем объединении, и получить при этом те же результаты. Для этого надо, заменив в операторе ключевые слова левого внешнего объединения на ключевые слова правого, поменять порядок следования таблиц на обратный:

```
SELECT *
FROM EMPLOYEE E RIGHT OUTER JOIN DEPT D
ON (D.DeptID = E.DeptID)
RIGHT OUTER JOIN LOCATION L
ON (L.LocationID = D.LocationID);
```

В такой формулировке первое объединение создает таблицу, в которой находятся все отделы, с персоналом или без него. А второе объединение создает таблицу со всеми представительствами, независимо от наличия в них отделов.

Так как правого внутреннего объединения не существует, то правое внешнее объединение можно называть правое объединение (в коде SQL это ключевые слова RIGHT JOIN).

## Полное внешнее объединение

**Полное внешнее объединение** (full outer join) соединяет в себе функции левого и правого внешних объединений. В результате выполнения в выводе остаются несоответствующие строки как из левой, так и из правой таблицы. Проанализируем теперь самый общий вариант корпоративной базы данных, которая уже рассматривалась в предыдущих примерах. В этой базе могут быть:

- представительства без отделов;
- отделы без представительств;
- отделы без сотрудников;
- сотрудники без отделов.

Чтобы показать все представительства, отделы и сотрудников, независимо от того, имеют они соответствующие строки в других таблицах или не имеют, используйте полное внешнее объединение, заданное в следующем виде:

```
SELECT *
FROM LOCATION L FULL JOIN DEPT D
```

```
ON (L.LocationID = D.LocationID)
FULL JOIN EMPLOYEE E
ON (D.DeptID = E.DeptID);
```

***Совет:***
*Так как полного внутреннего объединения не существует, то полное внешнее объединение можно называть, полное объединение (в коде SQL это ключевые слова FULL JOIN).*

## Объединение-слияние

В отличие от других видов объединения, **объединение-слияние** (union join) не пытается искать для строки из левой исходной таблицы хотя бы одну соответствующую строку из правой исходной таблицы. Это объединение создает виртуальную таблицу, в которой находятся все столбцы обеих исходных таблиц. В созданной виртуальной таблице столбцы, взятые из левой исходной таблицы, содержат все строки этой исходной таблицы. В этих строках все столбцы, взятые из правой исходной таблицы, содержат неопределенные значения. И, аналогично, столбцы, взятые из правой исходной таблицы, содержат все строки этой исходной таблицы. А в этих строках все столбцы, взятые из левой исходной таблицы, содержат неопределенные значения. Таким образом, таблица, получившаяся в результате объединения-слияния, содержит все столбцы из обеих исходных таблиц, причем количество ее строк равно сумме количества строк, имеющихся в обеих исходных таблицах.

В большинстве случаев сам по себе результат объединения-слияния лишь промежуточный. В процессе обработки данных он является таблицей с большим количеством неопределенных значений. Впрочем, для получения полезной информации эту таблицу можно использовать вместе с выражением COALESCE (соединение) (см. главу 8).

Предположим, что вы работаете в компании, которая проектирует и производит ракеты, предназначенные для экспериментальных запусков. У вас в работе имеется несколько проектов. Под вашим руководством работают несколько инженеров-проектировщиков, каждый из которых имеет квалификацию в нескольких видах деятельности. Как менеджера вас интересует, какие инженеры в каких видах деятельности имеют квалификацию и над какими проектами работали. В настоящее время эти данные разбросаны по трем таблицам: EMPLOYEE (сотрудник), PROJECTS (проекты) и SKILLS (области квалификации).

В таблице EMPLOYEE хранятся данные о сотрудниках, и ее первичным ключом является EMPLOYEE.EmpID. Каждый проект, над которым работал сотрудник, занимает одну строку в другой таблице – PROJECTS. PROJECTS.EmpID – это внешний ключ, который ссылается на таблицу EMPLOYEE. В таблице SKILLS для каждого сотрудника перечислены те виды деятельности, в которых он имеет квалификацию. SKILLS.EmpID– внешний ключ, который ссылается на таблицу EMPLOYEE.

В таблице EMPLOYEE для каждого сотрудника имеется в точности одна строка. А в таблицах PROJECTS и SKILLS таких строк может быть сколько угодно, в том числе и ни одной.

Примеры данных, хранящихся в трех указанных таблицах, приведены в табл. 10.4-10.6.

**Таблица 10.4**. Таблица EMPLOYEE.

**EmpID Name (фамилия)**

1 Ferguson
2 Frost
3 Toyon

**Таблица 10.5**. Таблица PROJECTS.

| ProjectName (название проекта) | EmpID |
|---|---|
| X-63 Structure (устройство ракеты X-63) | 1 |
| X-64 Structure (устройство ракеты X-64) | 1 |
| X-63 Guidance (система управления X-63) | 2 |
| X-64 Guidance (система управления X-64) | 2 |
| X-63 Telemetry (телеметрия X-63) | 3 |
| X-64 Telemetry (телеметрия X-64) | 3 |

Как видно в этих таблицах, Фергюсон работал над проектами устройства ракет X-63 и X-64, а также является специалистом по механическому проектированию и расчетам аэродинамической нагрузки.

Теперь предположим, что вы как менеджер хотите увидеть всю информацию обо всех своих сотрудниках. Для этого вы решили применить к таблицам EMPLOYEE, PROJECTS и SKILLS объединение, основанное на равенстве:

```
SELECT *
FROM EMPLOYEE E, PROJECTS P, SKILLS S
WHERE E.EmpID = P.EmpID
AND E.EmpID = S.EmpID;
```

**Таблица 10.6**. Таблица SKILLS.

| Skill (квалификация) | EmpID |
|---|---|
| Mechanical Design (механическое проектирование) | 1 |
| Aerodynamic Loading (расчеты аэродинамической нагрузки) | 1 |
| Analog Design (проектирование аналоговых устройств) | 2 |
| Gyroscope Design (проектирование гироскопов) | 2 |
| Digital Design (проектирование цифровых устройств) | 3 |
| R/F Design (проектирование РЛС) | 3 |

Эту же операцию можно представить в виде внутреннего объединения, используя для этого следующий синтаксис:

```
SELECT *
FROM EMPLOYEE E INNER JOIN PROJECTS P
ON (E.EmpID = P.EmpID)
INNER JOIN SKILLS S
ON (E.EmpID = S.EmpID);
```

Обе формулировки дают одинаковый результат, показанный в табл. 10.7.

**Таблица 10.7**. Результаты внутреннего объединения.

| E.EmpID | E.Name | P.EmpID | ProjectName | S.EmpID | S.Skill |
|---|---|---|---|---|---|
| 1 | Ferguson | 1 | X-63 Structure | 1 | Mechanical Design |
| 1 | Ferguson | 1 | X-63 Structure | 1 | Aerodynamic Loading |
| 1 | Ferguson | 1 | X-64 Structure | 1 | Mechanical Design |

| E.EmpID | E.Name | P.EmpID | ProjectName | S.EmpID | S.Skill |
|---|---|---|---|---|---|
| 1 | Ferguson | 1 | X-64 Structure | 1 | Aerodynamic Loading |
| 2 | Frost | 2 | X-63 Guidance | 2 | Analog Design |
| 2 | Frost | 2 | X-63 Guidance | 2 | Gyroscope Design |
| 2 | Frost | 2 | X-64 Guidance | 2 | Analog Design |
| 2 | Frost | 2 | X-64 Guidance | 2 | Gyroscope Design |
| 3 | Toyon | 3 | X-63 Telemetry | 3 | Digital Design |
| 3 | Toyon | 3 | X-63 Telemetry | 3 | R/F Design |
| 3 | Toyon | 3 | X-64 Telemetry | 3 | Digital Design |
| 3 | Toyon | 3 | X-64 Telemetry | 3 | R/F Design |

Такое расположение данных не отличается слишком большой ясностью. В каждой строке идентификатор сотрудника появляется три раза, а для каждого сотрудника его проекты и виды квалификации указываются несколько раз. Для ответа на возникшие у вас вопросы внутреннее произведение подходит недостаточно хорошо. Более подходящий результат можно получить, используя объединение-слияние с несколькими операторами SELECT. Начнем с простого объединения-слияния:

```
SELECT *
FROM EMPLOYEE E
UNION JOIN PROJECTS P UNION JOIN SKILLS S;
```

Обратите внимание, что в объединении-слиянии нет предложения ON. Дело в том, что сейчас данные не фильтруются, поэтому предложение ON не нужно. Результат, полученный при выполнении этого оператора, приведен в табл. 10.8.

**Таблица 10.8**. Результат операции union join.

| E.EmpID | E.Name | P.EmpID | ProjectName | S.EmpID | S.Skill |
|---|---|---|---|---|---|
| 1 | Ferguson | NULL | NULL | NULL | NULL |
| NULL | NULL | 1 | X-63 Structure | NULL | NULL |
| NULL | NULL | 1 | X-64 Structure | NULL | NULL |
| NULL | NULL | NULL | NULL | 1 | Mechanical Design |
| NULL | NULL | NULL | NULL | 1 | Aerodynamic Loading |
| 2 | Frost | NULL | NULL | NULL | NULL |
| NULL | NULL | 2 | X-63 Guidance | NULL | NULL |
| NULL | NULL | 2 | X-64 Guidance | NULL | NULL |
| NULL | NULL | NULL | NULL | 2 | Analog Design |
| NULL | NULL | NULL | NULL | 2 | Gyroscope Design |
| 3 | Toyon | NULL | NULL | NULL | NULL |
| NULL | NULL | 3 | X-63 Telemetry | NULL | NULL |
| NULL | NULL | 3 | X-64 Telemetry | NULL | NULL |
| NULL | NULL | NULL | NULL | 3 | Digital Design |
| NULL | NULL | NULL | NULL | 3 | R/F Design |

Каждая таблица была расширена справа или слева неопределенными (NULL) значениями, после чего проведено объединение в одну таблицу всех строк, получившихся в результате этого расширения.

Порядок этих строк произвольный и зависит от используемой реализации. Теперь можно представить полученные данные в более "удобоваримой" форме.

Обратите внимание, что для идентификатора сотрудника в таблице есть три столбца, но в любой из строк определенным является только один из них. Вид выводимой таблицы можно улучшить, если использовать для этих столбцов выражение **COALESCE** (соединить). Как уже говорилось в главе 8, это выражение выбирает из переданного ему списка значений первое, не являющееся неопределенным. В данном случае COALESCE выбирает из списка столбцов единственное значение:

```
SELECT COALESCE (E.EmpID, P.EmpID, S.EmpID) AS ID,
E.Name, P.ProjectName, S.Skill
FROM EMPLOYEE E UNION JOIN PROJECTS P
UNION JOIN SKILLS S
ORDER BY ID;
```

Предложение FROM здесь такое же, как и в предыдущем примере, но теперь три столбца EmpID соединяются с помощью выражения COALESCE в один, который называется ID. Кроме того, результат упорядочивается как раз по этому столбцу ID. Что в итоге получилось, показано в табл. 10.9.

В каждой строке этой таблицы имеются данные или о проекте, или о квалификации, но не о том и другом вместе. При чтении результата необходимо вначале определить, какого типа данные в каждой строке. Если в строке столбец ProjectName является определенным, то в ней указан проект, над которым работал сотрудник. А если определенным является столбец Skill, то в строке указаны навыки сотрудника.

**Таблица 10.9**. Результат применения операции union join вместе с предложением.

| ID | Name | ProjectName | Skill |
|---|---|---|---|
| 1 | Ferguson | X-63 Structure | NULL |
| 1 | Ferguson | X-64 Structure | NULL |
| 1 | Ferguson | NULL | Mechanical Design |
| 1 | Ferguson | NULL | Aerodynamic Loading |
| 2 | Frost | X-63 Guidance | NULL |
| 2 | Frost | X-64 Guidance | NULL |
| 2 | Frost | NULL | Analog Design |
| 2 | Frost | NULL | Gyroscope Design |
| 3 | Toyon | X-63 Telemetry | NULL |
| 3 | Toyon | X-64 Telemetry | NULL |
| 3 | Toyon | NULL | Digital Design |
| 3 | Toyon | NULL | R/F Design |

Можно получить чуть более ясный результат, если в оператор SELECT вставить еще одно предложение COALESCE, как это сделано в следующем примере:

```
SELECT COALESCE (E.EmpID, P.EmpID, S.EmpID) AS ID,
E.Name, COALESCE (P.Type, S.Type) AS Type,
P.ProjectName, S.Skill
FROM EMPLOYEE E
UNION JOIN (SELECT "Project" AS Type, *
FROM PROJECTS) P
UNION JOIN (SELECT "Skill" AS Type, *
FROM SKILLS) S
```

```
ORDER BY ID, Type;
```

В первом предложении UNION JOIN таблица PROJECTS заменена вложенным предложением SELECT, которое добавляет к столбцам, взятым из этой таблицы, еще один столбец, P.Type, с постоянным значением "Project" (проект). И, аналогично, во втором предложении UNION JOIN таблица SKILLS заменена другим вложенным предложением SELECT, которое добавляет к столбцам, взятым из этой таблицы, еще один столбец, S.Type, с постоянным значением "Skill" (квалификация). В каждой строке значением P.Type является или NULL, или "Project", а значением S.Type – или NULL, или "Skill".

В списке внешнего предложения SELECT указано выполнение операции COALESCE, при которой два столбца Type должны стать одним, также имеющим имя Type. Затем этот новый столбец Type указывается в предложении ORDER BY, которое таким образом сортирует все строки, чтобы вначале шли строки с проектами, а затем – с квалификационными навыками. Результат показан в табл. 10.10.

**Таблица 10.10**. Усовершенствованный результат применения операции union join вместе с предложением coalesce.

| ID | Name | Type | ProjectName | Skill |
|---|---|---|---|---|
| 1 | Ferguson | Project | X-63 Structure | NULL |
| 1 | Ferguson | Project | X-64 Structure | NULL |
| 1 | Ferguson | Skill | NULL | Mechanical Design |
| 1 | Ferguson | Skill | NULL | Aerodynamic Loading |
| 2 | Frost | Project | X-63 Guidance | NULL |
| 2 | Frost | Project | X-64 Guidance | NULL |
| 2 | Frost | Skill | NULL | Analog Design |
| 2 | Frost | Skill | NULL | Gyroscope Design |
| 3 | Toyon | Project | X-63 Telemetry | NULL |
| 3 | Toyon | Project | X-64 Telemetry | NULL |
| 3 | Toyon | Skill | NULL | Digital Design |
| 3 | Toyon | Skill | NULL | R/F Design |

Полученная теперь таблица представляет собой отчет – причем очень удобный для чтения – об опыте участия в проектах и о квалификации всех сотрудников, перечисленных в таблице EMPLOYEE.

Если учесть количество имеющихся сейчас разных операций объединения (JOIN), то связывание данных из разных таблиц не должно создавать проблему, какой бы ни была структура этих таблиц. Поверьте, что если только в вашей базе имеются какие-либо сырые данные, то в SQJL2OO3 найдутся средства, чтобы их оттуда извлечь, а затем показать в каком-либо содержательном виде.

## ON или WHERE

Роль, которую в объединениях разных видов играют предложения ON и WHERE, бывает достаточно запутанной. Прояснить ситуацию можно с помощью следующих фактов.

- Предложение ON является частью внутренних, левых, правых и полных объединений. В перекрестных объединениях и объединениях-слияниях такого предложения нет, потому что ни одно из них никакой фильтрации данных не выполняет.

- Во внутреннем объединении предложения ON и WHERE логически эквивалентны; одно и то же условие можно указать или с помощью ON, или с помощью WHERE.

Во внешних объединениях предложения ON и WHERE отличаются друг от друга. Предложение WHERE всего-навсего фильтрует строки, возвращаемые предложением FROM. Строки, отбракованные фильтром, просто не попадут в результат. А предложение ON, используемое во внешнем объединении, вначале фильтрует строки перекрестного произведения, а затем добавляет в результат и отбракованные строки, расширенные неопределенными значениями.

# Использование вложенных запросов

В этой главе…

- Извлечение данных из множества таблиц с помощью одного оператора SQL
- Поиск элементов данных путем сравнения значения из одной таблицы с набором значений из другой
- Поиск элементов данных путем сравнения значения из одной таблицы с выбранным с помощью оператора **select** единственным значением из другой
- Поиск элементов данных путем сравнения значения из одной таблицы со всеми соответствующими значениями из другой
- Создание коррелированных подзапросов
- Использование подзапросов для определения строк, которые требуется обновить, удалить или вставить

---

Одним из лучших способов защиты целостности данных является исключение аномалий модификации с помощью нормализации базы данных. Нормализация заключается в разбиении одной таблицы на несколько других по тематическому признаку. Например, нецелесообразно держать в одной таблице данные и о товарах, и об их покупателях, а лучше разнести эти данные по различным таблицам.

Правильно нормализованная база данных имеет множество таблиц. Типичный запрос к такой базе данных работает с данными не менее чем двух таблиц. Один из способов объединения данных нескольких таблиц заключается в использовании оператора объединения (JOIN) или одного из других реляционных операторов (UNION, INTERSECT или EXCEPT). Реляционные операторы позволяют объединить данные нескольких таблиц. Каждый из перечисленных операторов делает это по-своему. Другой способ объединения данных нескольких таблиц состоит в использовании вложенных запросов.

***Совет:***
*В SQL вложенным запросом называется такой запрос, в котором внешний замыкающий оператор содержит подзапрос. Этот подзапрос может сам быть замыкающим оператором для другого вложенного подзапроса. Теоретически число уровней вложенности для подзапросов не ограничено, а на практике зависит от реализации.*

Подзапросы обязательно должны быть операторами SELECT, но самый внешний замыкающий оператор может также быть INSERT, UPDATE или DELETE.

Второй способ извлечения данных из множества таблиц основан на том, что подзапрос может работать с одной таблицей, а его замыкающий оператор – с другой.

Предположим, например, что вы хотите отправить базе данных своей компании следующий запрос: найти всех руководителей отделов старше 50 лет. Если использовать операторы объединения (JOIN), о которых говорилось в главе 10, то можно отправить примерно такой запрос:

```
SELECT D.DeptNo, D.Name, E.Name, E.Age
FROM DEPT D, EMPLOYEE E
WHERE D.ManagerID = E.ID AND E.Age > 50;
```

Где D – это псевдоним для таблицы DEPT (отдел), а E – для таблицы EMPLOYEE (сотрудник). В EMPLOYEE имеется столбец ID (идентификатор), который является первичным ключом, а в DEPT – столбец ManagerlD (идентификатор руководителя отдела). Их значения совпадают для сотрудника, который одновременно является руководителем отдела. Чтобы соединить связанные друг с другом таблицы, используется простое объединение (список таблиц в предложении FROM), а для "отсева" всех строк, не соответствующих заданному выше критерию, – предложение WHERE. Обратите внимание, что в список параметров оператора SELECT включены из таблицы DEPT столбцы DeptNo (номер отдела) и Name (фамилия), а из таблицы EMPLOYEE – столбцы Age (возраст) и Name.

Затем предположим, что вас интересуют строки, отобранные по тому же условию, но только со столбцами таблицы DEPT. Другими словами, нужны отделы, начальники которых старше 50 лет, и вас при этом не интересует, кто именно эти начальники или сколько им лет. Тогда вместо объединения можно написать запрос с подзапросом:

```
SELECT D.DeptNo, D.Name
FROM DEPT D
WHERE EXISTS (SELECT * FROM EMPLOYEE E
WHERE E.ID = D.ManagerID AND E.Age > 50);
```

В этом запросе имеются два новых элемента: ключевое слово EXISTS (существует), а также SELECT * в предложении WHERE из первого предложения SELECT. Это второе предложение SELECT является подзапросом (или подвыборкой), а ключевое слово EXISTS – одним из нескольких рассмотренных в этой главе операторов, предназначенных для использования вместе с подзапросами.

## Зачем использовать подзапрос

Во многих случаях с помощью подзапроса можно получить тот же результат, что и с помощью объединения (JOIN). Как правило, сложность синтаксиса подзапроса сопоставима со сложностью синтаксиса объединения. Выбор способа построения запроса при этом часто становится делом вкуса пользователя базы данных. Одни предпочитают использовать объединения (JOIN), в то время как другие – применять вложенные запросы. Впрочем, иногда бывает так, что получить нужный результат с помощью объединений (JOIN) невозможно. Тогда приходится использовать вложенный запрос или разбивать задачу на несколько операторов SQL и выполнять их поочередно.

## Что делают подзапросы

Подзапросы находятся в предложении WHERE внешнего оператора. Их роль состоит в том, чтобы задавать для этого предложения условия поиска. Разные виды подзапросов дают разные результаты. Некоторые подзапросы создают список значений, который затем передается замыкающему оператору. Другие подзапросы создают единственное значение, которое затем проверяется замыкающим оператором с помощью оператора сравнения. Существуют также и подзапросы, возвращающие логические значения.

## Вложенные подзапросы, которые возвращают наборы строк

Предположим, что вы работаете на фирме по сборке компьютерных систем. В вашей компании, Zetec Corporation, из покупаемых комплектующих собирают системы, которые затем продают другим компаниям и правительственным агентствам. Информацию о своем бизнесе вы храните в реляционной базе данных. Она состоит из множества таблиц, но сейчас вас интересуют только три: PRODUCT (товар), COMP_USED (использованные компоненты) и COMPONENT (компонент). В таблице PRODUCT содержится список всех выпускаемых вашей фирмой стандартных товаров (табл. 11.1). В таблице COMPONENT перечисляются производственные компоненты товаров (табл. 11.2), а в таблице COMP_USED хранятся данные о том, из каких компонентов состоят произведенные товары (табл. 11.3).

**Таблица 11.1**. Таблица PRODUCT.

| Столбец | Тип | Ограничения |
|---|---|---|
| Model (модель) | Char (6) | PRIMARY KEY |
| ProdName (название товара) | Char (35) | |
| ProdDesc (описание товара) | Char (31) | |
| ListPrice (цена) | Numeric (9.2) | |

**Таблица 11.2**. Таблица COMPONENT.

| Столбец | Тип | Ограничения |
|---|---|---|
| CompiD (идентификатор компонента) | char (6) | PRIMARY KEY |
| CompType (тип компонента) | char (10) | |
| CompDesc (описание компонента) | char (31) | |

**Таблица 11.3**. Таблица COMPOSED.

| Столбец | Тип | Ограничения |
|---|---|---|
| Model (модель) | char (6) | FOREIGN KEY (ДЛЯ PRODUCT) |
| CompiD (идентификатор компонента) | char (б) | FOREIGN KEY (ДЛЯ COMPONENT) |

Компонент может использоваться во множестве товаров, а товар – состоять из множества компонентов (отношение "многие ко многим"). Такая ситуация может привести к нарушениям целостности данных. Чтобы этого не случилось, создайте промежуточную таблицу COMP_USED, связывающую COMPONENT с PRODUCT. Компонент может быть указан во многих строках COMEMJSED, но в каждой строке этой таблицы указывается только один компонент (отношение "один ко многим"). И, аналогично, товар может быть указан также во многих строках COMP_USED, но в каждой строке этой таблицы также указывается только один товар (еще одно отношение "один ко многим"). С помощью промежуточной таблицы сложное отношение "многие ко многим" разбивается на два относительно простых отношения "один ко многим". Этот процесс упрощения отношений – как раз один из примеров нормализации.

## Подзапросы, перед которыми стоит ключевое слово IN

Один из видов вложенных запросов работает по следующему принципу: одиночное значение сравнивается с набором значений, возвращаемым SELECT. В этом случае используется предикат IN (в):

```
SELECT список_столбцов
FROM таблица
WHERE выражение IN (подзапрос);
```

Проверяется значение выражения, которое находится в предложении WHERE. Если это значение есть в списке, возвращенном подзапросом, то предложение WHERE возвращает логическое значение True, а перечисленные табличные столбцы обрабатываются и добавляются в выводимую таблицу. В подзапросе можно указать или ту же таблицу, что и во внешнем запросе, или же какую-нибудь другую.

Чтобы показать, как работает подобный запрос, я воспользуюсь базой данных компании Zetec. Предположим, что в компьютерной отрасли образовался дефицит мониторов. Под вопросом оказывается выпуск готовых товаров, в состав которых должны входить мониторы. Вы хотите знать, что это за товары. Введите следующий запрос:

```
SELECT Model
FROM COMP_USED
WHERE CompID IN
(SELECT CompID
;FROM COMPONENT
WHERE CompType = 'Monitor');
```

Вначале SQL выполняет запрос самого нижнего уровня, т.е. обрабатывает таблицу COMPONENT, возвращая значения CompID из тех строк, в которых значением CotpType является 'Monitor'. В результате появляется список идентификационных номеров всех мониторов. Затем внешний запрос сравнивает с полученным списком значение CompID из каждой строки таблицы COMP_USED. Если сравнение было успешным, то значение Model из той же строки добавляется в виртуальную таблицу, создаваемую внешним оператором SELECT. В результате появляется список всех моделей ваших товаров, в состав которых входит монитор. Следующий пример показывает, что получится, если этот запрос действительно запустить на выполнение:

```
Model
--------
CX3000
CX3010
CX3020
MX3030
MX3020
MX3030
```

Теперь известно, каких товаров в скором времени не будет у вас на складе. Рекламу этих товаров следует на время, по возможности, свернуть.

Этот вид вложенного запроса предполагает, что подзапрос возвращает единственный столбец, и тип данных для этого столбца совпадает с типом данных аргумента, находящегося перед ключевым словом IN.

## Подзапросы, перед которыми стоит ключевое слово NOT IN

Запрос с ключевым словом IN, приведенный в предыдущем разделе, помог руководству фирмы узнать, какие товары нельзя будет продавать. Хотя это и ценная информация, но на ней много не заработаешь. А вот что действительно надо знать руководству Zetec – какие товары можно будет активно продавать. Руководство фирмы хочет продвигать именно те товары, в состав которых мониторы не входят. Такую информацию можно получить с помощью подзапроса, перед которым стоит ключевое слово NOT IN:

```
SELECT Model
FROM COMP_USED
WHERE Model NOT IN
(SELECT Model
FROM COMP_USED
```

```
WHERE CompID IN
(SELECT CompID
FROM COMPONENT
WHERE CompType = 'Monitor'));
```

В результате выполнения этого оператора получаем следующее:

```
Model
--------
PX3040
PB3050
PX3040
PB3050
```

Здесь надо сказать о двух моментах.

- **В этом запросе имеются два уровня вложенности**. Два подзапроса – это в точности оператор предыдущего запроса. Теперь он вложен во внешний замыкающий оператор SELECT. Он принимает список товаров, в состав которых входят мониторы. Перед SELECT стоит другое ключевое слово – NOT EM. В результате действия внешнего оператора создается еще один список. В нем перечислены модели всех продуктов, за исключением тех, в состав которых входят мониторы.
- **В получившейся виртуальной таблице некоторые строки могут повторяться**. Причина повторений следующая. Название товара, собранного из нескольких компонентов, среди которых нет мониторов, встречается в нескольких строках таблицы COMPJUSED. И каждой такой строке соответствует отдельная строка в получившейся виртуальной таблице.

В этом примере количество строк не является проблемой, потому что получившаяся виртуальная таблица является короткой. Однако в реальной жизни такая таблица может состоять из сотен и тысяч строк. Чтобы не было путаницы, повторяющиеся строки необходимо убирать. Это сделать достаточно легко, если в запрос вставить ключевое слово DISTINCT (различный). Тогда в виртуальную таблицу будут добавляться только те строки, которые отличаются от уже имеющихся:

```
SELECT DISTINCT Model
FROM COMP_USED
WHERE Model NOT IN
(SELECT Model
FROM COMP_USED
WHERE CompID IN
(SELECT CompID
FROM COMPONENT
WHERE CompType = 'Monitor'));
```

Как и ожидалось, результат получился следующий:

```
Model
--------
PX3040
PB3050
```

## Вложенные запросы, возвращающие одно значение

Часто перед подзапросом полезно ставить один из шести операторов сравнения (=, <>, <, <=, >, >=). Это можно делать тогда, когда у выражения, стоящего перед оператором, вычисляется единственное

значение, а подзапрос, стоящий после оператора, также выдает одно значение. Исключением является оператор сравнения, сразу после которого находится квантор (ANY, SOME или ALL).

Чтобы проиллюстрировать случай, когда вложенный подзапрос возвращает единственное значение, вернемся к базе данных корпорации Zetec. В ней имеется таблица CUSTOMER (покупатель), содержащая информацию о компаниях, которые покупают товары Zetec. Кроме того, в ней имеется еще другая таблица, CONTACT (представитель для контакта), с личными данными о контактных представителях каждой компании-клиента. Структура этих таблиц приведена в табл. 11.4 и 11.5.

**Таблица 11.4**. Таблица CUSTOMER.

| Столбец | Тип | Ограничения |
|---|---|---|
| CustiD (идентификатор покупателя) | INTEGER | PRIMARY KEY |
| Company (компания) | CHAR (40) | |
| CustAddress (адрес покупателя) | CHAR (30) | |
| Custcity (из какого города покупатель) | CHAR (20) | |
| Custstate (из какого штата) | CHAR (2) | |
| Custzip (почтовый код покупателя) | CHAR (10) | |
| CustPhone (телефон покупателя) | CHAR (12) | |
| ModLevel | INTEGER | |

**Таблица 11.5**. Таблица CONTACT.

| Столбец | Тип | Ограничения |
|---|---|---|
| CustID | INTEGER | FOREIGN KEY |
| ContFName (имя представителя) | CHAR (10) | |
| ContLName (фамилия представителя) | CHAR (16) | |
| ContPhone (телефон представителя) | CHAR (12) | |
| Continfo (информация о представителе) | CHAR (50) | |

Скажем, вам надо посмотреть контактную информацию о компании Olympic Sales, но вы не помните, какой у этой компании идентификатор в столбце CustiD. Используйте такой вложенный запрос:

```
SELECT *
FROM CONTACT
WHERE CustiD =
(SELECT CustiD
FROM CUSTOMER
WHERE Company = 'Olympic Sales');
```

Результат его выполнения примерно следующий:

| CustiD | ContFName | ContLName | ContPhone | ContInfo |
|---|---|---|---|---|
| 118 | Jerry | Attwater | 505-876-3456 | Will play major role in coordinating the wireless Web. |

В последнем столбце говорится, что этот представитель занимается вопросами, как-то связанными с беспроводным доступом в Internet. Так что можете теперь позвонить Джерри Эттуотеру в Olympic и рассказать ему о специальной продаже сотовых телефонов, подключаемых к Internet.

Если в сравнении '=' используется подзапрос, то в списке SELECT этого подзапроса должен находиться один столбец (CustiD в этом примере). Подзапрос должен возвратить только одну строку. Это необходимо для того, чтобы в сравнении было одно значение.

В этом примере я предполагаю, что в таблице CUSTOMER находится только одна строка, в которой столбец Company содержит значение 'Olympic Sales'. Если в операторе CREATE TABLE, с помощью которого была создана таблица CUSTOMER, для столбца Company было установлено ограничение UNIQUE (уникальный), то это дает гарантию, что подзапрос в предыдущем примере возвратит только одно значение (или вообще ни одного). Однако подзапросы, похожие на тот, что используется в примере, обычно используются со столбцами, для которых это ограничение не установлено. В этих случаях, чтобы значения в столбце не повторялись, приходится полагаться на другие средства.

А если окажется, что в столбце Company таблицы CUSTOMER находится больше одного значения 'Olympic Sales' (филиалы в разных штатах), то выполнение подзапроса вызовет ошибку.

С другой стороны, если ни один покупатель из CUSTOMER не работает в Olympic Sales, то подзапрос возвратит значение NULL и результатом сравнения будет значение "unknown" (неизвестно). В этом случае итоговая виртуальная таблица будет пустой. Дело в том, что предложение WHERE возвращает только строки, для которых было получено значение True, а строки со значениями False и "unknown" будут отфильтрованы. Такое может, скорее всего, произойти, если по чьей-то ошибке в столбце Company окажется неправильное название, например 'Olumpic Sales'.

Хотя в таких структурах оператор равенства (=) и является самым распространенным, но в них можно использовать и пять остальных операторов сравнения. Для каждой строки из таблицы, которая указана в предложении замыкающего оператора, единственное значение, возвращаемое подзапросом, сравнивается с выражением из предложения WHERE того же оператора. Если результатом сравнения является значение True, то строка добавляется в выводимую виртуальную таблицу.

Если в состав подзапроса будет включена итоговая функция, то он гарантированно возвратит единственное значение. Эти функции всегда возвращают единственное значение. (Об итоговых функциях см. в главе 3.) Естественно, такой подзапрос будет полезен только тогда, когда требуется получить значение именно итоговой функции.

Скажем, вы торговый представитель компании Zetec и, чтобы оплатить неожиданно свалившиеся на вас счета, должны заработать довольно большие комиссионные. У вас не остается другого выхода, кроме как перестать тратить время на мелочевку и сосредоточиться на продаже только самых дорогих товаров. Самый дорогой товар вы определяете с помощью вложенного запроса:

```
SELECT Model, ProdName, ListPrice
FROM PRODUCT
WHERE ListPrice =
(SELECT MAX(ListPrice)
FROM PRODUCT);
```

Это пример вложенного запроса, в котором подзапрос и замыкающий оператор работают с одной и той же таблицей. Подзапрос возвращает единственное значение – максимальную цену из столбца ListPrice таблицы PRODUCT. А внешний запрос возвращает все строки из той же таблицы, имеющие максимальное значение в столбце ListPrice.

В следующем примере показан подзапрос сравнения, в котором используется оператор сравнения, отличный от '=':

```
SELECT Model, ProdName, ListPrice
FROM PRODUCT
WHERE ListPrice <
(SELECT AVG(ListPrice)
FROM PRODUCT);
```

Подзапрос возвращает единственное значение – среднее значение цен, находящихся в столбце ListPrice таблицы PRODUCT. А внешний запрос возвращает все строки из той же таблицы, в которых значение столбца ListPrice меньше этого среднего значения.

Первоначально стандарт языка SQL разрешал иметь в сравнении только один подзапрос, который должен был находиться в правой части запроса. Согласно стандарту SQL:1999 подзапросом может быть любой из двух операндов сравнения и даже оба сразу. А стандарт SQL:2003 поддерживает эту возможность.

## Кванторы ALL, SOME и ANY

Другой способ сделать так, чтобы подзапрос возвращал единственное значение, – поставить перед этим подзапросом оператор сравнения с квантором. В сочетании с оператором сравнения квантор общности ALL (все) и кванторы существования SOME (некоторый) и ANY (какой-либо) обрабатывают список, возвращенный подзапросом, и в результате этот список сводится к единственному значению.

Воздействие этих кванторов на сравнение я проиллюстрирую примером, использующим базу данных из главы 10. В этой базе хранятся данные об играх, во время которых бейсбольные питчеры не менялись на подаче.

Содержимое двух таблиц получено с помощью следующих двух запросов:

SELECT * FROM NATIONAL;

| FirstName | LastName | CompleteGames |
|---|---|---|
| Sal | Maglie | 11 |
| Don | Newcombe | 9 |
| Sandy | Koufax | 13 |
| Don | Drysdale | 12 |

SELECT * FROM AMERICAN;

| FirstName | LastName | CompleteGames |
|---|---|---|
| Whitey | Ford | 12 |
| Don | Larson | 10 |
| Bob | Turley | 8 |
| Allie | Reynolds | 14 |

Теория состоит в том, что питчеры с самым большим количеством игр, бессменно сыгранных на подаче, должны находиться в Американской лиге потому, что в этой лиге разрешены назначенные хиттеры. Один из способов проверить эту теорию– создать запрос, который возвращает всех питчеров Американской лиги, бессменно сыгравших на подаче больше игр, чем все питчеры Национальной лиги. Для этого может быть сформулирован следующий запрос:

```
SELECT *
```

```
FROM AMERICAN
WHERE CompleteGames > ALL
(SELECT CompleteGames FROM NATIONAL);
```

Вот его результат:

| FirstName | LastName | CompleteGames |
|---|---|---|
| Allie | Reynolds | 14 |

Подзапрос (SELECT CompleteGames FROM NATIONAL) возвращает значения из столбца CompleteGames (количество бессменных игр) для всех питчеров Национальной лиги. Выражение > ALL означает, что надо возвращать только те значения CompleteGames из таблицы AMERICAN, которые больше любого значения, возвращаемого подзапросом. Иными словами, "больше наивысшего значения, возвращаемого подзапросом". В этом случае таким наивысшим значением является 13. В таблице AMERICAN единственной строкой, где находится большее значение, является запись Элли Рейнолдса (АШе Reynolds) с его 14 играми, бессменно сыгранными на подаче.

А что если ваше первоначальное допущение ошибочно? Что если лидером высшей лиги по количеству бессменных игр был все-таки питчер Национальной лиги, несмотря на то, что в Национальной лиге нет назначенного хиттера? Если это так, то запрос:

```
SELECT *
FROM AMERICAN
WHERE CompleteGames > ALL
(SELECT CompleteGames FROM NATIONAL);
```

…возвращает предупреждение о том, что нет строк, удовлетворяющих условиям запроса. А это означает, что в Американской лиге нет такого питчера, которых бессменно пробыл бы на подаче в течение большего количества игр, чем питчер-рекордсмен Национальной лиги.

## Вложенные запросы, которые являются проверкой на существование

Запрос возвращает данные из всех табличных строк, которые удовлетворяют его условиям. Иногда возвращается много строк, а иногда – только одна. Бывает так, что в таблице ни одна строка не удовлетворяет условиям и поэтому ни одна из них не возвращается. Перед подзапросом можно ставить предикаты EXISTS (существует) и NOT EXISTS (не существует). Такая структура, в которой сочетаются подзапрос и один из этих предикатов, сообщает, имеются ли в таблице, указанной в предложении FROM (из) подзапроса, какие-нибудь строки, соответствующие условиям предложения WHERE (где) того же подзапроса.

Подзапросы, перед которыми ставится один из предикатов EXISTS или NOT EXISTS, принципиально отличаются от тех подзапросов, о которых уже говорилось в этой главе. Во всех предыдущих случаях SQL вначале выполняет подзапрос, а затем применяет результат этой операции по отношению к замыкающему оператору. А подзапросы с предикатами EXISTS и NOT EXISTS – это коррелированные подзапросы, и выполняются они по-другому.

**Коррелированный подзапрос** вначале находит таблицу и строку, указанные замыкающим оператором, а затем выполняет подзапрос в той строке его таблицы, которая коррелирует (соотносится) с текущей строкой таблицы замыкающего оператора.

Подзапрос или возвращает одну, или несколько строк, или вообще не возвращает ни одной. Если он возвращает хотя бы одну строку, то предикат EXISTS является истинным и свое действие выполняет замыкающий оператор. В тех же условиях предикат NOT EXISTS является ложным, и замыкающий

оператор свое действие не выполняет. После обработки строки в таблице внешнего оператора та же операция выполняется со следующей строкой. Это действие повторяется до тех пор, пока не будут обработаны все строки из таблицы, указанной замыкающим оператором.

## EXISTS

Допустим, вы являетесь продавцом из Zetec Corporation и хотите позвонить контактным представителям всех калифорнийских организаций, покупающих продукцию Zetec. Попробуйте использовать следующий запрос:

```
SELECT *
FROM CONTACT
WHERE EXISTS
(SELECT *
FROM CUSTOMER
WHERE CustStat" - 'CA'
AND CONTACT.CuSl.? = CUSTOMER.CustID);
```

Обратите внимание на такую ссылку, как CONTACT.CuslTD. Она указывает на столбец из внешнего запроса. Этот столбец сравнивается с другим столбцом, CUSTOMER.CustID, находящемся в таблице внутреннего запроса. Для каждой строки внешнего запроса вы проверяете внутренний запрос, т.е. в предложении WHERE внутреннего запроса используется значение столбца CustID из текущей строки таблицы CONTACT. Эта таблица указана во внешнем запросе.

Столбец CustID связывает таблицу CONTACT с таблицей CUSTOMER. SQL переходит в первую строку таблицы CONTACT, затем находит строку в таблице CUSTOMER, имеющую то же значение CustID, и проверяет в этой строке значение столбца CustState. Если CUSTOMER.CustState = 'CA, то в выводимую таблицу добавляется текущая строка таблицы CONTACT. Точно так же обрабатывается и следующая запись этой таблицы. Так как запрос указывает SELECT * FROM CONTACT, то возвращаются все поля таблицы с данными контактных представителей, в том числе поля с фамилиями и телефонными номерами представителей.

## NOT EXISTS

В предыдущем примере продавец из Zetec хотел узнать имена и телефонные номера представителей для контакта из всех калифорнийских организаций, покупающих продукцию его компании. Предположим, что другой продавец работает со всеми остальными штатами, кроме Калифорнии. Данные о контактных представителях из других штатов можно получить с помощью запроса, похожего на предыдущий, но с предикатом NOT EXISTS:

```
SELECT *
FROM CONTACT
WHERE NOT EXISTS
(SELECT *
FROM CUSTOMER
WHERE CustState = 'CA'
AND CONTACT.CustID = CUSTOMER.CustID);
```

В выводимую таблицу добавляются только те строки из таблицы CONTACT, для каждой из которых подзапрос не возвращает ни одной строки.

## Другие коррелированные подзапросы

Как уже говорилось в предыдущем разделе, подзапросы с ключевым словом IN или оператором сравнения не обязательно должны быть коррелированными, хотя, с другой стороны, такой вариант вполне возможен.

## Коррелированные подзапросы, перед которыми стоит ключевое слово IN

Выше, в разделе "Подзапросы, перед которыми стоит ключевое слово IN", рассказывалось, каким образом некоррелированный подзапрос можно использовать вместе с предикатом IN. А чтобы увидеть, каким образом этот предикат может использоваться, наоборот, коррелированным подзапросом, задайте тот же самый вопрос, что и в случае с предикатом EXISTS. Итак, какие фамилии и телефонные номера у представителей для контакта во всех организациях-покупателях продукции Zetec в Калифорнии? Ответ можно получить с помощью коррелированного подзапроса с IN:

```
SELECT *
FROM CONTACT
WHERE 'CA' IN
(SELECT CustState
FROM CUSTOMER
WHERE CONTACT.CustID = CUSTOMER.CustID);
```

Оператор выполняется с каждой записью таблицы CONTACT. Если значение столбца CustID этой записи совпадает с соответствующим значением столбца таблицы CUSTOMER, то значение CUSTOMER.CustState сравнивается со значением 'CA. Результатом выполнения подзапроса является список, в котором содержится не более одного элемента. Ваш этот единственный элемент представляет собой 'CA', то выполняется условие предложения WHERE из замыкающего оператора и строка добавляется в выводимую запросом таблицу.

## Коррелированные подзапросы, перед которыми стоят операторы сравнения

Как будет показано в следующем примере, перед коррелированным подзапросом может стоять также любой из шести операторов сравнения.

Компания Zetec выплачивает каждому своему продавцу премию, которая зависит от общей суммы, вырученной им от продаж за месяц. Чем выше эта сумма, тем выше процент премии. Список этих процентов хранится в таблице BONUSRATE (ставка премии) со столбцами MIN_AMOUNT (нижняя граница), MAX_AMOUNT (верхняя граница) (процент премии).

| MIN_AMOUNT | MAX_AMOUNT | BONUS_PCT |
|---|---|---|
| 0.00 | 24999.99 | 0 |
| 25000.00 | 49999.99 | 0.001 |
| 50000.00 | 99999.99 | 0.002 |
| 100000.00 | 249999.99 | 0.003 |
| 250000.00 | 499999.99 | 0.004 |
| 500000.00 | 749999.99 | 0.005 |
| 750000.00 | 999999.99 | 0.006 |

Если у продавца ежемесячная сумма продаж составляет 100000-249999.99 долл., то он получает премию в размере 0.3% от этой суммы.

Продажи записываются в главную таблицу сделок TRANSMASTER.

**TRANSMASTER**

| Столбец | Тип | Ограничения |
|---|---|---|
| TRANSID (идентификатор сделки) | INTEGER | PRIMARY KEY |
| CUSTID (идентификатор покупателя) | INTEGER | FOREIGN KEY |
| EMPID (идентификатор сотрудника) | INTEGER | FOREIGN KEY |
| TRANSDATE (дата сделки) | DATE | |
| NET_AMOUNT (облагаемая налогом сумма) | NUMERIC | |
| FREIGHT (стоимость перевозки) | NUMERIC | |
| TAX (налог) | NUMERIC | |
| INVOICETOTAL (итоговая сумма счета-фактуры) | NUMERIC | |

Премии начисляются на основе суммы значений из столбца NET_AMOUNT для всех сделок, которые совершены продавцом за месяц. Размер премии (в процентах) для любого продавца можно найти с помощью коррелированного подзапроса, в котором используются операторы сравнения:

```
SELECT BONUS_PCT
FROM BONUSRATE
WHERE MIN_AMOUNT <=
(SELECT SUM (NET_AMOUNT)
FROM TRANSMASTER
WHERE EMPID = 133)
AND MAX_AMOUNT >=
(SELECT SUM (NET_AMOUNT)
FROM TRANSMASTER
WHERE EMPID =133);
```

Этот запрос интересен тем, что в нем содержатся два подзапроса, для которых приходится использовать логическую связку AND. В подзапросах применяется итоговый оператор SUM, и он возвращает единственное значение – общую сумму продаж за месяц для сотрудника с идентификационным номером 133. Затем это значение сравнивается со значениями в столбцах MIN_AMOUNT и MAX_AMOUNT из таблицы BONUSRATE, и в результате получается процент премии для этого сотрудника.

Если идентификатор продавца, хранящийся в столбце EMPID, вам не известен, но известна фамилия, то такой же ответ можно получить, используя более сложный запрос:

```
SELECT BONUS_PCT
FROM BONUSRATE
WHERE MIN_AMOUNT <=
(SELECT SUM (NET_AMOUNT)
FROM TRANSMASTER WHERE EMPID =
(SELECT EMPID
FROM EMPLOYEE
WHERE EMPLNAME = 'Coffin'))
AND MAX_AMOUNT >=
(SELECT SUM (NET_AMOUNT)
FROM TRANSMASTER WHERE EMPID =
(SELECT EMPID
FROM EMPLOYEE
WHERE EMPLNAME = 'Coffin'));
```

В этом примере, чтобы получить процент премии для сотрудника по фамилии Коффин, используются подзапросы, вложенные в другие подзапросы, а те, в свою очередь, вложены в замыкающий запрос. Эта структура работает только тогда, когда вам наверняка известно, что в компании работает один-единственный сотрудник с этой фамилией. А если вы знаете, что имеются несколько сотрудников с фамилией Коффин? Тогда в предложение WHERE из подзапроса самого нижнего уровня можно добавлять все новые и новые условия, пока не появится уверенность, что будет выбрана единственная строка таблицы EMPLOYEE.

## Подзапросы в предложении HAVING

Коррелированный подзапрос можно задавать не только в предложении WHERE, но и в предложении HAVING. Как уже говорилось в главе 9, перед этим предложением обычно находится предложение GROUP BY. Предложение HAVING действует как фильтр, который должен ограничивать группы, созданные предложением GROUP BY. Группы, которые не удовлетворяют условию предложения HAVING, в результат не попадут.

Если предложение HAVING используется таким образом, то оно проверяется для каждой группы, созданной предложением GROUP BY. Если же предложения GROUP BY нет, то предложение HAVING проверяется для всего набора строк, переданного предложением WHERE. Тогда этот набор считается одной группой. А если нет ни предложения WHERE, ни предложения GROUP BY, то условие предложения HAVING проверяется уже для всей таблицы:

```
SELECT TM1.EMPID
FROM TRANSMASTER TM1
GROUP BY TM1.EMPID
HAVING MAX (TM1.NET_AMOUNT) >= ALL
(SELECT 2 * AVG (TM2.NET_AMOUNT)
FROM TRANSMASTER TM2
WHERE TM1.EMPID <> TM2.EMPID);
```

В этом запросе для одной и той же таблицы используются два псевдонима. В результате можно получить идентификаторы всех тех продавцов, у кого размер максимальной сделки как минимум в два раза превысил средний размер сделок остальных продавцов. Запрос работает следующим образом.

1. Строки таблицы TRANSMASTER группируются внешним запросом по значениям столбца EMPID. Это делается с помощью предложений SELECT, FROM и GROUP BY.
2. Получившиеся группы фильтруются предложением HAVING. В нем для каждой из групп вычисляется (с помощью функции MAX) максимум значений из столбца NET_AMOUNT, которые находятся в строках этой группы.
3. Внутренний запрос дважды проверяет среднее значение NET_AMOUNT для всех тех строк, в которых значения столбца EMPID отличаются от значения этого столбца в текущей группе внешнего запроса. Обратите внимание, что в последней строке запроса приходится указывать два значения, взятые из разных EMPID. Поэтому в предложениях FROM из внешнего и внутреннего запросов приходится для таблицы TRANSMASTER указывать два разных псевдонима.
4. Эти псевдонимы затем используются в сравнении, расположенном в последней строке запроса. Цель их использования состоит в том, чтобы показать – обращение должно идти к значению столбца EMPID из текущей строки внутреннего подзапроса (TM2.EMPID), а также к значению того же столбца, но на этот раз из текущей группы внешнего подзапроса (TM1.EMPID).

## Операторы UPDATE, DELETE и INSERT

Кроме операторов SELECT, предложения WHERE могут быть и в операторах UPDATE, DELETE и INSERT. А в этих предложениях, в свою очередь, могут быть такие же подзапросы, как и в предложениях WHERE, используемых в операторе SELECT.

Например, Zetec только что заключила с Olympic Sales соглашение о партнерстве, согласно которому Zetec "задним числом" предоставляет Olympic Sales десятипроцентную скидку на весь прошлый месяц. Информацию об этой скидке можно ввести в базу данных, используя оператор UPDATE:

```
UPDATE TRANSMASTER
SET NET_AMOUNT = NET_AMOUNT * 0.9
WHERE CUSTID =
(SELECT CUSTID
FROM CUSTOMER
WHERE COMPANY = 'Olympic Sales')
```

В операторе UPDATE можно также использовать и коррелированный подзапрос. Предположим, что в таблице CUSTOMER имеется столбец LAST_MONTHS_MAX (максимум за последние месяцы), а руководство Zetec хочет предоставить скидку для всех сделок, которые превышают значение LAST_MONTHS_MAX данного клиента:

```
UPDATE TRANSMASTER TM
SET NET__AMOUNT = NET_AMOUNT * 0.9
WHERE NET_AMOUNT >
(SELECT LAST_MONTHS_MAX
FROM CUSTOMER C
WHERE C.CUSTID = TM.CUSTID);
```

Обратите внимание, что этот подзапрос является коррелированным. Дело в том, что предложение WHERE, расположенное в последней строке оператора, обращается одновременно и к значению CUSTID из строки, полученной с помощью подзапроса из таблицы CUSTOMER, и к значению CUSTID из текущей строки-кандидата на обновление, которая находится в таблице TRANSMASTER.

Подзапрос в операторе UPDATE может обращаться и к обновляемой таблице. Предположим, что руководство Zetec хочет дать десятипроцентную скидку покупателям, купившим товаров на сумму более 10 000 долларов:

```
UPDATE TRANSMASTER TM1
SET NET_AMOUNT = NET_AMOUNT * 0.9
WHERE 10000 < (SELECT SUM(NET_AMOUNT)
FROM TRANSMASTER TM2
WHERE TM1.CUSTID = TM2.CUSTID);
```

Во внутреннем подзапросе для всех строк таблицы TRANSMASTER, которые относятся к одному и тому же покупателю, вычисляется (с помощью функции SUM) сумма значений из столбца NET_AMOUNT. Что это означает? Предположим, что в таблице TRANSMASTER к покупателю со значением CUSTID, равным 37, относятся четыре строки, в которых столбец NET_ AMOUNT имеет такие значения: 3000, 5000, 2000 и 1000. Для этого значения CUSTID сумма значений NET_AMOUNT равна 11000.

Обратите внимание, что порядок, в котором оператор UPDATE обрабатывает строки, определяется конкретной реализацией и обычно является непредсказуемым. Этот порядок может зависеть от того, каким образом строки хранятся на диске. Предположим, что в имеющейся реализации для значения столбца CUSTID, равного 37, строки таблицы TRANSMASTER обрабатываются в следующем порядке. Первой – строка со значением NET_AMOUNT, равным 3000, затем – с NET_AMOUNT,

равным 5000, и т.д. После обновления первых трех строк со значением CUSTID, равным 37, у них в столбце NET_AMOUNT будут такие значения: 2700 (90% от 3000), 4500 (90% от 5000) и 1800 (90% от 2000). А затем, когда в TRANSMASTER идет обработка последней строки, в которой значение CUSTID равно 37, а NET_AMOUNT равно 1000, то значение функции SUM, возвращенное подзапросом, должно быть равно 10000.

Это значение получается как сумма новых значений NET_AMOUNT из первых трех строк со значением CUSTID, равным 37, а также старого значения из последней строки, имеющей то же значение CUSTID. Таким образом, может показаться, что последняя строка для значния CUSTID, равного 37, не должна обновляться – ведь сравнение с этим значением SUM не будет истинным (10000 не меньше SELECT SUM(NET_AMOUNT)). Но при обращении подзапроса к обновляемой таблице оператор UPDATE работает уже по-другому. В этом операторе при всех проверках подзапросов используются старые значения обновляемой таблицы. В предыдущем операторе UPDATE для столбца CUSTID, равного 37, подзапрос возвращает 11000, т.е. первоначальное значение SUM.

Подзапрос в предложении WHERE работает точно так же, как оператор SELECT или UPDATE. То же самое верно для DELETE или INSERT. Чтобы удалить записи обо всех сделках Olympic Sales, используйте такой оператор:

```
DELETE TRANSMASTER
WHERE CUSTID =
(SELECT CUSTID
FROM CUSTOMER
WHERE COMPANY = 'Olympic Sales');
```

Как и в случае с UPDATE, подзапросы DELETE также могут быть коррелированными и также могут обращаться к изменяемой таблице (у которой в данном случае удаляют строки). Здесь действуют правила, похожие на те, что используются для подзапросов оператора UPDATE. Предположим, вы хотите удалить из таблицы CUSTOMER все строки тех пользователей, для которых итог NET_AMOUNT больше 10000 долларов:

```
DELETE TRANSMASTER TM1
WHERE 10000 < (SELECT SUM(NET_AMOUNT))
FROM TRANSMASTER TM2
WHERE TM1.CUSTID = TM2.CUSTID);
```

Этот запрос удаляет из таблицы TRANSMASTER все строки, в которых столбец CUSTID содержит 37, а также строки, относящиеся к другим пользователям, сумма покупок которых превышает 10000 долларов. Все обращения к TRANSMASTER, имеющиеся в подзапросе, указывают на содержимое этой таблицы, которое было перед любыми удалениями, уже выполненными текущим оператором. Поэтому даже при удалении из таблицы TRANSMASTER последней строки, в которой значение столбца CUSTID равно 37, подзапрос все равно выполняется на этой таблице таким образом, как если бы не было никаких удалений. В итоге подзапрос возвращает значение 11000.

При обновлении, удалении или вставке записей базы данных есть риск сделать так, что данные в изменяемой таблице не будут соответствовать данным в других таблицах из этой базы. Такое несоответствие называется аномалией изменения (см. главу 5). Если из таблицы TRANSMASTER удаляются записи, а от нее зависит другая таблица, TRANSDETAIL (подробности сделок), то записи, соответствующие удаленным записям из первой таблицы, необходимо удалять и из второй. Эта операция называется каскадным удалением, поскольку удаление родительской записи должно вызывать каскад удалений связанных с ней дочерних записей. В противном случае неудаленные дочерние записи становятся "записями-призраками".

В операторе INSERT может находиться предложение SELECT. Такие операторы применяются для заполнения таблиц с текущей информацией. Ниже приведен запрос для создания таблицы с содержимым TRANSMASTER за 27 октября.

```
CREATE TABLE TRANSMASTER_1027
(TRANSID INTEGER, TRANSDATE DATE,
…);
INSERT INTO TRANSMASTER_1027
(SELECT * FROM TRANSMASTER
WHERE TRANSDATE = 2000-10-27);
```

Если требуется информация лишь о крупных сделках, то запрос будет таким:

```
INSERT INTO TRANSMASTER_102 7
(SELECT * FROM TRANSMASTER TM
WHERE TM.NET_AMOUNT > 10000
AND TRANSDATE 2000-10-27);
```

# Рекурсивные запросы

В этой главе…

- Управление рекурсией
- Как определять рекурсивные запросы
- Способы применения рекурсивных запросов

---

SQL-92 и более ранние версии часто критиковали за отсутствие реализации рекурсивной обработки. Многие важные задачи, которые трудно решить другими средствами, легко решаются с помощью рекурсии. В SQL: 1999 появились расширения, позволяющие создавать рекурсивные запросы. Благодаря этим расширениям мощь языка SQL существенно возрастает. Если ваша реализация SQL включает в себя расширения для рекурсии, то вы можете эффективно решать новый большой класс задач. Впрочем, в основной части стандарта SQL:2003 поддержка рекурсии не предусмотрена. Поэтому во многих используемых реализациях этой поддержки может и не быть.

## Что такое рекурсия

Это довольно старая возможность таких языков программирования, как Logo, LISP и C++. В этих языках можно определить функцию (совокупность одной или множества команд), которая выполняет заданную операцию. Главная программа вызывает функцию, выполняя для этого команду, которая называется вызовом функции. В процессе своей работы функция вызывает сама себя – это самая простая форма рекурсии.

Для иллюстрации достоинств и недостатков рекурсии приведем простую программу, в которой одна из функций использует рекурсивные вызовы. Эта программа, написанная на C++, чертит на экране компьютера спираль, начиная с единичного сегмента, направленного вверх.

В ее состав входят три функции.

- Функция line(**n**) чертит отрезок длины **n.**
- Функция Jeft_turn(**d**) поворачивает "чертежный инструмент" на **d** градусов против часовой стрелки.
- Функция spiral(segment), которая определяется следующим образом:
- `void spiral(int segment)`
- `{`
- `line(segment);`
- `left_turn(90);`
- `spiral(seement + 1);`
- `}`

Если из главной программы вызвать spiral(1), то будут выполняться такие действия:

- spiral(1) чертит единичный отрезок (т.е. единичной длины), направленный вверх;
- spiral(1) выполняет поворот на 90 градусов против часовой стрелки;
- spiral(1) вызывает spiral(2);
- spiral(2) чертит отрезок, равный по длине двум единичным и направленный влево;
- spiral(2) выполняет поворот на 90 градусов против часовой стрелки;
- spiral(2) вызывает spiral(3);
- и т.д.

Постепенно благодаря программе появляется спиральная кривая, изображенная на рис. 12.1.

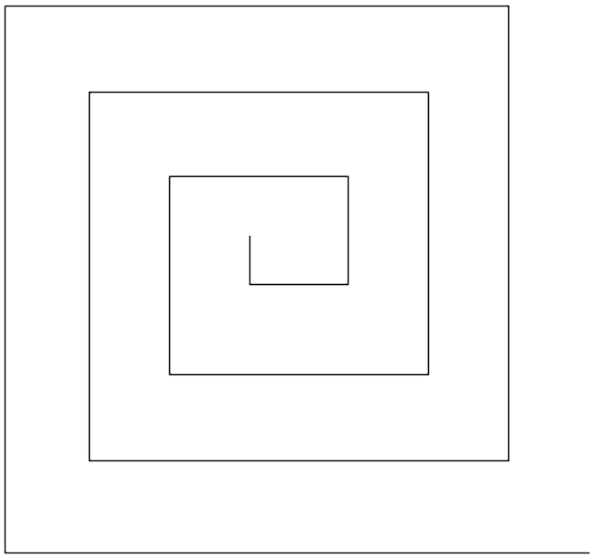

**Рис. 12.1**. Результат вызова spiral(1)

## Маленькие трудности

Ну ладно. Здесь ситуация не такая серьезная, как с Аполлоном-13, когда на пути к Луне прорвало его главный кислородный бак. Но и мы тоже испытываем трудности: наша маленькая программа от нас "убегает". Она все продолжает и продолжает вызывать сама себя и чертит все большие и большие отрезки. Программа будет делать это до тех пор, пока компьютер, пытающийся ее выполнить, не исчерпает свои ресурсы и не выведет на экран сообщение об ошибке. А если вам не повезет, то компьютер просто зависнет.

## Сбой недопустим

Такой сценарий развития событий показывает одну из опасностей, связанных с использованием рекурсии. Программа, написанная для того, чтобы обращаться к себе самой, вызывает на выполнение свой новый экземпляр, а тот, в свою очередь, вызывает еще один, и так до бесконечности. Обычно это не то, что нужно. Чтобы решить проблему, программисты помешают в рекурсивную функцию **условие завершения** – предел того, насколько глубоко должна зайти рекурсия. В результате программа выполняет нужные действия, а затем красиво завершается. Условие завершения мы можем поместить и в нашу программу черчения спиралей, чтобы сберечь ресурсы компьютера и избежать головокружения у программистов:

```
void spiral2(int segment)
{
if (segment <= 10)
{
line(segment);
left_turn(90),
spiral2(segment+ 1);
}
}
```

При вызове программа spiral2(l) выполняется и затем рекурсивно вызывает сама себя до тех пор, пока значение segment не превысит 10. Как только значение segment станет равным 11, конструкция if (segment <= 10) возвратит значение False, и код, находящийся во внутренних скобках, будет пропущен. Управление снова передается предыдущему вызову spiral2(l), а оттуда постепенно возвращается к самому первому вызову, после которого программа завершается.

Каждый раз, когда функция вызывает саму себя, она еще на один уровень удаляется от главной программы – места начала операции. Чтобы эта главная программа продолжила работу, самая последняя итерация (т.е. повторение выполнения) должна вернуть управление предпоследней – той, которая ее вызвала. Предпоследняя итерация обязана поступить точно так же, и процесс продолжается, пока управление не вернется в главную программу, в которой был сделан первый вызов рекурсивной функции.

Рекурсия – это мощный инструмент для повторного выполнения кода. Она идеально подходит для поиска в древовидных структурах, например, в файловых системах, сложных электронных схемах или многоуровневых распределенных сетях.

## Что такое рекурсивный запрос

Рекурсивным называется запрос, который функционально зависит от себя самого. Самой простой формой такой функциональной зависимости является случай, когда внутри оператора запроса Q1 находится вызов этого же запроса. Более сложный случай будет тогда, когда запрос Q1 зависит от запроса Q2, который, в свою очередь, зависит от Q1. И сколько бы запросов ни находилось между первым и вторым вызовом одного и того же запроса, главное, чтобы имела место функциональная зависимость.

## Где можно использовать запрос

Во многих трудных ситуациях рекурсивные запросы помогают сэкономить и время, и нервы. Предположим, например, что у вас есть пропуск, который дает право бесплатно летать любым авиарейсом воображаемой компании Vannevar Airlines. Неплохо, правда? И тут встает вопрос: Куда же можно бесплатно попасть? Все авиарейсы Vannevar Airlines перечислены в таблице FLIGHT (авиарейс), и для каждого из них указан его номер, начальный пункт и место назначения (табл. 12.1).

**Таблица 12.1**. Авиарейсы компании Vannevar Airlines.

| Flight No. (номер авиарейса) | Source (начальный пункт) | Destination (место назначения) |
|---|---|---|
| 3141 | Portland (Портленд) | Orange County (округ Ориндж) |
| 2173 | Portland | Charlotte (Шарлотт) |
| 623 | Portland | Daytona Beach (Дейтона-Бич) |
| 5440 | Orange County | Montgomery (Монтгомери) |

| | |
|---|---|
| 221 Charlotte | Memphis (Мемфис) |
| 32 Memphis | Champaign (Шампейн) |
| 981 Montgomery | Memphis |

Чтобы начать реализацию своего плана проведения отпуска, создайте с помощью SQL в базе данных таблицу FLIGHT:

```
CREATE TABLE FLIGHT (
FlightNo    INTEGER             NOT NULL,
Source      CHARACTER (30),
Destination CHARACTER (30)
);
```

Как только таблица будет создана, ее можно заполнить данными из табл. 12.1.

Предположим, вы хотите лететь из Портленда к своему другу в Монтгомери. Естественно, что вы зададите себе вопросы: "В какие города я попаду самолетами Vannevar Airlines, если начинать с Портленда? А куда я смогу долететь самолетами этой же авиакомпании, если садиться на самолет в Монтгомери?" В некоторые города долететь без промежуточных посадок можно, а в другие – нельзя. По пути в некоторые города придется делать не менее одной такой посадки. Конечно, можно найти все города, куда самолеты Vannevar Airlines могут вас доставить из любого выбранного вами города просто, что называется, "в лоб". Но если вы будете искать города, выполняя один запрос за другим, то тогда вами выбран…

## Трудный способ

Найти то, что хотите узнать, – при условии, что у вас есть терпение и время, – можно с помощью последовательности запросов, в первом из которых начальным пунктом является Портленд:

```
SELECT Destination FROM FLIGHT WHERE Source = "Portland";
```

Этот первый запрос возвращает Orange County, Charlotte и Daytona Beach. Первый из них, если хотите, можно сделать начальным пунктом уже во втором запросе:

```
SELECT Destination FROM FLIGHT WHERE Source = "Orange County";
```

В результате второй запрос возвращает Montgomery. В третьем же запросе можете снова использовать результаты первого запроса, взяв на этот раз в качестве начального пункта второй город:

```
SELECT Destination FROM FLIGHT WHERE Source = "Charlotte";
```

Этот запрос возвращает Memphis. Результаты первого запроса можно использовать ив четвертом, взяв в качестве начального пункта последний из этих результатов:

```
SELECT Destination FROM FLIGHT WHERE Source = "Daytona Beach";
```

Прошу прощения, четвертый запрос возвращает неопределенное значение – у Vannevar Airlines нет авиарейсов из Дейтона-Бич. Но в качестве начального пункта можете также использовать город (Montgomery), который возвращен вторым запросом, что и делается в очередном, пятом, запросе:

```
SELECT Destination FROM FLIGHT WHERE Source = "Montgomery";
```

В результате его выполнения возвращается Memphis, но для вас это не имеет значения. Вы еще раньше узнали, что в этот город попасть можно через Шарлотт. Но Мемфис в качестве начального пункта можно использовать в следующем запросе:

```
SELECT Destination FROM FLIGHT WHERE Source = "Memphis";
```

Этот запрос возвращает Champaign. Им также можно пополнить список городов, куда вы можете попасть (пусть даже с промежуточной посадкой). А так как вас интересуют авиарейсы и с промежуточными посадками, то в запросе в качестве начального пункта можно использовать и этот город:

```
SELECT Destination FROM FLIGHT WHERE Source = "Champaign";
```

Обидно! Запрос возвращает неопределенное значение; оказывается у Vannevar Airlines нет авиарейсов и из Шампейн. (Пока что семь запросов. Они еще не действуют кому-то на нервы?)

Конечно, с помощью этой авиакомпании из Дейтона-Бич улететь нельзя. Так что если вы туда попадете, то там и застрянете. Впрочем, если это случится во время пасхальных каникул – а они, как известно, длятся целую неделю, то особой беды не будет. (Но если вы, чтобы узнать, куда еще можно долететь, будете неделю напролет запускать на выполнение один запрос за другим, то заработаете головную боль похуже, чем от недельного загула.) Или, возможно, застрянете в Шампейн. В этом случае вы можете, кроме всего прочего, поступить в Университет штата Иллинойс и прослушать в нем пару курсов по базам данных.

Конечно, когда-нибудь, со временем, этот метод даст исчерпывающий ответ на вопрос: "В какие города можно попасть из Портленда?" Но отправлять на выполнение один запрос за другим, при этом составляя каждый из них (кроме самого первого) на основе результатов предыдущего, – это работа сложная, требующая много времени, и, скажу прямо, нудная.

## Экономия времени с помощью рекурсивного запроса

Получить нужную информацию будет проще, если создать единственный рекурсивный запрос, который сделает всю работу за одну операцию. Вот его синтаксис:

```
WITH RECURSIVE
ReachableFrom (Source, Destination)
AS (SELECT Source, Destination
FROM FLIGHT
UNION
SELECT in.Source, out.Destination
FROM ReachableFrom in, FLIGHT out
WHERE in.Destination = out.Source
)
SELECT * FROM ReachableFrom
WHERE Source = "Portland";
```

В начале первого прохода, выполняемого во время рекурсии, в таблице FLIGHT будет семь строк, а в ReachableFrom (означает "можно попасть из") – ни одной. Оператор UNION берет семь строк из FLIGHT и копирует их в таблицу ReachableFrom. Тогда в ReachableFrom появятся данные, показанные в табл. 12.2.

**Таблица 12.2**. Таблица ReachableFrom после одного прохода рекурсии.

| Source | Destination |
|---|---|
| Portland | Orange County |

| Source | Destination |
|---|---|
| Portland | Charlotte |
| Portland | Daytona Beach |
| Orange County | Montgomery |
| Charlotte | Memphis |
| Memphis | Champaign |
| Montgomery | Memphis |

Интересное начнется уже при втором проходе. Предложение WHERE (WHERE in. Destination = out. Source)означает, что просматриваются только те строки в которых поле Destination таблицы ReachableFrom равно полю Source таблиш FLIGHT. Для каждой такой строки берутся значения поля Source из ReachableFrom и пол Destination из FLIGHT, а затем в качестве новой строки добавляются в ReachableFrom. Результат этого прохода показан в табл. 12.3.

**Таблица 12.3**. Таблица ReachableFrom после двух проходов рекурсии.

| Source | Destination |
|---|---|
| Portland | Orange County |
| Portland | Charlotte |
| Portland | Daytona Beach |
| Orange County | Montgomery |
| Charlotte | Memphis |
| Memphis | Champaign |
| Montgomery | Memphis |
| Portland | Montgomery |
| Portland | Memphis |
| Orange County | Memphis |
| Charlotte | Champaign |

Эти результаты выглядят намного более полезными. Теперь в таблице ReachableFrom поле Destination содержит все города, в которые можно попасть из любого города, находящегося в поле Source той же таблицы, делая при этом не более одной промежуточной посадки. Затем во время следующего прохода рекурсия обработает маршруты с двумя промежуточными посадками и будет так продолжать до тех пор, пока не будут найдены все города, куда только можно попасть.

После завершения рекурсии третий и последний оператор SELECT (который в рекурсии не участвует) выделяет из ReachableFrom только те города, в которые можно попасть из Портленда. В этом примере можно попасть во все остальные шесть городов, причем с достаточно малым числом промежуточных посадок. Так что вам не придется метаться, как будто вы скачете на ходуле с пружиной.

Если вы внимательно изучите код рекурсивного запроса, то увидите, что он не выглядит проще, чем семь отдельных запросов. Однако у этого запроса есть два преимущества:

- после его запуска постороннее вмешательство больше не требуется;
- он быстро работает.

Если можете, представьте себе настоящую авиакомпанию, у которой на карте ее маршрутов находится намного больше городов. И чем больше возможных мест назначения, тем больше пользы от рекурсивного метода.

Что же делает запрос рекурсивным? То, что мы определяем таблицу ReachableFrom на основе ее самой. Рекурсивной частью определения является второй оператор SELECT, который расположен сразу после UNION. **ReachableFrom** – это временная таблица, которая наполняется данными по мере выполнения рекурсии. И это наполнение продолжается до тех пор, пока все возможные пункты назначения не окажутся в ReachableFrom. Повторяющихся строк в этой таблице не будет, потому что туда их не пропустит оператор UNION. Когда рекурсия завершится, в таблице ReachableFrom окажутся все города, в которые можно попасть из любого города-начального пункта. Третий и последний оператор SELECT возвращает только те города, в которые вы можете попасть из Портленда. Так что желаем приятного путешествия.

## Где еще можно использовать рекурсивный запрос

Любая задача, которую можно представить в виде древовидной структуры, поддается решению с помощью рекурсивного запроса. Классическим примером того, как такие запросы используются в промышленности, является обработка материалов (процесс превращения сырья в конечный продукт). Предположим, ваша компания выпускает новый гибридный бензиново-электрический автомобиль. Такую машину собирают из узлов (двигателя, батарей и т.п.), которые, в свою очередь, состоят из меньших подузлов (коленчатого вала, электродов и пр.), а те – из еще меньших компонентов.

Данные обо всех этих компонентах компонентов сохранять в реляционной базе очень трудно – если, конечно, в ней не используется рекурсия. Рекурсия дает возможность, начав с целой машины, добраться любым путем к самой малой детали. Хотите найти данные о крепежном винте, который держит клемму отрицательного электрода вспомогательной батареи? Это можно – и причем без особых затрат времени. Справляться с такими задачами SQL может с помощью структуры WITH RECURSIVE (рекурсивный оператор).

Кроме того, рекурсия вполне естественна при анализе "что, если?". Например, что произойдет, если руководство авиакомпании Vannevar Airlines решит прекратить полеты из Портленда в Шарлотт? Как это повлияет на полеты в те города, куда сейчас можно добраться из Портленда? Рекурсивный запрос незамедлительно даст ответ на эти вопросы.

# Обеспечение безопасности базы данных

В этой главе…

- Управление доступом к таблицам базы данных
- Принятие решения о предоставлении доступа
- Предоставление полномочий доступа
- Аннулирование полномочий доступа
- Предотвращение попыток несанкционированного доступа
- Предоставление полномочий предоставлять полномочия

---

Системный администратор должен обладать различными специальными знаниями, касающимися работы системы. Поэтому в предыдущих главах рассказывалось о тех частях SQL, с помощью которых создаются базы и выполняется обработка данных. В главе 3 вы познакомились с теми средствами SQL, которые предназначены для защиты базы данных. В этой главе более подробно рассматривается вопрос неправильного использования базы данных.

Администратор базы данных может управлять доступом к ней пользователей, а также определять уровень этого доступа, предоставляя или отзывая доступ к отдельным компонентам системы. Администратор даже может предоставлять или отзывать права на предоставление или отзыв полномочий. Правильное использование средств поддержания безопасности SQL обеспечивает мощную защиту важных данных. В то же время неразумное использование этих средств лишь создает проблемы обычным пользователям, которые всего лишь пытаются выполнить свою работу.

Конечно, в базах данных часто находится жизненно важная информация, к которой нельзя подпускать кого угодно. Поэтому в SQL имеются разные уровни доступа – от полного доступа до полного его отсутствия, и между ними находится несколько уровней частичного доступа. Решив, какие именно операции может выполнять каждый уполномоченный пользователь, администратор базы данных может моментально предоставить пользователю все нужные для его работы права. При этом администратор может ограничить доступ этого пользователя к определенным частям базы данных.

## Язык управления данными как часть SQL

Операторы SQL, используемые для создания баз данных, составляют группу, которая называется **языком определения данных** (**Data Definition Language**, DDL). Создав базу данных, для добавления, изменения или удаления из нее данных можно использовать другие инструкции, известные под собирательным названием **язык манипулирования данными** (**Data Manipu-п**

**Language**, DML). В SQL есть также операторы, которые не попадают ни в одну из этих категорий. Иногда программисты называют эти операторы **языком управления данными** (**Data Control Language**, DCL). Операторы DCL в основном защищают базу данных от несанкционированного доступа, от нежелательных последствий одновременной работы сразу нескольких пользователей, а также от аварий в электрических сетях и неисправностей оборудования. В этой главе рассказывается о защите от несанкционированного доступа.

## Уровни пользовательского доступа

SQL:2003 обеспечивает контролируемый доступ к девяти функциям управления базами данных.

- **Создание, просмотр, обновление и удаление**: эти функции соответствуют операторам INSERT, SELECT, UPDATE и DELETE (см. главу 6).
- **Ссылки** задаются при помощи ключевого слова REFERENCES (см. главы 3 и 5). Ссылки задают ограничения таблицы, зависящей от другой таблицы базы данных.
- **Использование**, которое указывается с помощью ключевого слова USAGE, относится к доменам, наборам символов, сопоставлениям и трансляциям. (О доменах, наборах символов, сопоставлениях и трансляциях см. в главе 5.)
- **Определение новых типов** данных задается с помощью ключевого слова UNDER при работе с типами данных, определяемыми пользователем.
- **Ответ на событие** задается с помощью ключевого слова TRIGGER. Благодаря ему выполняется заданный оператор или целый блок операторов SQL всякий раз при возникновении предопределенного события.
- **Выполнение** стандартной программы задается с помощью ключевого слова EXECUTE.

## Администратор базы данных

В большинстве крупных баз данных с большим количеством пользователей высшей властью обладает **администратор базы данных** (**database administrator**, DBA). У администратора имеются права и полномочия на любые действия с базой данных. Впрочем, администратор еще должен нести и огромную ответственность. Он может легко испортить базу данных и "пустить на ветер" тысячи часов работы. Все администраторы должны ясно и тщательно продумывать те последствия, которые может иметь каждое их действие.

Администратор не только обладает полным доступом к базе данных, но под его контролем также находятся и все права других пользователей. Некоторые избранные пользователи должны получать доступ к большему количеству данных и, возможно, к большему количеству таблиц, чем большинство пользователей.

Самый лучший способ стать администратором – это самостоятельно установить систему управления базой данных. В руководстве по установке находится учетная запись, или **регистрационное имя** (**login**), а также пароль. Это регистрационное имя удостоверяет, что вы являетесь привилегированным пользователем. В системе такой привилегированный пользователь называется администратором базы данных, иногда – **системным администратором**, или **суперпользователем** (к сожалению, ему, в отличие от Супермена, плащ и сапоги не положены). В любом случае первым официальным актом, который вы совершите после ввода учетной записи и пароля (иначе говоря, регистрации), должно стать изменение полученного вами пароля на свой собственный, секретный. Если пароль не будет изменен, то любой, кто прочитает руководство пользователя СУБД, сможет также зарегистрироваться с полным набором полномочий. Вряд ли вам это понравится. Но если изменение сделано, то зарегистрироваться в качестве администратора смогут только те, кто знает ваш новый пароль.

Предпочтительно, чтобы ваш новый пароль администратора базы данных был известен малому кругу особо доверенных людей. Вдруг на вас свалится метеорит или выигрыш в лотерею – всякое может случиться. Вашим коллегам надо иметь возможность работать и в ваше отсутствие. Каждый, кто знает регистрационное имя администратора базы данных и пароль, становится еще одним администратором – следующим после того, кто уже использует эти реквизиты для доступа к системе.

***Совет:***
*Если у вас есть полномочия администратора, регистрироваться в системе в качестве такового следует только тогда, когда нужно выполнять какую-либо специальную задачу, для которой требуются эти полномочия. Как только закончите с задачей, тут же выходите из системы. А для выполнения обычной работы регистрируйтесь с помощью своей личной учетной записи и пароля. Такой подход защитит вас от совершения ошибок, имеющих серьезные последствия для таблиц других пользователей, не говоря уже о ваших собственных таблицах*.

## Владельцы объектов базы данных

Кроме администраторов, есть еще один класс привилегированных пользователей – это **владельцы объектов базы данных**. Такими объектами, например, являются таблицы и представления. Любой пользователь, создающий какой-либо объект базы данных, может назначить владельца этого объекта. Владелец таблицы обладает всеми возможными полномочиями, которые только связаны с этой таблицей, включая управление доступом к ней. Представление создается на основе таблиц, причем владелец представления необязательно должен быть владельцем этих таблиц. Однако в этом случае владелец представления получает на него полномочия, аналогичные имеющимся у него на таблицы, на основе которых это представление создано. Отсюда следует вывод, что нельзя обойти защиту таблицы, принадлежащей другому пользователю, создав на ее основе какое-либо представление.

## Неплохая работенка, но

Вас, вероятно, интересует, как можно стать администратором базы данных и купаться в лучах славы, уважения и восхищения, сопутствующих этой должности. Очевидный ответ состоит в том, чтобы подлизаться к своему боссу. Иногда достаточно демонстрировать компетентность, честность и надежность при выполнении своих ежедневных обязанностей. На самом же деле, главное, что требуется для этой должности, – иметь крепкие нервы. Говоря о славе, уважении и восхищении, я всего лишь шутил. Если с базой данных происходит что-то не то, всегда виноват администратор. А ведь рано или поздно это случается. Так что начинайте тренировать выдержку.

## "Публика"

В сетевой терминологии словом "public" обозначают всех пользователей, не имеющих специальных полномочий администратора или владельцев объектов, и кому эти привилегированные пользователи специально не предоставили права доступа. Если привилегированный пользователь предоставляет некоторые права доступа PUBLIC, их получают все пользователи системы.

В большинстве установленных баз данных пользовательские полномочия представляют собой иерархическую структуру. В этой структуре полномочия администратора находятся на самом высоком уровне, а полномочия рядовых пользователей – на самом низком. Пример иерархической структуры полномочий приведен на рис. 13.1.

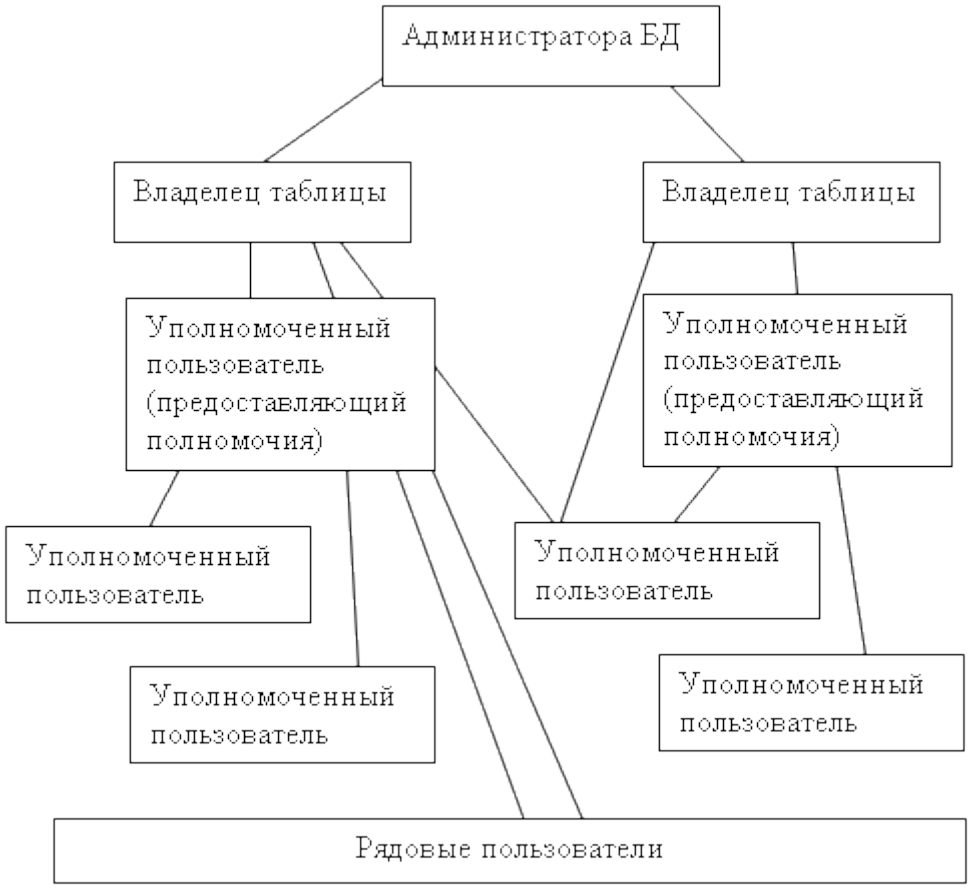


**Рис. 13.1**. Иерархическая структура полномочий доступа

## Предоставление полномочий пользователям

В силу своего положения администратор базы данных имеет все полномочия на все ее объекты. Но, в конце концов, владелец объекта имеет на него все полномочия, а база данных сама является объектом. Ни у кого из пользователей не будет полномочий, относящихся к какому-либо объекту, если только их ему специально не предоставит тот, у которого эти полномочия уже есть (а также право их передавать). Предоставлять полномочия кому-либо другому вы можете с помощью оператора GRANT (предоставить). У этого оператора следующий синтаксис:

```
GRANT СПИСОК-ПОЛНОМОЧИЙ
ON объект
TO список-пользователей
[WITH GRANT OPTION];
```

Предложение WITH GRANT OPTION означает "предоставляющий полномочия"; список полномочий в операторе GRANT определяется следующим образом:

```
ПОЛНОМОЧИЯ [, ПОЛНОМОЧИЯ]…
```

…или:

```
ALL PRIVILEGES
```

В свою очередь, вот как здесь определяются полномочия:

```
SELECT
| DELETE
| INSERT [(имя-столбца[, имя-столбца]…) ]
| UPDATE [(имя-столбца[, имя-столбца]…)]
|REFERENCES [(имя-столбца[, имя-столбца]…)]
| USAGE
| UNDER
| TRIGGER
| EXECUTE
```

А объект в операторе GRANT определяется таким способом:

```
[TABLE] <имя таблицы>
|DOMAIN <имя домена>
| COLLATION <имя сопоставления>
| CHARACTER SET <имя символьного набора>
|TRANSLATION <имя трансляции>
| TYPE <схематмчески обозначенный определенный пользователем тип>
|SEQUENCE <имя генератора последовательности спецификатор указателя шаблона>
```

И наконец, **список пользователей** в операторе определяется следующим образом:

```
регистрационное-имя [, регистрационное-имя]…
| PUBLIC
```

Указанный синтаксис применяется к представлению точно так же, как и к таблице. Полномочия SELECT, DELETE, INSERT, UPDATE и REFERENCES относятся только к таблицам и представлениям. А полномочие USAGE имеет отношение к доменам, наборам символов, сопоставлениям и трансляциям. В последующих разделах приведены различные примеры использования оператора GRANT.

## Роли

Одним из типов идентификатора подтверждения полномочий, причем не единственным, является имя **пользователя**. Это имя удостоверяет пользователя или программу, имеющих полномочия на выполнение с базой данных одной или множества операций. Если в большой организации с большим числом пользователей предоставлять полномочия отдельно каждому сотруднику, то такая операция может занять очень много времени. В стандарте SQL:2OO3 есть понятие роли. Оно-то и помогает решить эту проблему.

**Роль**, определяемая именем, – это набор полномочий, предоставляемый совокупности пользователей, которым нужен одинаковый уровень доступа к базе данных. Например, одинаковые полномочия должны быть у всех пользователей, имеющих роль SecurityGuard (означает "охранник"). Эти полномочия, скорее всего, должны отличаться от тех, что предоставляются пользователям, имеющим роль SalesClerk ("торговый служащий").

***Помни:***
*Так как в основном стандарте SQL:2OO3 ролей нет, то в некоторых реализациях их также может не быть. Перед тем как пытаться использовать роли, проверьте документацию конкретной СУБД.*

Для создания роли можно использовать примерно такой синтаксис:

```
CREATE ROLE SalesClerk;
```

После того как роль создана, вы можете назначить ее тому или иному пользователю с помощью оператора GRANT:

```
GRANT SalesClerk to Becky;
```

Полномочия ролям предоставляются точно так же, как и пользователям, за одним, правда, исключением: роль не будет спорить и жаловаться на вас начальству.

## Вставка данных

Ниже приведен пример предоставления полномочий на вставку данных в таблицу:

```
GRANT INSERT
ON CUSTOMER
TO SalesClerk;
```

Эти полномочия позволяют служащему из отдела продаж добавлять в таблицу CUSTOMER (клиент) новые записи.

## Просмотр данных

А вот пример предоставления полномочий просмотра:

```
GRANT SELECT
ON PRODUCT
TO PUBLIC;
```

Эти полномочия позволяют пользователям системы просматривать содержимое таблицы PRODUCT (товар).

***Внимание:***
*Этот оператор может быть по-настоящему опасным. В столбцах таблицы PRODUCT – например, таких как CostOfGoods (стоимость товаров), – может храниться информация, не предназначенная для всеобщего обозрения. И чтобы предоставить доступ к большей части информации, скрывая при этом важные данные, определите на основе таблицы представление, в котором не будет столбцов с конфиденциальной информацией. Затем предоставляйте полномочия SELECT не на саму таблицу, а на ее представление. Синтаксис этой процедуры приведен ниже.*

```
CREATE VIEW MERCHANDISE AS
SELECT Model, ProdName, ProdDesc, ListPrice
FROM PRODUCT;
GRANT SELECT
ON MERCHANDISE
TO PUBLIC;
```

Пользуясь лишь представлением MERCHANDISE (товары), рядовой пользователь не сможет увидеть CostOfGoods или любой другой столбец из таблицы PRODUCT, за исключением тех четырех, которые перечислены в операторе CREATE VIEW. Это столбцы Model (модель), ProdName (название товара), ProdDesc (описание товара), ListPrice (цена по прейскуранту).

## Модификация табличных данных

В любой работающей организации табличные данные со временем меняются. Поэтому некоторым сотрудникам необходимо предоставить возможность обновлять данные базы, а всем остальным,

наоборот, запретить этим заниматься. Ниже приведен пример предоставления полномочий на обновление.

```
GRANT UPDATE (BonusPct)
ON BONUSRATE
TO SalesMgr;
```

Исходя из рыночной конъюнктуры, менеджер по продажам может регулировать значения премиальных процентов, на основе которых рассчитываются премии продавцов (столбец BonusPct). Однако менеджер не может изменять значения в столбцах Min Amount и Max Amount, определяющие диапазон, который соответствует каждому уровню шкалы премий. Чтобы разрешить модификацию значений всех столбцов таблицы, необходимо указать все имена столбцов или, как в следующем примере, – ни одного:

```
GRANT UPDATE
ON BONUSRATE
TO VPSales;
```

## Удаление из таблицы устаревших строк

Покупатели переезжают в другие города. Сотрудники увольняются, уходят на пенсию или в мир иной. Товары устаревают. Жизнь продолжается, данные базы теряют актуальность. Устаревшие записи необходимо удалять. С другой стороны, следует тщательно контролировать, кто какие записи может удалять. И с этой задачей справится оператор GRANT:

```
GRANT DELETE
ON EMPLOYEE
TO PersonnelMgr;
```

Менеджер по персоналу может удалять записи из таблицы EMPLOYEE (сотрудник). Этим также может заниматься администратор или владелец этой таблицы. Кроме них, записи о сотрудниках больше никто удалять не может (если только кто-то не получит такую возможность благодаря другому оператору GRANT).

## Ссылки для связанных друг с другом таблиц

Если в одной таблице в качестве внешнего ключа находится первичный ключ другой таблицы, то пользователи первой таблицы имеют доступ к данным из второй. Эта ситуация создает потенциально опасную лазейку, через которую неуполномоченные пользователи могут получать секретную информацию. При этом пользователю, чтобы что-то узнать о содержимом таблицы, не нужны никакие права доступа к этой таблице. Если у этого пользователя есть права доступа к первой таблице, которая ссылается на вторую, то этих прав часто бывает достаточно, чтобы иметь доступ и ко второй таблице.

Предположим, например, что в таблице LAYOFF_LIST (список временно уволенных) находятся имена и фамилии сотрудников, которых в следующем месяце временно уволят. Доступ с правом SELECT к этой таблице имеют только уполномоченные сотрудники администрации. Однако один неуполномоченный сотрудник обнаружил, что первичным ключом таблицы LAYOFF_LIST является EmpID (идентификатор сотрудника). Тогда этот сотрудник создает новую таблицу SNOOP ("ищейка"), в которой EmpID является внешним ключом. Этот внешний ключ и дает возможность потихоньку заглядывать в таблицу LAYOFF_LIST. Как создать внешний ключ с помощью предложения REFERENCES (ссылки), см. в главе 5. Все эти приемы должны быть известны каждому системному администратору.

```
CREATE TABLE SNOOP
```

```
(EmpID INTEGER REFERENCES LAYOFF_LIST);
```

Теперь все, что нужно сделать, – это пытаться с помощью оператора INSERT вставить в таблицу SNOOP строки, соответствующие идентификатору каждого сотрудника. Вставки, принимаемые этой таблицей, как раз и относятся к сотрудникам, внесенным в список временно увольняемых, в то время как все отвергаемые вставки – к сотрудникам, отсутствующим в этом списке.

Стандарт SQL:2003 не позволяет таким способом взламывать систему защиты. Он требует, чтобы любые права на использование ссылок предоставлялись уполномоченным пользователем другим пользователям только в явном виде. Каким образом это сделать, показано в следующем примере:

```
GRANT REFERENCES (EmpID)
ON LAYOFF_LIST
TO PERSONNEL_CLERK;
```

## Использование доменов, наборов символов, сопоставлений и трансляций

На безопасность также влияют домены, наборы символов, сопоставления и трансляции. В частности, создавая домены, внимательно следите, чтобы из-за них не пострадала ваша система безопасности.

Можно определить домен, который охватывает какой-либо набор столбцов. Таким образом, у всех этих столбцов был один и тот же тип, а также одни и те же ограничения. Теперь столбцы, создаваемые оператором CREATE TABLE, смогут унаследовать тип и ограничения домена. Конечно, если нужно, то для отдельных столбцов эти характеристики можно перезаписать. Однако домены – это удобное средство, которое дает возможность с помощью одного объявления задавать многочисленные характеристики сразу для целого набора столбцов.

Домены удобны тогда, когда есть множество таблиц, имеющих столбцы с похожими характеристиками. Например, база данных вашей фирмы может состоять из нескольких таблиц. Представим, что в каждой из них находится столбец PRICE (цена), у которого должен быть тип данных DECIMAL(10.2), а значения в этом столбце должны быть не отрицательными и не больше 10000. Тогда, прежде чем создавать таблицы с такими столбцами, нужно создать домен, указывающий характеристики этих столбцов. Создание домена **PriceTypeDomain** (домен типа цены) показано в следующем примере:

```
CREATE DOMAIN PriceTypeDomain DECIMAL (10.2)
CHECK (Price > = 0 AND Price <= 10000);
```

Возможно, в каком-либо наборе таблиц ваши товары будут определяться с помощью столбца ProductCode (код товара), у которого в каждой таблице тип данных составляет CHAR(5), первый символ должен быть X, C или H, а последний – или 9, или 0. Для таких столбцов также можно создать домен, например ProductCodeDomain (домен кода товара), что и делается в следующем примере:

```
CREATE DOMAIN ProductCodeDomain CHAR (5)
CHECK (SUBSTR (VALUE, 1.1) IN ("X", "C", "H")
AND SUBSTR (VALUE, 5.1) IN ("9", "0"));
```

Определив домены, можно приняться за создание таблиц, например таблицы PRODUCT (товар):

```
CREATE TABLE PRODUCT
(ProductCode ProductCodeDomain,
ProductName CHAR (30),
Price PriceTypeDomain);
```

Как только в определении таблицы для поля ProductCode или Price нужно задавать тип данных, указывается соответствующий домен. Таким образом, эти столбцы получают нужные типы данных и, кроме того, для них устанавливаются ограничения, определенные в операторах CREATE DOMAIN.

При использовании доменов возникают вопросы, связанные с безопасностью. Если кто-то другой вдруг захочет использовать созданные вами домены, то может ли такое использование привести к осложнениям? Может. Что если кто-то создаст таблицу со столбцом, в котором используется домен PriceTypeDomain? Пользователь может в этом столбце постепенно увеличивать значения и делать это до тех пор, пока столбец не перестанет их принимать. Таким образом можно будет определить верхнюю границу значений PriceType (тип цены), которую вы указали в предложении CHECK (проверка) оператора CREATE DOMAIN. И если значение этой верхней границы является закрытой информацией, необходимо запретить использовать домен PriceType неуполномоченным пользователям. Чтобы защитить вас в подобных ситуациях, SQL позволяет использовать чужие домены только тем, кому владельцы доменов явно предоставят соответствующее разрешение. Такое разрешение может предоставлять только владелец домена (и, конечно же, администратор). А само предоставление разрешения выглядит так:

```
GRANT USAGE ON DOMAIN PRICE_TYPE TO SALES_MGR;
```

***Внимание:***
*Если к доменам применять оператор DROP, то могут возникнуть проблемы, связанные с безопасностью. И когда вы попытаетесь с помощью DROP отправить домен "в небытие", а в каких-либо таблицах есть столбцы, определенные с помощью этого домена, то такие таблицы станут источником неприятностей. Не исключено, что перед тем как применить оператор DROP к самому домену, этот оператор вначале придется применить к каждой из использующих его таблиц. Возможно, вы обнаружите, что к доменам этот оператор вообще не применим. Действие оператора DROP зависит от конкретной реализации. SQL Server может сильно отличаться в этом смысле от Oracle. Но, видимо, в любом случае придется ограничить круг лиц с разрешением применять оператор DROP к доменам. То же самое относится к наборам символов, сопоставлениям и трансляциям.*

## Инициирование выполнения операторов SQL

В некоторых случаях выполнение одного оператора SQL может вызвать запуск другого оператора или даже целого их блока. Поддержка такой функции (триггерной схемы) и была осуществлена в версии SQL:2003.

Триггер – это механизм, который задает триггер-событие (событие для запуска), время активизации триггера и одно или несколько запускаемых действий. Триггер-событие инициирует запуск, выражаясь простым языком, дает команду "огонь". Время активизации триггера указывает, в какой момент должно произойти действие: непосредственно перед триггер-событием или после него. Запускаемое действие – это выполнение одного или нескольких операторов SQL. При запуске более одного оператора SQL все операторы должны содержаться в пределах структуры BEGIN ATOMIC… END. Само триггер-событие может использовать оператор INSERT, UPDATE или DELETE.

К примеру, вы можете использовать триггер для выполнения оператора, который контролирует истинность новых значений, перед применением обновления данных. Если новые значения будут неверными, обновление данных будет прервано.

Как показано в следующем примере, пользователь или роль должны иметь привилегию на создание триггера:

```
CREATE TRIGGER CustomerDelete BEFORE DELETE
ON CUSTOMER FOR EACH ROW
WHEN State = NY
INSERT INTO CUSTLOG VALUES ('deleted a NY customer'):
```

Теперь при каждом удалении нью-йоркского клиента из таблицы CUSTOMER в регистрационной таблице CUSTLOG будет сделана запись об удалении.

## Предоставление полномочий

Администратор базы данных может предоставить любому пользователю любые полномочия. Владелец объекта также может предоставить любому пользователю любые полномочия, связанные с этим объектом. Однако те, кто получил таким образом свои полномочия, не могут их, в свою очередь, предоставить третьим лицам. Это ограничение позволяет администратору или владельцу объекта в достаточной степени сохранять контроль над ситуацией. Доступ к объекту могут получить только пользователи, уполномоченные на это администратором или владельцем объекта.

Если смотреть с точки зрения безопасности, то представляется разумным ограничить возможность раздавать полномочия доступа. Тем не менее часто пользователям нужны именно права на предоставление полномочий. Конвейер не может остановиться только из-за того, что кто-то заболел, находится в отпуске или ушел на обед. Вы можете дать некоторым пользователям право предоставлять их права доступа надежным сменщикам. Для передачи пользователю такого права в операторе GRANT используется предложение WITH GRANT OPTION (предоставляющий полномочия). Следующий оператор показывает пример того, как можно использовать это предложение:

```
GRANT UPDATE (BonusPct)
ON BONUSRATE
TO SalesMgr
WITH GRANT OPTION;
```

Теперь менеджер по продажам может предоставить права на обновление данных при помощи следующего оператора:

```
GRANT UPDATE (BonusPct)
ON BONUSRATE
TO AsstSalesMgr;
```

После того как этот оператор выполнится, заместитель менеджера по продажам сможет обновлять данные таблицы BONUSRATE, т.е. получит полномочия, которых у него до этого не было.

***Внимание:***
*Приходится искать компромисс между безопасностью и удобством. Владелец таблицы BONUSRATE, предоставляя менеджеру по продажам полномочия UPDATE вместе с атрибутом WITH GRANT OPTION, делится с ним значительной частью своей власти. Ему остается надеяться, что менеджер по продажам серьезно отнесется к этой ответственности и будет осторожен с передачей полномочий другим лицам.*

## Аннулирование полномочий

Наряду с предоставлением полномочий доступа существует необходимость иметь возможность аннулировать эти полномочия. Обязанности сотрудников со временем изменяются, следовательно, изменяются их потребности в доступе к данным. Нередки случаи перехода на работу к конкуренту. В этом случае все полномочия перешедших сотрудников придется отозвать. В SQL удаление

полномочий на доступ выполняется с помощью оператора REVOKE (отозвать). Его синтаксис аналогичен синтаксису оператора GRANT, но только результат получается противоположный.

```
REVOKE [GRANT OPTION FOR] список-полномочий
ON объект
FROM список-пользователей [RESTRICT | CASCADE];
```

С помощью этой структуры можно отзывать перечисленные в списке полномочия, не затрагивая при этом все остальные. Главное отличие между операторами REVOKE и GRANT состоит в том, что в первом из них применяется одно из двух необязательных ключевых слов – RESTRICT (ограничить) или CASCADE (каскадное удаление). Пусть для предоставления полномочий вы использовали оператор GRANT вместе с WITH GRANT OPTION. Тогда применение ключевого слова CASCADE в операторе REVOKE приводит к отзыву указанных полномочий как у того пользователя, которому вы их предоставили, так и у всех пользователей, кому (благодаря атрибуту WITH GRANT OPTION) эти полномочия он уже успел предоставить.

С другой стороны, оператор REVOKE с ключевым словом RESTRICT будет отзывать полномочия пользователя, который никому больше их не предоставлял. Если пользователь уже с кем-то поделился полномочиями, указанными в операторе REVOKE с ключевым словом RESTRICT, то выполнение этого оператора будет прервано и будет выведено сообщение об ошибке.

Оператор REVOKE с необязательным предложением GRANT OPTION FOR (возможность предоставления) можно использовать, чтобы отзывать у пользователя возможность предоставлять указанные полномочия, но оставляя их для самого этого пользователя. Если оператор содержит предложение GRANT OPTION FOR и ключевое слово CASCADE, то отзываются все полномочия, предоставленные пользователем, а также полномочия этого пользователя на предоставление полномочий. А если в операторе есть и GRANT OPTION FOR и RESTRICT, то события развиваются по одному из двух вариантов.

- Если пользователь не предоставил никому другому те полномочия, которые вы у него отзываете, то выполняется оператор REVOKE и удаляет способность этого пользователя предоставлять полномочия.
- Если пользователь уже успел предоставить кому-нибудь хотя бы одно из отзываемых у него полномочий, то полномочия не отзываются, а возвращается код ошибки.

***Внимание:***
*Возможность предоставлять полномочия с помощью предложения WITH GRANT OPTION – да еще в сочетании с выборочным отзывом полномочий – делает обеспечение безопасности вашей системы намного более сложным, чем кажется на первый взгляд. Например, любой пользователь, конечно же, может получить одни и те же полномочия от множества "дарителей". И если один из этих "дарителей" затем отзовет предоставленные им полномочия, у пользователя они все равно останутся. Дело в том, что продолжают действовать те же полномочия, но предоставленные другим "дарителем". Если благодаря предложению WITH GRANT OPTION полномочия передаются от одного пользователя к другому, то такая ситуация порождает цепочку зависимости. В этой цепочке полномочия одного пользователя зависят от таких же полномочий другого.*
*Если вы администратор или владелец объекта, то никогда не забывайте о том, что полномочия, предоставленные с помощью предложения WITH GRANT OPTION, могут "всплыть" в самых неожиданных местах. Достаточно трудным может оказаться отзыв полномочий у нежелательных пользователей при одновременном сохранении их у законных пользователей. Вообще говоря, предложения GRANT OPTION и CASCADE содержат в себе многочисленные ловушки. Поэтому при использовании этих предложений сверяйте стандарт SQL:2003 и документацию по имеющемуся у вас продукту. Это нужно, чтобы четко понимать, каким образом работают конструкции GRANT OPTION и CASCADE.*

## Экономия времени и сил благодаря совместному использованию операторов GRANT и REVOKE

Предоставление множеству пользователей множества полномочий на выбранные столбцы таблицы сопряжено с вводом большого количества кодов. Проанализируйте следующий пример. Вице-президент по продажам хочет, чтобы все те, кто занимается продажами, могли просматривать все содержимое таблицы CUSTOMER (клиент). Но обновлять, удалять или вставлять строки должны только менеджеры по продажам. И никто не должен обновлять поле CustID (идентификатор клиента). Соответствующие полномочия можно предоставить с помощью следующих операторов GRANT:

```
GRANT SELECT, INSERT, DELETE
ON CUSTOMER
TO Tyson, Keith, David;
GRANT UPDATE
ON CUSTOMER (Company, CustAddress, CustCity,
CustState, CustZip, CustPhone, ModelLevel)
TO Tyson, Keith, David;
GRANT SELECT
ON CUSTOMER
TO Jenny, Valerie, Melody, Neil, Robert, Sam,
Brandon, MichelleT, Allison, Andrew,
Scott, MishelleB, Jaime, Linleigh, Matt, Amanda;
```

А теперь попробуем упростить этот код. Все пользователи обладают полномочиями просмотра таблицы CUSTOMER. Менеджеры по продажам имеют на эту таблицу полномочия вставки и удаления, а также могут обновлять любой ее столбец, кроме CustlD. Поэтому тот же результат, что и в предыдущих операторах, можно получить более легким способом:

```
GRANT SELECT
ON CUSTOMER
TO SalesReps
GRANT INSERT, DELETE, UPDATE
ON CUSTOMER
TO Tyson, Keith, David;
REVOKE UPDATE
ON CUSTOMER (CustlD)
FROM Tyson, Keith, David;
```

Это та же защита, что и в предпоследнем примере, и для нее также надо использовать три оператора. Никто не может изменить данные в столбце CustlD. Полномочия INSERT, DELETE и UPDATE имеют только Тайсон, Кейт и Дэвид. Как видно, три последних оператора значительно короче, так как в них не приходится вводить имя каждого сотрудника отдела продаж и перечислять каждый столбец таблицы.

# Защита данных

В этой главе…

- Как избежать повреждения базы данных
- Проблемы, вызванные одновременными операциями
- Решение этих проблем с помощью механизмов SQL
- Задание требуемого уровня защиты с помощью команды set transaction
- Защита данных, не препятствующая нормальной работе с ними

---

Каждый слышал о законе Мерфи, который формулируется обычно так: "Если какая-нибудь неприятность может случиться, она случается". Большую часть времени дела идут хорошо, и мы потешаемся над этим псевдозаконом. Временами нам даже кажется, что мы из тех немногих счастливчиков, над кем не властен один из основных законов мироздания. Даже если неприятности все-таки происходят, то мы обычно легко с ними справляемся.

Однако в очень сложной структуре вероятность возникновения проблем резко возрастает (как сказал бы математик, "приближенно описывается квадратичной зависимостью от сложности системы"). Поэтому большие программные разработки почти всегда появляются с опозданием и часто с немалым количеством невыявленных ошибок. Реальное многопользовательское приложение СУБД является большой и сложной структурой. Во время работы этого приложения существует большая вероятность сбоев. И хотя были разработаны специальные методы разработки приложений, направленные на то, чтобы свести последствия этих сбоев к минимуму, полностью исключить их невозможно. Это хорошая новость для профессионалов, занятых поддержкой баз данных, и специалистов по ремонту. Автоматика, скорее всего, никогда их не заменит.

## Угрозы целостности данных

Киберпространство (в том числе ваша собственная сеть) может быть прекрасным местом для посещения, но для размещенных в нем данных это отнюдь не райский уголок. Данные могут быть повреждены или совсем испорчены самыми разными способами. В главе 5 рассказывалось о проблемах, возникающих в результате ввода неправильных данных, ошибок оператора, злонамеренного повреждения данных и одновременного доступа. "Неаккуратные" команды SQL и неправильно спроектированные приложения также могут повредить данные, и не нужно иметь много фантазии, чтобы представить, как это может произойти. Впрочем, данным угрожают и две очевидные опасности: нестабильность платформы и аппаратные сбои. В данном разделе как раз говорится об этих двух опасностях, а также о неприятностях, связанных с одновременным доступом.

## Нестабильность платформы

**Нестабильность платформы** относится к таким неприятностям, которых даже быть-то не должно, но, увы, они существуют. Чаще всего она проявляется при запуске одного или множества новых и относительно непроверенных компонентов системы. Проблемы могут "притаиться" в новом выпуске СУБД, в новой версии операционной системы или же в новом оборудовании. Проявляются они обычно тогда, когда вы запускаете очень важное задание. В результате система блокируется, а данные – портятся. И вам больше ничего не остается, кроме как ругать последними словами свой компьютер и тех, кто его сделал, – особенно, если не осталось резервной копии.

***Внимание:***
*Никогда не выполняйте ответственное задание в системе, имеющей хотя бы один непроверенный*

*компонент. Не поддавайтесь искушению немедленно перейти на только что появившуюся бета-версию СУБД или операционной системы, даже если эта версия предоставляет расширенную функциональность. По необходимости экспериментируйте с новым программным обеспечением на машине, полностью изолированной от рабочей сети.*

## Аппаратный сбой

Даже хорошо проверенное, высоконадежное оборудование иногда отказывает, отправляя данные на тот свет. Все материальное со временем изнашивается, даже современные полупроводниковые схемы. Если такой отказ происходит тогда, когда база данных открыта, то данные можно потерять, даже не осознавая этого. Опыт показывает – рано или поздно это произойдет. Уж если закон Мерфи и проявляет себя, так только в самое неподходящее время.

Одним из способов защитить данные от отказов оборудования является **резервное копирование.** Сохраняйте лишние копии всего подряд. Если ваша организация может себе это позволить, для обеспечения максимальной безопасности можно продублировать аппаратные средства вместе с их настройками с таким расчетом, чтобы, если потребуется, можно было быстро установить и запустить резервные копии базы данных и приложений на резервном оборудовании. Ну а если ограниченность в средствах не позволяет дублировать все подряд, включая расходы? Тогда, по крайней мере, делайте достаточно часто резервные копии вашей базы данных и приложений – причем настолько часто, чтобы вам после неожиданного отказа не пришлось заново вводить слишком много данных.

Другой способ избежать наихудших последствий аппаратных сбоев – использование транзакций. Это основная тема данной главы. **Транзакция** – это неделимая единица работы. Или транзакция выполняется целиком, или не выполняется вовсе. Если этот подход "все или ничего" кажется вам слишком радикальным, примите к сведению, что самые большие проблемы возникают в результате неполного выполнения операции с базой данных.

## Одновременный доступ

Представим, что программы и оборудование, с которыми вы работаете, проверены, данные введены правильно, приложения свободны от ошибок, а оборудование абсолютно надежно. Получается, что данным ничего не грозит? К сожалению, нет. Если несколько людей одновременно пытаются использовать одну и ту же таблицу из базы данных, создается ситуация одновременного доступа и их компьютеры соревнуются за право первоочередного доступа. Многопользовательские системы баз данных должны иметь возможность эффективно разрешать возникающие коллизии одновременного доступа.

## Проблемы взаимодействия транзакций

Проблемы, связанные с одновременным доступом, возникают даже в относительно простых приложениях. Представьте, например, такой случай. Вы пишете приложение, которое предназначено для обработки заказов и включает в себя четыре таблицы: ORDER_MASTER (главная таблица заказов), CUSTOMER (таблица клиентов), LINE_ITEM (строка заказа) и INVENTORY (таблица с описанием товаров). Выполняются следующие условия.

- В таблице ORDERJVlASTER первичным ключом является поле OrderNumber (номер заказа), а поле CustomerNumber (номер клиента) – внешним ключом, который ссылается на таблицу CUSTOMER.
- В таблице LINEJTEM первичным ключом является поле LineNumber (номер строки), а поле ItemNumber (идентификационный номер товара) – внешним ключом, который ссылается на таблицу INVENTORY, и, наконец, одним из ее полей является Quantity (количество).

- В таблице INVENTORY первичным ключом является поле ItemNumber; кроме того, в ней есть поле QuantityOnHand (количество в наличии).
- Во всех трех таблицах есть еще и другие столбцы, но они в этом примере не рассматриваются.

Политика вашей компании состоит в том, чтобы каждый заказ или выполнять полностью, или не выполнять вовсе. Частичные выполнения заказов не допускаются. (Спокойно. Это же воображаемая ситуация.) Вы пишете приложение ORDER_PROCESSING (обработка заказа), которое должно обрабатывать в таблице ORDER_MASTER каждый новый заказ, причем делать это следующим образом. Приложение вначале определяет, возможна ли отгрузка всех заказанных товаров. Если да, то приложение оформляет заказ, соответственно уменьшая в таблице INVENTORY значение столбца QuantityOnHand и удаляя из таблиц ORDER JvlASTER и LINE_ITEM записи, относящиеся к этому заказу. Пока все хорошо. Ваше приложение должно обрабатывать заказы одним из двух способов.

- Первый способ состоит в том, чтобы в таблице INVENTORY обрабатывалась запись, которая соответствует каждой записи таблицы LINEJTEM. Если значение в QuantityOnHand является достаточно большим, то приложение его уменьшит. Но если это значение меньше требуемого, выполняется откат транзакции. Это делается для того, чтобы можно было восстановить все изменения, уже внесенные в таблицу INVENTORY в результате обработки предыдущих строк таблицы LINEJTEM этого заказа.
- Второй способ состоит в том, что проверяется каждая запись таблицы INVENTORY, соответствующая какой-либо записи заказа, находящейся в таблице LINEJTEM. Если значения во всех этих записях таблицы INVENTORY достаточно большие, тогда выполняется их уменьшение.

Если заказ выполним, большей эффективностью обладает первый способ, если же нет – второй. Таким образом, если большая часть заказов выполнима, необходимо использовать первый способ. В противном случае больше подойдет второй. Предположим, что это приложение установлено и запущено в многопользовательской системе, в которой нет достаточного управления одновременным доступом. Сразу же возникают следующие проблемы.

- Пользователь 1 запускает обработку заказа с помощью первого способа. На складе находится 10 единиц товара 1, и все это количество требуется для выполнения заказа. После выполнения заказа количество товара 1 становится равным нулю. Вот тут-то и начинается самое интересное. Пользователь 2 запускает обработку небольшого заказа на одну единицу товара 1 и обнаруживает, что заказ оформить нельзя, так как на складе нет нужного количества этого товара. А так как заказ оформить нельзя, выполняется откат. В это время пользователь 1 еще пытается заказать пять единиц товара 37, но на складе их всего четыре. Поэтому и для заказа пользователя 1 также выполняется откат – этот заказ нельзя полностью оформить. И таблица INVENTORY возвращается в состояние, в котором она была перед тем, как эти пользователи начали работать. Получается, что не оформлен ни один из заказов, хотя заказ пользователя 2 вполне можно было выполнить.
- Второй способ не лучше, хотя и по другим причинам. Пользователь 1 может проверить все заказываемые товары и решить, что все они имеются. Но если в дело вмешается пользователь 2 и обработку заказа на один из этих товаров запустит до того, как пользователь 1 выполнит операцию уменьшения, то транзакция пользователя 1 может окончиться неудачно.

## Последовательное выполнение исключает нежелательные взаимодействие

Конфликт транзакций не происходит, если они выполняются **последовательно**. Главное – побыстрее занять очередь. Если невыполнимая транзакция пользователя 1 заканчивается перед началом транзакции пользователя 2, то функция ROLLBACK (откат) возвращает все товары, заказанные пользователем 1, делая их доступными во время транзакции пользователя 2. Если бы во втором примере транзакции выполнялись последовательно, то у пользователя 2 не было бы возможности

изменить количество единиц любого товара, пока не закончится транзакция пользователя 1. Только после окончания транзакции пользователя 1 пользователь 2 сможет увидеть, сколько есть в наличии единиц требуемого товара.

Если транзакции выполняются последовательно, одна за другой, они не смогут друг с другом взаимодействовать и нежелательные последствия таких взаимодействий исключаются, Если результат одновременных транзакций будет таким же, как и при последовательном выполнении, то эти транзакции называются упорядочиваемыми (serializable).

***Внимание:***
*Последовательное выполнение транзакций не является панацеей. Приходится искать компромиссы между производительностью и степенью защиты. Чем больше транзакции изолированы друг от друга, тем больше времени требуется на выполнение какой-либо функции; в киберпространстве, как и в реальной жизни, на ожидание в очереди требуется время. Старайтесь найти такие компромиссы, чтобы настройки вашей системы создавали достаточную защиту, но не большую, чем та, которая вам нужна. Слишком жесткий контроль за одновременным доступом может свести производительность системы на нет.*

## Уменьшение уязвимости данных

Чтобы уменьшить шансы потери данных из-за несчастного случая или непредвиденного взаимодействия, можно принимать меры предосторожности на нескольких уровнях. Можно так настроить СУБД, чтобы она без вас принимала некоторые из этих мер. Вы даже иногда не будете знать о них. Кроме того, администратор базы данных может по своему усмотрению обеспечить и другие меры предосторожности. О них вы также можете быть либо осведомлены, либо нет. И наконец, как разработчик вы можете сами принять определенные меры предосторожности при написании кода. Можно избавить себя от большей части неприятностей, если выработать привычку автоматически придерживаться следующих простых принципов и всегда реализовывать их в своем коде или во время работы с базой данных.

- Использовать транзакции SQL.
- Обеспечить такой уровень изоляции, чтобы соблюдалось равновесие между производительностью и защитой.
- Знать, когда и как запускать транзакции на выполнение, блокировать объекты базы данных и выполнять резервное копирование.

Теперь поговорим об этих принципах подробно.

## Использование транзакций SQL

Одним из главных инструментов SQL, предназначенных для сохранения целостности баз данных, является транзакция. **Транзакция SQL** состоит из любых операторов SQL, которые могут воздействовать на базу данных. Транзакция SQL завершается одним из двух операторов: COMMIT (завершение) или ROLLBACK (откат).

- Если транзакция заканчивается оператором COMMIT, то действие всех ее операторов выполняется в виде одной "пулеметной очереди".
- Если транзакция заканчивается оператором ROLLBACK, то выполняется **откат**, т.е. отмена действия всех ее операторов, а база данных возвращается в то состояние, в котором она находилась перед началом транзакции.

***Помни:***
*Под **приложением** мы сейчас будем подразумевать программу, выполняющуюся на COBOL, C или*

*другом языке программирования, или последовательность команд, вводимых с терминала во время одного сеанса.*

Приложение может состоять из последовательности транзакций SQL. Первая из них начинается как раз в начале приложения, последняя же транзакция заканчивается в его конце. Каждый выполняемый приложением оператор COMMIT или ROLLBACK завершает одну транзакцию SQL и начинает следующую. Например, приложение с тремя транзакциями должно иметь такой общий вид:

## Начало приложения

```
Различные операторы SQL (SQL-транзакция-1)
COMMIT или ROLLBACK
Различные операторы SQL (SQL-транзакция-2)
COMMIT или ROLLBACK
Различные операторы SQL (SQL-транзакция-3)
Конец приложения
```

***Помни:***
*Термин "транзакция SQL" (или "SQL-транзакция") применяют из-за того, что приложение может использовать другие возможности ограничения одновременного доступа (например, связанные с сетевым доступом) и выполнять другие виды транзакций. Далее в этой книге под "транзакцией" подразумевается именно "транзакция SQL".*

Обычная транзакция SQL может выполняться в одном из двух режимов: READ-WRITE (чтение-запись) или READ-ONLY (только чтение). Для нее можно задать один из следующих уровней изоляции: SERIALIZABLE (последовательное выполнение), REPEATABLE READ (повторяющееся чтение), READ COMMITTED (чтение подтвержденных данных) или READ UNCOMMITTED (чтение неподтвержденных данных). (Характеристики транзакций приводятся ниже, в разделе "Уровни изоляции".) Характеристиками по умолчанию являются READ-WRITE и SERIALIZABLE. Если нужно использовать любые другие характеристики, то их следует указать с помощью оператора SET TRANSACTION (задать транзакцию), например, такого:

```
SET TRANSACTION READ ONLY;
```

…или:

```
SET TRANSACTION READ ONLY REPEATABLE READ;
```

…или:

```
SET TRANSACTION READ COMMITTED;
```

В одном приложении может иметься множество операторов SET TRANSACTION, но в каждой транзакции можно указывать только один из них – и он обязательно должен быть первым оператором SQL в транзакции. Если нужно использовать оператор SET TRANSACTION, то его надо выполнять или в начале приложения, или после оператора COMMIT либо ROLLBACK. Этот оператор следует обязательно выполнять в начале каждой транзакции, для которой требуются установки, не совпадающие с предусмотренными по умолчанию. Дело в том, что после оператора COMMIT или ROLLBACK каждая новая транзакция автоматически получает характеристики по умолчанию.

***Технические подробности:***
*В операторе SET TRANSACTION можно также задать значение параметра DIAGNOSTIC SIZE (размер диагностики), определяющего количество ошибочных условий, информацию о которых должна сохранять реализация. Такое ограничение необходимо, потому что при выполнении*

*оператора SQL может произойти сразу много ошибок. Значение по умолчанию этого параметра определяется реализацией, и, как правило, лучше его не трогать.*

## Транзакция по умолчанию

Характеристики транзакции SQL, задаваемые по умолчанию, обычно подходят для большинства пользователей. Впрочем, если необходимо, то с помощью оператора SET TRANSACTION, как уже говорилось в предыдущем разделе, для транзакции можно задать и другие значения параметров. В оставшейся части главы оператору SET TRANSACTION будет уделено много внимания.

Транзакция по умолчанию создается исходя из двух неявных допущений:

- база данных будет со временем изменяться;
- всегда лучше себя обезопасить, чем потом жалеть.

Эта транзакция задает режим READ-WRITE, и он, как можно ожидать, разрешает отправлять на выполнение операторы, изменяющие базу данных. Транзакция по умолчанию имеет уровень изоляции SERIALIZABLE, который является максимально безопасным. Значение параметра DIAGNOSTIC SIZE зависит от используемой реализации и приведено в ее документации.

## Уровни изоляции

Конечно, транзакция должна быть полностью изолирована от других транзакций, выполняющихся одновременно с вашей. Однако в многопользовательской системе из реального мира полная изоляция не всегда осуществима. Она может обернуться слишком высокой потерей производительности. И тут встает вопрос о компромиссе: **какой уровень изоляции нужен вам на самом деле и какое количество производительности вы за него согласны отдать?**

## "Черновое" чтение

Самый слабый уровень изоляции называется READ UNCOMMITED и позволяет выполнять так называемое "**черновое**" **чтение**. При "черновом" чтении изменение, внесенное первым пользователем, может быть прочитано вторым пользователем еще до того, как первый пользователь подтвердит это изменение с помощью оператора COMMIT. Проблема возникает, если первый пользователь прерывает транзакцию и делает для нее откат. Все последующие действия второго пользователя выполняются теперь на основе неправильного значения. Приведем в качестве примера следующую ситуацию. Пусть имеется приложение, работающее с наличными товарами. Один пользователь уменьшает их количество, а второй читает новое, меньшее значение. Первый пользователь делает откат своей транзакции (восстанавливает первоначальное количество), но второй, думая, что товара осталось мало, заказывает его у поставщика, в результате на складе образуется товарный избыток. И это еще не худший случай.

***Внимание:***
*Если вам нужны точные результаты, уровнем изоляции READ UNCOMMITED лучше не пользоваться.*

Уровень READ UNCOMMITED можно использовать тогда, когда нужно статистически обрабатывать такие малоизменяющиеся данные:

- максимальная задержка в оформлении заказов;
- средний возраст продавцов, не выполняющих норму;
- средний возраст новых сотрудников.

Во многих подобных случаях приблизительной информации вполне достаточно; дополнительное снижение производительности, нужное для получения более точного результата, не оправдано.

## Проблемы неповторяющегося чтения

Следующим, более высоким уровнем изоляции является READ COMMITED: изменение, производимое другой транзакцией, невидимо для вашей транзакции до тех пор, пока другой пользователь не завершит ее с помощью оператора COMMIT. Этот уровень обеспечивает лучший результат, чем READ UNCOMMITED, но он все-таки подвержен **неповторяющемуся чтению** – новой серьезной неприятности.

Для пояснения рассмотрим классический пример с наличными товарами. Пользователь 1 отправляет запрос в базу данных, чтобы узнать количество определенного товара, имеющееся на складе. Это количество равно десяти. Почти в то же самое время пользователь 2 начинает, а затем с помощью оператора COMMIT завершает транзакцию, которая записывает заказ на десять единиц того же товара, уменьшая, таким образом, его запасы до нуля. И тут пользователь 1, думая, что в наличии имеется десять единиц товара, пытается оформить заказ на пять единиц. Но и такого количества уже нет. Пользователь 2, по существу, опустошил склад. Первоначальная операция чтения, выполненная пользователем 1 по имеющемуся количеству, является неповторяющейся. Это количество изменили прямо под носом пользователя 1, и все предположения, сделанные на основе полученных им данных, являются неправильными.

## Риск фиктивного чтения

Уровень изоляции REPEATABLE READ дает гарантию, что такой неприятности, как неповторяющееся чтение, уже не будет. Однако на этом уровне все равно часто происходит **фиктивное чтение** – неприятность, возникающая тогда, когда данные, читаемые пользователем, меняются в результате другой транзакции, **причем меняются как раз во время чтения.**

Предположим, например, что пользователь 1 отправляет на выполнение команду, условие поиска которой (предложение WHERE или HAVING) выбирает какое-либо множество строк. И сразу же после этого пользователь 2 выполняет, а затем с помощью оператора COMMIT завершает операцию, в результате которой данные, хранящиеся в одной из этих строк, становятся другими. Вначале эти данные удовлетворяли условию поиска, заданному пользователем 1, а теперь – нет. Или, возможно, некоторые строки, которые вначале этому условию не соответствовали, теперь вполне для него подходят. А пользователь 1, транзакция которого еще не завершена, и понятия не имеет об этих изменениях; само же приложение ведет себя так, как будто ничего не произошло. И вот несчастный пользователь 1 оправляет на выполнение еще один оператор SQL. В нем условия поиска те же самые, что и в первоначальном операторе, поэтому пользователь надеется, что получит те же строки, что и перед этим. Но вторая операция выполняется уже не с теми строками, что первая. В результате фиктивного чтения надежная информация оказалась негодной.

## Тише едешь – дальше будешь

Уровню изоляции SERIALIZABLE не свойственны неприятности, характерные для остальных трех уровней. Одновременные транзакции с уровнем SERIALIZABLE будут выполняться не параллельно, а последовательно. Если вы задали этот уровень изоляции, то только отказы оборудования и программ могут привести к невыполнению транзакции, и зная, что ваша система работает корректно, можно не беспокоиться о правильности результатов операций с базой данных.

Конечно, наивысшая надежность имеет свою цену в виде уменьшения производительности, так что во всем нужно знать меру. В табл. 14.1 приведены четыре уровня изоляции и решаемые ими проблемы.

**Таблица 14.1**. Уровни изоляции и решаемые ими проблемы.

| Уровень изоляции | Решаемые проблемы |
|---|---|
| READ UNCOMMITED | Нет |
| READ COMMITED | "Черновое" чтение |
| REPEATABLE READ | "Черновое" чтение<br>Неповторяющееся чтение |
| SERIALIZABLE | "Черновое" чтение<br>Неповторяющееся чтение<br>Фиктивное чтение |

## Неявный оператор начала транзакции

Некоторые реализации требуют, чтобы о начале транзакции сообщалось явным образом с помощью специального оператора, такого как BEGIN (начало) или BEGIN TRAN (начало транзакции). Стандарт SQL:2003 этого не требует. Если еще никакая транзакция не начата и на выполнение отправляется команда SQL, то в системе, совместимой с SQL:20039, создается транзакция по умолчанию. Например, операторы CREATE TABLE, SELECT и UPDATE выполняются только в транзакции. Достаточно запустить один из них на выполнение, и начнется транзакция по умолчанию.

## Оператор SET TRANSACTION

Время от времени для транзакции приходится использовать характеристики, отличные от устанавливаемых по умолчанию. Эти нестандартные характеристики можно задавать с помощью оператора SET TRANSACTION. Этот оператор должен быть первым оператором транзакции. Оператор SET TRANSACTION позволяет задавать режим, уровень изоляции и размер диагностики.

Чтобы изменить, например, все три характеристики, можно отправить на выполнение такую команду:

```
SET TRANSACTION
READ ONLY,
ISOLATION LEVEL READ UNCOMMITED,
DIAGNOSTICS SIZE 4;
```

Операторы данной транзакции не смогут изменить базу данных (режим READ ONLY). Кроме того, определен самый низкий и, следовательно, наиболее опасный уровень изоляции (READ UNCOMMITED). Для области диагностики выбран размер 4. Как видно, параметры подобраны таким образом, чтобы транзакция использовала поменьше системных ресурсов.

Теперь сравните предыдущую команду с этой:

```
SET TRANSACTION
READ WRITE,
ISOLATION LEVEL SERIALIZABLE,
DIAGNOSTICS SIZE 8;
```

Эти значения позволяют транзакции изменить базу данных, задают наивысший уровень изоляции и назначают большую область диагностики. Эта транзакция предъявляет гораздо более высокие требования к системным ресурсам. В зависимости от реализации указанные значения могут оказаться значениями по умолчанию. Естественно, в операторе SET TRANSACTION можно использовать и другие значения уровня изоляции и размера диагностики.

***Совет:***
*Не задавайте слишком высокий уровень изоляции. Может показаться, что для надежности всегда лучше выбирать значение SERILIZABLE. В зависимости от конкретной ситуации это может оказаться неплохой идеей. С другой стороны, не всем транзакциям требуется такой высокий уровень изоляции, очень снижающий общую производительность системы. Если во время транзакции не требуется модифицировать базу данных, смело задавайте режим READ ONLY. В общем, лучше обойтись без фанатизма*.

## Оператор COMMIT

Хотя в SQL:2003 не предусмотрен явный оператор начала транзакции, но зато есть два опера-юра ее завершения: COMMIT и ROLLBACK. Первый из них используйте тогда, когда вы уже дошли до конца транзакции и собираетесь подтвердить изменения, внесенные в базу данных. В операторе COMMIT может находиться необязательное ключевое слово WORK (COMMIT WORK означает "завершить работу") – Если при выполнении оператора COMMIT произойдет ошибка или системный сбой, возможно, потребуется выполнить откат транзакции и повторить ее снова.

## Оператор ROLLBACK

В конце транзакции иногда требуется отменить изменения, внесенные во время этой транзакции, т.е. восстановить базу данных в том состоянии, в каком она была перед самым началом транзакции. Для этого можно воспользоваться оператором ROLLBACK, который является отказоустойчивым. Даже если во время его выполнения в системе произойдет аварийный сбой, то после перезагрузки оператор ROLLBACK можно запустить снова. И он должен восстановить базу данных в состоянии, в каком она была перед началом транзакции.

## Блокирование объектов базы данных

Уровень изоляции, который установлен по умолчанию или с помощью оператора SET TRANSACTION, указывает СУБД, сколько усилий ей нужно прилагать, чтобы ваша работа не взаимодействовала с работой других пользователей. Главная защита со стороны СУБД от нежелательных транзакций – это блокировка используемых вами объектов базы данных. Ниже приведено несколько примеров такой блокировки.

- Заблокированной может быть табличная строка, с которой вы работаете в данный момент времени. Пока вы ее используете, никто другой не имеет к ней доступ.
- Заблокированной может быть вся таблица, если вы выполняете операцию, которая повлияет на таблицу в целом.
- Блокируется запись, а чтение табличных данных допускается. Иногда же блокируется и чтение, и запись.

В каждой СУБД управление блокировкой реализовано по-своему. И хотя в некоторых реализациях "броня" толще, чем в других, но все равно большинство из нынешних систем в состоянии защитить данные от негативных последствий одновременного доступа.

## Резервное копирование

Резервное копирование – комплекс мер, регулярно выполняемый администратором базы данных. Резервное копирование всех элементов системы должно проводиться через определенные интервалы, которые зависят от того, насколько часто обновляются эти элементы. Если ваша база данных обновляется ежедневно, то ее резервное копирование также следует проводить ежедневно. Приложения, формы и отчеты также могут меняться, хотя и не так часто. Администратор должен успевать создавать и их резервные копии тоже.

***Совет:***
*Храните несколько версий резервных копий. Иногда ущерб, причиненный базе данных, становится очевидным лишь через некоторое время. Чтобы вернуться к последней работоспособной версии, вам, возможно, придется вернуться на несколько версий назад.*

Есть много разных способов резервного копирования.

- Создавать резервные таблицы и копировать в них данные с помощью SQL.
- Использовать определяемый реализацией механизм, который создает резервную копию всей базы или ее частей. Обычно этот механизм намного удобнее и эффективнее, чем использование SQL.
- В вашей операционной системе может быть механизм, предназначенный для резервного копирования всего, чего угодно, в том числе баз данных, программ, документов, электронных таблиц и компьютерных игр. В этом случае вам придется лишь проверить, проводится ли резервное копирование с достаточной частотой.

***Технические подробности:***
*Возможно, вы слышали от проектировщиков баз данных, что базам данных нужна ACID (означает "кислота"). Нет, конечно, они не собираются одурманить свои творения психоделией из 1960-годов или растворить в пузырящемся месиве хранящиеся в базах данные. ACID– это аббревиатура, образованная от слов* ***Atomicity*** *(атомарность),* ***Consistency*** *(согласованность),* ***Isolation*** *(изоляция) и* ***Durability*** *(долговечность). Эти четыре характеристики необходимы для защиты базы данных от искажения данных.*

- **Атомарность**. Транзакции в базах данных должны быть атомарными в классическом смысле этого слова, т.е. вся транзакция считается неделимой единицей. Или она выполняется целиком (завершается оператором COMMIT), или база данных восстанавливается в том состоянии, в котором она была бы, если бы транзакция не выполнялась (происходит откат с помощью оператора ROLLBACK).
- **Согласованность**. Как это ни странно, но само значение слова "согласованность" не является согласованным; в разных приложениях оно разное. Например, при переводе фондов в банковском приложении с одного счета на другой вы хотите, чтобы общая сумма денег на обоих счетах была в конце транзакции такой же, как и в ее начале. В другом приложении критерий согласованности может быть другим.
- **Изоляция**. Транзакции в базах данных должны быть полностью изолированы от других транзакций, выполняемых в это же время. Если транзакции можно выполнять последовательно, полная изоляция является достижимой. Если же система должна обрабатывать транзакции с максимальной скоростью, то увеличение производительности иногда достигается за счет более низкого уровня изоляции.
- **Долговечность**. Необходимо, чтобы после завершения или отката транзакции базу данных можно было считать находящейся в рабочем состоянии, т.е. заполненной неповрежденными, надежными и обновленными данными. Даже если с системой во время транзакции произойдет аварийный сбой, долговечная СУБД должна обеспечивать возможность восстановления базы данных в рабочем состоянии.

## Точки отката и субтранзакции

В идеале транзакции должны быть атомарными – такими же неделимыми, как представлялись атомы древним грекам. Однако в действительности даже атомы неделимыми не являются. С появлением реализации SQL: 1999 транзакции в базах данных также перестали быть атомарными. Транзакция теперь может состоять из множества субтранзакций. Субтранзакции отделяются друг от друга

точками отката, задаваемыми с помощью оператора SAVEPOINT. Он может использоваться в сочетании с оператором ROLLBACK. До появления точек отката оператор ROLLBACK применялся только для отмены всей транзакции, теперь его можно использовать для отката транзакции до имеющейся в ней точки отката. Вы можете спросить: а для чего это нужно?

В основном оператор ROLLBACK применяется для восстановления исходных данных после того, как транзакция была прервана из-за ошибки. Понятно, что если во время транзакции произошла ошибка, то бессмысленно делать откат только к ближайшей точке отката. Ведь для того, чтобы вернуть базу данных в состояние, в котором она была перед началом транзакции, нужен откат **всей** транзакции. Но это еще не вся правда. Возникают ситуации, в которых нужен откат именно части транзакции.

Скажем, со своими данными вы проводите сложную последовательность операций. Где-то во время этого процесса получаются результаты, которые приводят вас к заключению, что вами выбран неверный путь. Если установить точку отката в соответствующем месте этой последовательности, можно выполнить откат к ней и попробовать другой вариант. Если нужно изменить действие лишь части кода, а остальной код является безупречным, то лучше поступить так, чем прекращать всю текущую транзакцию и запускать новую ради небольшого изменения.

Чтобы поместить в код SQL точку отката, используйте следующий синтаксис:

```
SAVEPOINT имя_точки_сохранения;
```

Откат транзакции к этой точке можно выполнить с помощью следующего кода:

```
ROLLBACK TO SAVEPOINT имя_точки_сохранения;
```

Некоторые реализации SQL не поддерживают оператор SAVEPOINT. Если в вашей реализации его нет, то использовать этот оператор вы не сможете.

## Ограничения в транзакциях

Проверка достоверности данных заключается не только в проверке правильности типа данных. Кроме этого, возможно, потребуется следить, чтобы в некоторых столбцах не было неопределенных значений, а в других – значения не выходили за границы определенного диапазона. Об этих **ограничениях** см. в главе 5.

Говорить о связи ограничений с транзакциями имеет смысл потому, что первые влияют на работу последних. Предположим, например, что нужно добавить данные в таблицу, в которой имеется столбец с ограничением NOT NULL. Как правило, в начале создают пустую строку и затем заполняют ее значениями. Однако ограничение NOT NULL не позволит воспользоваться данным способом, поскольку SQL не даст ввести строку с неопределенным значением, находящимся в столбце с ограничением NOT NULL, даже если данные в этот столбец предполагается ввести еще до конца транзакции. Для решения этой проблемы в SQL:2003 существует возможность определять ограничения как DEFERRABLE (задерживаемые) или NOT DEFERRABLE (незадерживаемые).

Ограничения, определенные как NOT DEFERRABLE, применяются немедленно. А те, что определены как DEFERRABLE, первоначально могут быть заданы как DEFERRED (задержанные) или IMMEDIATE (немедленные). Если ограничение типа DEFERRABLE задано как IMMEDIATE, то оно действует так, как и ограничение типа NOT DEFERRABLE, – немедленно. Если же ограничение типа DEFERRABLE задано как DEFERRED, то его действие может быть отсрочено.

Чтобы добавлять пустые записи или выполнять другие операции, которые могут нарушить ограничения типа DEFERRABLE, можно использовать следующий оператор:

```
SET CONSTRAINTS ALL DEFERRED;
```

Он определяет все ограничения типа DEFERRABLE как DEFERRED. На ограничения типа NOT DEFERRABLE этот оператор не действует. После того как выполнены все операции, которые могут нарушить ваши ограничения, и таблица достигла того состояния, когда их нарушать уже нельзя, тогда эти ограничения можно применить заново. Соответствующий оператор выглядит следующим образом:

```
SET CONSTRAINTS ALL IMMEDIATE;
```

Если вы сделали ошибку и данные не соответствуют каким-либо ограничениям, вы сразу же это увидите после выполнения данного оператора.

Если ограничения, ранее заданные как DEFERRED, явно не задаются вами как IMMEDIATE, то, когда вы попытаетесь завершить свою транзакцию с помощью оператора COMMIT, SQL активизирует все задержанные ограничения. Если к этому моменту не все ограничения выполняются, транзакция будет отменена и SQL выдаст сообщение об ошибке.

Ограничения в SQL защищают от ввода неправильных данных (или, что так же важно, от недопустимого **отсутствия** данных), причем у вас есть возможность на время действия транзакции временно отменить имеющиеся ограничения.

Чтобы увидеть, насколько важна отсрочка применения ограничений, проанализируйте пример с платежными ведомостями.

Предположим, что в таблице EMPLOYEE (сотрудник) имеются столбцы EmpNo (номер сотрудника), EmpName (фамилия сотрудника), DeptNo (номер отдела) и Salary (оклад). DeptNo является внешним ключом, который ссылается на таблицу DEPT (отдел). Предположим также, что в таблице DEPT имеются столбцы DeptNo и DeptName (название отдела), причем DeptNo – это первичный ключ.

Представим, что, кроме этого, вы хотите иметь такую же таблицу, как и DEPT, но в которой еще есть столбец Payroll (платежная ведомость), и в нем для каждого отдела имеется сумма значений Salary сотрудников этого отдела.

Эквивалент этой таблицы можно создать с помощью следующего представления:

```
CREATE VIEW DEPT2 AS
SELECT D.*, SUM(E.Salary) AS Payroll
FROM DEPT D, EMPLOYEE E
WHERE D.DeptNo = E.DeptNo
GROUP BY D.DeptNo;
```

Точно такое же представление можно определить еще и другим способом:

```
CREATE VIEW DEPT3 AS
SELECT D.*,
(SELECT SUM(E.Salary)
FROM EMPLOYEE E
WHERE D.DeptNo = E.DeptNo) AS Payroll
FROM DEPT D;
```

Но предположим, что для большей эффективности вы не собираетесь вычислять значение SUM каждый раз, когда ссылаетесь на столбец DEPT.Payroll. Вместо этого вы хотите, чтобы в таблице

DEPT в действительности находился столбец Payroll. Значения в этом столбце вы будете обновлять каждый раз, когда будете вносить изменения в столбце Salary.

И, чтобы обеспечить правильность значений в столбце Salary, в определение таблицы можно вставить ограничение:

```
CREATE TABLE DEPT
(DeptNo CHAR(5),
DeptName CHAR(20),
Payroll DECIMAL(15.2),
CHECK (Payroll = (SELECT SUM(Salary)
FROM EMPLOYEE E WHERE E.DeptNo = DEPT.DeptNo)));
```

Теперь предположим, что вам надо увеличить на 100 значение Salary сотрудника под номером 123. Это можно сделать с помощью следующего обновления:

```
UPDATE EMPLOYEE
SET Salary = Salary + 100
WHERE EmpNo = '123';
```

Кроме того, следует обновить таблицу DEPT:

```
UPDATE DEPT D
SET Payroll = Payroll + 100
WHERE D.DeptNo = (SELECT E.DeptNo
FROM EMPLOYEE E
WHERE E.EmpNo 423');
```

*(Подзапрос используется для того, чтобы получить ссылку на значение DeptNo сотрудника с идентификационным номером 123.)*

Но здесь имеется трудность: после каждого оператора ограничения проверяются. Теоретически должны проверяться все ограничения. Но на практике реализациями проверяются только те из них, которые относятся к значениям, измененным в ходе работы оператора.

После первого из двух последних операторов UPDATE реализация проверяет все ограничения, которые ссылаются на измененные им значения. В число этих ограничений входит то, которое определено в таблице DEPT, потому что оно относится к столбцу Salary таблицы EMPLOYEE, а значения в этом столбце изменяются оператором UPDATE. После выполнения первого оператора UPDATE это ограничение оказалось нарушенным. Допустим, что перед выполнением этого оператора база данных находится в полном порядке и каждое значение Payroll в таблице DEPT равно сумме значений Salary из соответствующих строк таблицы EMPLOYEE. Тогда, как только первый оператор UPDATE увеличит значение Salary, это равенство выполняться не будет. Такая ситуация исправляется вторым оператором UPDATE, после выполнения которого значения базы данных соответствуют имеющимся ограничениям. Но в промежутке между этими обновлениями ограничения не выполняются.

Оператор SET CONSTRAINTS DEFERRED дает возможность временно отключить все или только указанные вами ограничения. Действие этих ограничений будет задержано до тех пор, пока выполнится или оператор SET CONSTRAINTS IMMEDIATE, или один из двух операторов: COMMIT и ROLLBACK. Следовательно, в нашем случае перед оператором UPDATE и после него необходимо поместить операторы SET CONSTRAINTS:

```
SET CONSTRAINTS DEFERRED;
UPDATE EMPLOYEE
SET Salary = Salary + 100
```

```
WHERE EmpNo = '123';
UPDATE DEPT D
SET Payroll = Payroll + 100
WHERE D.DeptNo = (SELECT E.DeptNo
FROM EMPLOYEE E
WHERE E.EmpNo = '123');
SET CONSTRAINTS IMMEDIATE;
```

Эта процедура откладывает действие всех ограничений. Например, при вставке в DEPT новых строк первичные ключи проверяться не будут; т.е. вы удалили и ту защиту, которая, возможно, вам нужна. Поэтому следует явно указывать ограничения, которые надо задержать. Для этого при создании ограничений им следует давать имена:

```
CREATE TABLE DEPT
(DeptNo CHAR(5),
DeptName CHAR(20),
Payroll DECIMAL(15.2),
CONSTRAINT PayEqSumsal
CHECK (Payroll = SELECT SUM(Salary)
FROM EMPLOYEE E
WHERE E.DeptNo = DEPT.DeptNo));
```

На ограничения с именами можно ссылаться индивидуально:

```
SET CONSTRAINTS PayEqSumsal DEFERRED;
UPDATE EMPLOYEE
SET Salary = Salary + 100
WHERE EmpNo = '12 3';
UPDATE DEPT D
SET Payroll = Payroll + 100
WHERE D.DeptNo = (SELECT E.DeptNo
FROM EMPLOYEE E
WHERE E.EmpNo = 423');
SET CONSTRAINTS PayEqSumsal IMMEDIATE;
```

Если для ограничения в операторе CREATE не указано имя, то SQL создает это имя неявно. Оно находится в специальных таблицах – таблицах каталога. Но все-таки лучше явно задавать имена ограничений.

Теперь предположим, что во втором операторе UPDATE в качестве значения возрастания вы по ошибке указали 1000. Это значение в операторе UPDATE является разрешенным, потому что имеющееся ограничение было отсрочено. Но при выполнении оператора SET CONSTRAINTS…IMMEDIATE будут проверяться указанные в нем ограничения. Если они не соблюдаются, то будет сгенерировано исключение. Но если вместо оператора SET CONSTRAINTS…IMMEDIATE выполняется оператор COMMIT, а ограничения не соблюдаются, то вместо оператора COMMIT выполняется оператор ROLLBACK.

Подведем итог. Временно отменять действие ограничений можно только внутри транзакции. Как только транзакция завершается, ограничения сразу же вступают в силу. Если использовать эту возможность правильно, то транзакция не создаст никаких данных, нарушающих какие-либо ограничения.

# Использование SQL в приложениях

В этой главе…

- SQL в приложении
- Совместное использование SQL с процедурными языками
- Как избежать несовместимости
- Код SQL, встроенный в процедурный код
- Вызов модулей SQL из процедурного кода
- Вызов SQL из RAD-инструмента

---

В предыдущих главах мы в основном рассматривали SQL-команды в отдельности, т.е. формулировалась задача обработки данных, и под нее создавался SQL-запрос. Такой подход, предполагающий интерактивное использование SQL, прекрасно подходит для изучения его возможностей, но в повседневной практике обычно SQL применяют по-другому.

Хотя синтаксис языка SQL похож на синтаксис английского языка, изучить SQL все равно нелегко. Подавляющее большинство сегодняшних пользователей им не владеют в достаточной мере. И можно предположить, что даже если эта книга и завоюет широкую популярность, то все равно подавляющее большинство компьютерных пользователей так никогда и не будут свободно владеть SQL. Если обычному пользователю поставить задачу, связанную с базой данных, он и не подумает садиться к терминалу и вводить оператор SELECT. Системные аналитики и разработчики приложений, свободно владеющие SQL, также не занимаются вводом разовых (ad hoc) запросов с консоли. Эти специалисты разрабатывают приложения, которые сами создают запросы.

Если вы собираетесь много раз выполнять одну и ту же команду SQL, необязательно каждый раз вводить ее заново. Напишите приложение и затем запускайте его столько раз, сколько нужно. Код SQL может быть частью приложения, но в этом случае он работает немного иначе, чем в интерактивном режиме.

## SQL в приложении

В главе 2 SQL был назван неполным языком программирования. Для использования в приложении язык SQL необходимо комбинировать с процедурным языком, таким как Pascal, FORTRAN, Visual Basic, C, C++, Java или COBOL. У SQL есть как сильные, так и слабые стороны. У процедурных языков, имеющих другую структуру, также есть сильные и слабые стороны, но не такие, как у SQL. К счастью, сильные стороны SQL обычно компенсируют слабость процедурных языков, а сильные стороны процедурных языков проявляются как раз там, где SQL оказывается не на высоте. Если совместно использовать процедурные языки с SQL, можно создать мощные приложения с широким набором возможностей. Недавно появились объектно-ориентированные RAD-инструменты, которые позволяют встраивать код SQL в приложения, создаваемые из объектов, а не с помощью написания процедурного кода. В качестве примера таких инструментов можно привести Delphi и C++ Builder.

***Внимание:***
*В предыдущих главах мы часто использовали звездочку ('*') для краткого обозначения всех столбцов в таблице. Если в таблице много столбцов, то звездочка помогает уменьшить ввод кода. Если же SQL применяется в прикладной программе, лучше раз и навсегда от нее отказаться. После того как приложение будет написано, вы или кто-то другой, возможно, добавите в таблицу новые столбцы или удалите старые. Однако в таком случае меняется значение понятия "все столбцы". Ваше приложение, когда оно указывает с помощью звездочки "все столбцы", может получить не те столбцы, на которые рассчитывает.*

Такое изменение в таблице не окажет влияния на имеющиеся программы, пока для исправления ошибки или внесения какого-либо изменения эти программы не будут перекомпилированы. Тогда воздействие символа шаблона ('*') расширится на все имеющиеся столбцы. Это изменение может привести к аварийному завершению работы приложения, не связанному с текущей отладкой, или вызвать другие изменения, которые сделают отладку настоящим кошмаром.

***Совет:***
*Чтобы обезопасить себя, указывайте имена всех столбцов в явном виде, а не полагайтесь на символ звездочки.*

## Сильные и слабые стороны SQL

Одной из сильных сторон языка SQL является получение данных. Если важная информация размещена где-то в однотабличной или многотабличной базе данных, то ее можно отыскать с помощью инструментов SQL. Этот язык не работает с отдельными строками или столбцами, поэтому вам не нужно знать порядок расположения в таблице строк или столбцов. Имеющиеся в SQL средства обработки транзакций гарантируют, что на операции, выполняемые с базой данных, не повлияют любые другие пользователи, которые одновременно с вами могут получить доступ к одним и тем же таблицам.

Однако одной из слабых сторон SQL является его недоразвитый пользовательский интерфейс. В SQL нет средств для форматирования вывода и генерации отчетов. Результат выполнения запроса передается на терминал построчно.

Иногда качество, которое в одном случае является достоинством, может в другом оказаться, наоборот, недостатком. Например, одна из сильных сторон языка SQL состоит в том, что он может работать сразу с целой таблицей. Сколько бы в таблице ни было строк: одна, сто или сто тысяч – нужные вам данные можно извлечь из нее с помощью единственного оператора SELECT. Однако SQL не может легко работать с каждой строкой индивидуально. Для этого придется использовать или курсоры, которые описаны в главе 18, или процедурный базовый язык.

## Сильные и слабые стороны процедурных языков

В противоположность SQL, процедурные языки легко оперируют с отдельными строками, позволяя разработчику приложения точно управлять способом обработки таблицы. Такое управление является очень большим достоинством процедурных языков. Впрочем, имеете" и слабая сторона подобного подхода. Разработчик приложения должен точно знать, каким образом данные хранятся в таблицах базы данных. Имеет значение порядок расположении строк и столбцов, и его приходится учитывать.

Благодаря своей природе процедурные языки являются очень гибкими, и с их помощью можно создавать удобные экраны ввода и просмотра данных. Кроме того, для вывода на печать можно создавать очень сложные отчеты, имеющие любую требуемую структуру.

## Трудности совместного использования SQL с процедурным языком

Учитывая то, как достоинства SQL компенсируют недостатки процедурных языков и наоборот, имеет смысл комбинировать их таким образом, чтобы объединить их достоинства, а не недостатки. Чтобы реализовать такое объединение на практике, приходится преодолевать некоторые трудности.

## Противоположные режимы работы

При комбинировании SQL с процедурными языками немалая трудность состоит в том, что при работе с таблицами язык SQL сразу может обрабатывать целый их набор, в то время как

процедурные языки – только одну строку. Иногда это не так уж и важно. Операции с наборами таблиц можно выполнять отдельно от операций со строками, используя каждый раз нужный инструмент. Однако в некоторых случаях приходится проверять все записи таблицы на соответствие определенным условиям и в зависимости от этого выполнять с ней те или иные действия. Для такого процесса требуются как мощные возможности SQL по извлечению данных, так и возможности условных переходов, имеющиеся у процедурных языков. Эту комбинацию возможностей можно достичь с помощью SQL-кода, встроенного в программу, написанную на обычном процедурном языке.

## Несовместимость типов данных

Другое препятствие к легкой интеграции SQL с любым процедурным языком состоит в том, что типы данных SQL отличаются от типов данных основных языков программирования. Это не должно вас удивлять, ведь даже типы данных одного процедурного языка отличаются от типов данных другого. Общего стандарта типов данных не существует. В ранних версиях SQL, предшествовавших SQL-92, одной из главных проблем была несовместимость данных. Однако в SQL-92 (а также в SQL: 1999 и SQL:2003) с этой проблемой справляется оператор CAST. Как с его помощью преобразовать тип элемента данных, поддерживаемый в процедурном языке, в тип, который используется SQL, см. в главе 8.

## SQL – код в программе, написанной на процедурном языке

Хотя на пути интеграции SQL с процедурными языками может возникать множество трудностей, эта задача выполнима. Во многих случаях эта интеграция – единственный способ решения поставленной задачи за разумное время. О трех методах подобной интеграции – **встроенном SQL**, **модульном языке** и **инструментах RAD** – кратко рассказывается в следующих разделах.

## Встроенный SQL

Самый распространенный метод сочетания SQL с процедурными языками называется **встроенным SQL**. Как понятно из названия, операторы SQL вставляются в нужные места процедурной программы. Естественно, внезапно появившись в программе, написанной, скажем, на языке C, оператор SQL может стать неприятным сюрпризом для компилятора. По этой причине программы, в которых имеется встроенный SQL, перед компиляцией или интерпретацией пропускают через препроцессор. О встроенном SQL-коде препроцессор предупреждается директивой EXEC SQL.

В качестве примера использования встроенного SQL рассмотрим программу, написанную на языке Pro*C фирмы Oracle, являющемся вариантом языка C. Программа получает доступ к таблице с данными сотрудников компании, предлагает пользователю ввести имя сотрудника, а затем выводит зарплату и комиссионные этого сотрудника. Затем она предлагает ввести новые данные по зарплате и комиссионным этого же сотрудника и обновляет этими данными таблицу.

```
EXEC SQL BEGIN DECLARE SECTION;
VARCHAR uid[20];
VARCHAR pwd[20];
VARCHAR ename[10];
FLOAT salary, comm;
SHORT salary_ind, comm_ind;
EXEC SQL END DECLARE SECTION;
main ()
{
int sret; /* Возвращаемый код scanf */
/* Регистрация */
strcpy(uid.arr,"FRED"); /*Копировать имя пользователя */
uid.len=strlen(uid.arr);
```

```
strcpy(pwd.arr,"TOWER"); */ Копировать пароль /*
pwd.len=strlen(pwd.arr);
EXED SQL WHENEVER SQLERROR STOP;
EXED SQL WHENEVER NOT FOUND STOP;
EXEC SQL CONNECT:uid;
printf("Соединение с пользователем: percents \n",uid.arr);
printf("Введите имя пользователя для обновления: ");
scanf("percents",ename.arr);
ename.len=strlen(ename.arr);
EXEC SQL SELECT SALARY,COMM INTO:salary,:cornm
FROM EMPLOY
WHERE ENAME=:ename;
printf("Сотрудник: percents оклад: percent6.2f комиссионные:
percent6.2f \n",
ename.arr, salary, comm);
printf("Введите новый оклад: ");
sret=scanf("percentf",&salary);
salary_ind =0; /* Инициализировать индикатор */
if (sret == EOF!! sret == 0)
salary_ind =-1; /* Установить индикатор отсутствия ввода */
printf("Введите новые комиссионные: ");
sret=scanf("percentf",&comm);
comm_ind =0; /* Инициализировать индикатор */
if (sret == EOF!! sret == 0)
comm_ind=-1; /* Установить индикатор отсутствия ввода */
EXEC SQL UPDATE EMPLOY
SET SALARY=:salary:salary_ind
SET COMM=:comm:comm_ind
WHERE ENAME=:ename;
printf("Данные сотрудника percents обновлены. \n",ename.arr);
EXEC SQL COMMIT WORK;
exit(0);
}
```

Не надо быть экспертом по языку С, чтобы понять суть того, что и каким образом делает программа. Ниже приведена последовательность выполнения операторов.

- SQL объявляет базовые переменные.
- Язык С контролирует процедуру регистрации пользователя.
- SQL активизирует обработку ошибок и соединяется с базой данных.
- Язык С запрашивает у пользователя имя сотрудника, которое помещает в переменную rename.
- Оператор SQL SELECT извлекает данные по зарплате и комиссионным указанного пользователя и сохраняет их в двух переменных базового языка, salary и xomm.
- Далее язык С выводит имя сотрудника, его зарплату и комиссионные, а затем требует ввести новые значения зарплаты и комиссионных. Он также проверяет, введены ли требуемые значения, и если нет, то устанавливает индикатор отсутствия ввода.
- SQL обновляет данные в базе на основе новых значений.
- Язык С отображает сообщение о завершении операции.
- SQL завершает транзакцию, и С завершает выполнение программы.

***Совет:***
*Смешивать команды двух языков так, как это делается здесь, можно благодаря препроцессору. Он отделяет операторы SQL от команд базового языка, помещая эти операторы в отдельную внешнюю процедуру. Каждый оператор SQL заменяется вызовом соответствующей внешней процедуры. Теперь за свою работу может приниматься компилятор этого языка. Способ, с помощью которого операторы SQL будут передаваться базе данных, зависит от реализации. Но вы как разработчик ни о чем таком беспокоиться не должны. Этим займется препроцессор. Вам надо лишь позаботиться о базовых переменных и совместимости типов данных*.

## Объявление базовых переменных

***Помни:***
*Между программой, написанной на базовом языке, и кодом SQL должна передаваться информация. Для этого используются базовые переменные. Чтобы SQL признал эти переменные, их перед использованием необходимо объявить. Объявления находятся в сегменте объявлений, который расположен перед программным сегментом. Сегмент объявлений начинается со следующей директивы:*

```
EXEC SQL BEGIN DECLARE SECTION;
```

А конец этого сегмента отмечается с помощью такого выражения:

```
EXEC SQL END DECLARE SECTION;
```

Перед каждым оператором SQL должна стоять директива SQL EXEC. Конец сегмента SQL может быть отмечен директивой завершения. В языке COBOL такой директивой является '"END-EXEC", а в языке FORTRAN – это конец строки, а в языках Ada, C, Pascal и PL/1 – точка с запятой.

## Преобразование типов данных

В зависимости от типов данных, поддерживаемых базовым языком и языком SQL, вам, воз-рсно, для преобразования некоторых из этих типов данных придется использовать оператор CAST. Можно использовать базовые переменные, которые были объявлены с помощью DECLARE SECTION. А когда в операторах SQL применяются имена этих переменных, то не надо забывать следующее: перед этими именами необходимо ставить двоеточие (**:**). Это как раз и делается в следующем примере с таблицей FOODS (продукты питания) со столбцами FOODNAME (название продукта), CALORIES (калории), PROTEIN (белки), FAT (жиры), CARBOHYDRATE (углеводы):

```
INSERT INTO FOODS
(FOODNAME, CALORIES, PROTEIN, FAT, CARBOHYDRATE)
VALUES
(: foodname, rcalories,.-protein,:fat,:carbo);
```

## Модульный язык

Использовать SQL вместе с процедурным программным языком можно и по-другому – с помощью **модульного языка**. Благодаря ему вы сможете разместить все операторы SQL в отдельном модуле SQL.

***Помни:***
*Модуль SQL– это список команд SQL. Каждая из этих команд вызывается* ***процедурой SQL****, и перед ней стоят имя процедуры, имена и типы ее параметров*.

В каждой процедуре SQL находится не менее одного оператора SQL. В любом месте программы на базовом языке можно явно вызвать эту процедуру. Вызов процедуры SQL происходит так, как если бы она была подпрограммой, написанной на базовом языке.

Таким образом, создать модуль SQL вместе со связанной с ним программой на базовом языке – это, в сущности, просто написать вручную тот код, который получается после обработки препроцессором встроенного SQL-кода.

Встроенный SQL получил большее распространение, чем модульный. И хотя большинство поставщиков предлагают какой-либо вариант модульного языка, но мало кто из них в своей

документации уделяет ему особое внимание. Впрочем, у модульных языков есть несколько преимуществ.

- Так как SQL полностью отделен от процедурного языка, то для написания модулей можно нанять самых лучших программистов, работающих в SQL. И неважно, имеют они какой-либо опыт работы с процедурным языком или нет. На самом деле решение вопроса о том, какой процедурный язык использовать, можно даже отложить до того времени, пока не будут написаны и отлажены модули SQL.
- Программу на базовом языке может написать программист, не знакомый с SQL.
- Важнее всего то, что в процедурном коде не будет фрагментов SQL-кода, поэтому можно будет использовать отладчик, позволяющий сэкономить большое количество времени, которое приходится тратить на разработку. И снова оговоримся: то, что с одной точки зрения может выглядеть как преимущество, с другой может выглядеть как недостаток. Модули SQL отделены от процедурного кода, и если вы попытаетесь понять работу программы, то вам будет труднее это сделать, чем при использовании встроенного SQL.

## Объявления модулей

Синтаксис объявлений в модулях имеет такой вид:

```
MODULE [имя-модуля]
[NAMES ARE имя-набора-символов]
LANGUAGE (ADA|C|COBOL|FORTRAN|MUMPS I PASCAL IPLI|SQL}
[SCHEMA имя – схемы]
[AUTHORIZATION идентификатор-подтверждения-полномочий]
[объявления-временных-таблиц… ]
[объявления-курсоров… ]
[объявления-динамических-курсоров…]
процедуры…
```

Как видно, имя модуля не является обязательным. Но, вероятно, все-таки неплохо модуль как-то назвать, чтобы избежать слишком большой путаницы. Необязательное предложение NAMES ARE (имена) определяет набор символов, используемый для имен. Если этого предложения в объявлении не будет, то используется набор символов, установленный в вашей реализации по умолчанию. Предложение LANGUAGE (язык) сообщает модулю, на каком языке написаны программы, которые должны его вызывать. Компилятору надо обязательно знать, что это за язык. Ведь он собирается преобразовать команды SQL так, чтобы для вызывающей их программы они выглядели как подпрограммы, написанные на том же языке, что и сама эта программа.

Хотя оба предложения, SCHEMA (схема) и AUTHORIZATION (подтверждение полномочий), являются необязательными, непременно надо указать хотя бы одно из них. Можно также указать и оба сразу. Первое из них определяет схему по умолчанию, а второе – идентификатор подтверждения полномочий. Если зы его не укажете, то для предоставления вашему модулю полномочий СУБД будет использовать полномочия текущего сеанса. Если на выполнение операции, которую вызывает ваша процедура, у вас нет прав, то процедура не будет выполняться.

***Совет:***
*Если вашей процедуре нужны временные таблицы, то объявляйте их с помощью специального предложения объявлений временных таблиц. Объявляйте курсоры и динамические курсоры еще до того, как они будут использованы какой-либо процедурой. Объявление курсора после процедуры*

*возможно в случае, если его будет использовать не она, а процедуры, объявленные после. Более подробная информация, относящаяся к курсорам, приведена в главе 18*.

## Процедуры модуля

И наконец, после всех этих объявлений в модуле находятся его функциональные части – процедуры. У процедуры в модульном SQL есть имя, объявления параметров и выполняющиеся операторы SQL. Программа, написанная на процедурном языке, вызывает процедуру по ее имени и передает ей значения через объявленные параметры. Синтаксис процедуры следующий:

```
PROCEDURE имя-процедуры
(объявление-параметра [, объявление-параметра]…)
оператор SQL;
[операторы SQL];
```

Объявление параметра должно иметь такой вид:

```
имя-параметра тип-данных
```

…или такой:

```
SQLSTATE
```

Объявляемые вами параметры могут быть входными, выходными или теми и другими одновременно. SQLSTATE является параметром состояния, посредством которого выводится информация об ошибках. Чтобы подробнее разобраться с параметрами, обратитесь к главе 20.

## Объектно-ориентированные RAD-инструменты

Используя современные RAD-инструменты, можно создавать сложные приложения и при этом не знать, как написать хотя бы одну строку кода на C, Pascal, COBOL или FORTRAN. Вместо того чтобы писать код, вы выбираете из библиотеки объекты и помещаете каждый из шх в нужное место на экране.

***Помни:***
*У объектов разных стандартных типов имеются характеризующие их свойства, а каждому типу объектов свойственны специальные событыя. С объектом можно также связать какой-либо метод. Метод – это процедура, написанная на процедурном языке. Впрочем, возможно создать очень сложные приложения, не написав при этом ни одного метода*.

Хотя и можно написать сложные приложения, не пользуясь при этом процедурным языком, но SQL вам рано или поздно все же понадобится. SQL настолько богат выразительными средствами, что их трудно или даже невозможно выразить в объектной парадигме. Поэтому RAD-инструменты позволяют вставлять операторы SQL в объектно-ориентированные приложения. Примером объектно-ориентированной среды разработки, в которой можно работать с SQL, является Borland C++ Builder. Microsoft Access – это несколько другая среда разработки приложений, которая позволяет использовать язык SQL совместно с процедурным языком VBA.

В главе 4 описывалось создание таблиц базы данных с помощью приложения Access. Эти операции представляют собой лишь малую часть возможностей Access. Основная цель этого инструмента – разработка приложений, которые обрабатывают данные в таблицах базы данных. Разработчик помещает объекты в формы, а затем назначает им свойства, события и возможные методы. Для обработки форм и объектов используется код VBA, содержащий встроенный язык SQL.

***Внимание:***
*Хотя RAD-инструменты, подобные Access, могут помочь в создании высококачественных приложений за сравнительно короткое время, но обычно эти инструменты работают только на одной или на нескольких платформах. Например, Access работает только в операционной системе Microsoft Windows. Не забывайте об этом, если хотите предусмотреть возможный перенос своих приложений на другую платформу*.

RAD-инструменты, такие как Access, являются предвестниками окончательного объединения процессов проектирования реляционных и объектно-ориентированных баз данных. При этом основные достоинства реляционного подхода и SQL сохранятся. К ним будет добавлена возможность быстрой – и сравнительно свободной от ошибок – разработки, свойственная объектно-ориентированному программированию.

# ODBC и JDBC

В этой главе…

- Определение ODBC
- Описание частей ODBC
- Использование ODBC в среде клиент/сервер
- Использование ODBC в Internet
- Использование ODBC в локальных сетях
- Использование JDBC

С каждым годом компьютеры одной организации или нескольких различных организаций все чаще соединяются друг с другом. Поэтому возникает необходимость в налаживании совместного доступа к базам данных по сети. Главным препятствием для совместного использования баз данных является несовместимость системного программного обеспечения и приложений, работающих на разных компьютерах. Важными этапами на пути преодоления этой несовместимости стали создание и продолжающаяся эволюция SQL.

К сожалению, "стандарт" SQL еще не реализован на практике в чистом виде. Производители СУБД, утверждающие, что их продукты совместимы с международным стандартом SQL, зачастую включают в свои реализации расширения, несовместимые с продуктами других производителей. Производители не склонны отказываться от своих расширений, так как покупатели к ним привыкли и зависят от них. Большим организациям для совместного использования различных реализаций СУБД необходим другой подход, не требующий от производителей приводить их продукты к общему знаменателю. Этим подходом и является **ODBC** (**Open DataBase Connectivity** – открытый интерфейс доступа к базам данных).

## ODBC

**ODBC** – это стандартный интерфейс между базой данных и приложением, взаимодействующим с ней. Наличие подобного стандарта позволяет приложению на клиентском компьютере получать доступ к любой базе данных на сервере, используя SQL. Единственное требование заключается в том, чтобы и клиентская, и серверная части поддерживали стандарт ODBC. ODBC 4.0 – текущая версия данного стандарта.

Приложение получает доступ к конкретной базе данных, используя специально разработанный под эту базу **драйвер**. Клиентская часть драйвера, работающая напрямую с приложением, должна соответствовать стандарту ODBC. Для приложения безразлично, какая СУБД установлена на сервере. Серверная часть драйвера адаптирована к конкретной базе данных. С использованием такой архитектуры приложения не только не нужно настраивать на определенную СУБД, а даже и знать не обязательно, какая именно СУБД используется. Драйвер скрывает различия между различными типами серверных частей СУБД.

## Интерфейс ODBC

**Интерфейс ODBC** – это стандартизированный набор определений, которые включают все необходимое для организации взаимодействия приложения и требуемой базы данных. Интерфейс ODBC определяет следующее.

- Библиотека вызовов функций.
- Стандартный синтаксис SQL.
- Стандартные типы данных SQL.

- Стандартный протокол соединения с базой данных.
- Стандартные коды ошибок.

Вызовы функций ODBC обеспечивают соединение приложения с сервером базы данных. выполнение операторов SQL и возврат результатов приложению.

Для того чтобы выполнить какое-либо действие с базой данных, необходимо использовать соответствующую команду SQL в качестве аргумента функции ODBC. При условии использования стандартного для ODBC синтаксиса SQL результат выполнения этой функции не зависит от того, какая база данных установлена на сервере.

## Компоненты ODBC

Интерфейс ODBC состоит из четырех функциональных компонентов. Благодаря каждому из них достигается гибкость ODBC, позволяющая взаимодействовать любым ODBC-совместимым клиентам и серверам. ODBC интерфейс включает в себя четыре уровня.

- **Приложение**. Это та часть ODBC, с которой непосредственно работает пользователь. Естественно, приложения могут быть не только в ODBC-совместимых системах. Однако включение приложения в интерфейс ODBC имеет свой смысл. Приложение должно быть написано с учетом использования в нем ODBC. Оно должно взаимодействовать с диспетчером драйвера в строгом соответствии со стандартом ODBC.
- **Диспетчер драйвера**. Это библиотека динамической компоновки (**dinamic link library**, DLL), обычно поставляемая Microsoft. Она загружает необходимые драйверы для системных источников данных (возможно, нескольких) и направляет вызовы функций приложения с помощью драйверов к соответствующим источникам данных. Диспетчер драйвера прямо управляет некоторыми вызовами функций ODBC, а также перехватывает и обрабатывает некоторые типы ошибок.
- **Драйвер**. В связи с тем, что источники данных могут отличаться друг от друга, причем в некоторых случаях весьма существенно, необходим способ преобразования стандартных вызовов функций ODBC в код конкретного источника данных. Этим занимается драйвер DLL. Каждый драйвер DLL получает вызовы функций посредством стандартного интерфейса ODBC и переводит их в код, понятный источнику данных. Как только источник данных возвращает результат, драйвер в обратном порядке преобразует его в стандартный для ODBC вид. Драйвер является основным элементом, который позволяет ODBC-совместимому приложению управлять структурой и содержимым ODBC-совместимого источника данных.
- **Источник данных**. Существует множество различных источников данных. Таким источником может быть база данных на основе реляционной СУБД, находящаяся на одном компьютере с приложением, или база данных на удаленном компьютере. В роли источника данных может выступать обходящийся без СУБД **индексно-последовательный файл** (ISAM file) на локальном или удаленном компьютере. Для каждого вида источников данных требуется свой драйвер.

## ODBC в среде клиент/сервер

В среде клиент/сервер интерфейс между клиентской и серверной частью называется **программным интерфейсом приложения** (application programmer's interface, API). API может быть как специальным, так и стандартным. **Специальным** (proprietary) API называется интерес, созданный для работы с определенной серверной частью. Программой, которая формирует этот интерфейс, является драйвер, и в специальной системе он называется **собственным драйвером** (native driver). Собственный драйвер оптимизирован для работы с определенной клиентской частью и связанной с ней серверной частью источника данных. В связи с тем, что собственные драйверы настроены как

для работы с приложением, так и с СУБД, они передают команды и информацию достаточно быстро и без задержек.

***Совет:***
*Если система клиент/сервер рассчитана на использование с определенным источником данных и заведомо не будет использовать другой, лучше воспользоваться собственным драйвером из поставки СУБД. С другой стороны, если система должна уметь работать с различными источниками данных, использование интерфейса ODBC избавит разработчика от выполнения огромного количества ненужной работы.*

Каждый драйвер ODBC создан для работы с конкретным источником данных, однако у всех них одинаковый интерфейс с диспетчером драйверов. Любой драйвер, не оптимизированный для работы с конкретным клиентом, скорее всего проиграет в быстродействии собственному драйверу. Основным недостатком первого поколения драйверов ODBC была их плохая производительность по сравнению с собственными драйверами. Последние измерения, однако, показали, что производительность драйверов ODBC 4.0 вполне сравнима с производительностью собственных драйверов. На сегодняшний день технология достигла уровня, когда уже не нужно жертвовать производительностью ради преимуществ стандартизации.

## ODBC и Internet

Операции с базами данных в Internet кое в чем серьезно отличаются от операций с базами данных в среде клиент/сервер. Самое заметное отличие с точки зрения пользователя заключается в клиентской части системы, которая включает в себя интерфейс пользователя. В системе клиент/сервер интерфейс пользователя – это часть приложения, которое связывается с источником данных на сервере через ODBC-совместимые операторы SQL. В World Wide Web клиентской частью системы является Web-браузер, который взаимодействует с источником данных на сервере с помощью протокола HTTP посредством языка HTML (**HyperText Markup Language**).

Так как любой Web-пользователь имеет доступ к данным в Internet, открытие доступа к базе данных в Internet называется **опубликованием базы данных**. Теоретически база данных в Internet доступна гораздо большему количеству людей, чем база данных на сервере в локальной сети. Обычно даже неизвестно, кто они. Таким образом, размещение данных в Internet больше похоже на публикацию их по всему миру, нежели на распределение информации между несколькими сотрудниками. На рис. 16.1 приведены различия между системой клиент/сервер и системой на базе Web.

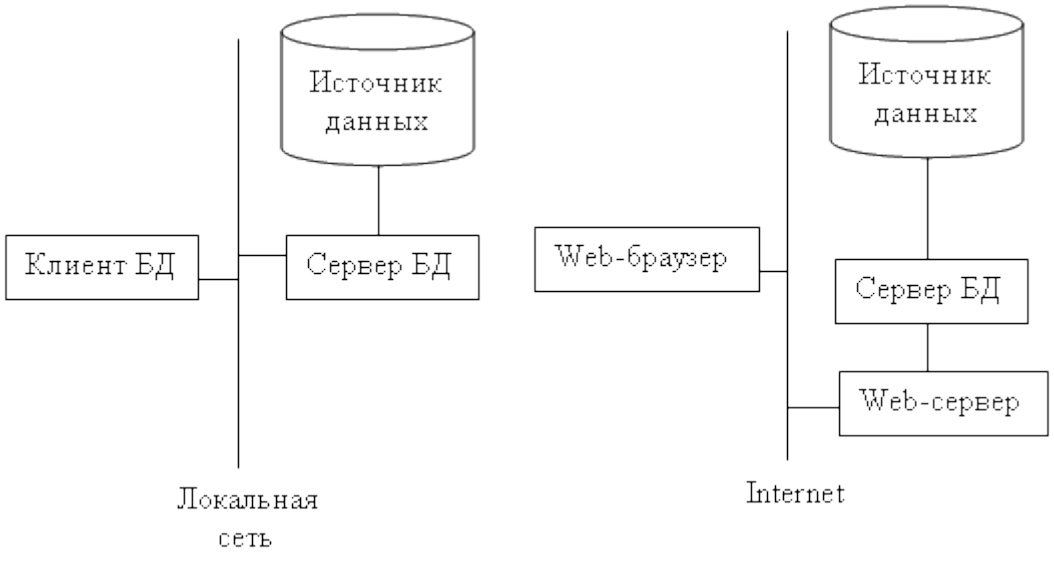


**Рис. 16.1**. Система клиент/сервер в сравнении с системой на базе Web

## Серверные расширения

В системе на базе Web общение между клиентским компьютером и Web-сервером происходит с помощью HTTP. **Серверное расширение** – это компонент системы, который переводит HTML-текст в ODBC-совместимый SQL-код, после чего сервер базы данных связывается с источником данных и выполняет этот код. В обратном направлении источник данных пересылает результат запроса серверу базы данных и далее серверному расширению, которое преобразует результат запроса в форму, понятную Web-серверу. Затем данные отсылают к клиентскому компьютеру по Internet, и Web-браузер пользователя их отображает. На рис. 16.2 приведена схема подобной системы.


**Рис. 16.2**. Система базы данных на основе Web с серверным расширением

## Клиентские расширения

Web-браузеры разрабатываются с целью создать простой и понятный интерфейс для любых Web-страниц. Программы Netscape Navigator, Microsoft Internet Explorer и Apple's Safari изначально не были предназначены для использования в качестве клиентской части базы данных, Чтобы добиться необходимого уровня взаимодействия с базой данных в Internet, необходимы дополнительные функциональные возможности. Для обеспечения этих возможностей были разработаны различные **клиентские расширения**. Эти расширения включают вспомогательные приложения, включаемые модули Netscape Navigator, управляющие элементы ActiveX, Java-аплеты и сценарии. Расширения общаются с сервером с помощью протокола HTTP на языке HTML. Любой HTML-код, который оперирует данными из базы, преобразуется серверным расширением в ODBC-совместимый SQL-код перед тем, как быть направленным к источнику данных.

## Вспомогательные приложения

Первые клиентские расширения назывались **вспомогательными приложениями** (helper application). Вспомогательное приложение – это самостоятельная программа, выполняющаяся на компьютере пользователя. Она не интегрирована в Web-страницу и не отображается в окне Web-браузера. Ее можно использовать как программу просмотра для графических файлов, формат которых не поддерживается браузером. Чтобы воспользоваться такой программой, ее необходимо установить. После загрузки рисунка подходящего формата браузер вызывает программу просмотра. Серьезным недостатком этого метода является то, что перед запуском вспомогательного приложения весь файл

данных должен быть записан во временный файл. Таким образом, для больших файлов время ожидания заметно увеличивается.

## Включаемые модули Netscape Navigator

**Модули Netscape Navigator** (Netscape Navigator plug-in) так же, как и вспомогательные приложения, помогают обрабатывать информацию, которую не понимает браузер. От последних они отличаются тем, что совместимы только с браузерами Netscape и в большей степени интегрированы с данными браузерами. Подобная интеграция позволяет браузеру отображать файл еще до полной его загрузки. Это качество является серьезным преимуществом такого рода программ. Пользователь не обязан ждать конца загрузки, чтобы увидеть изображение. Большое и всевозрастающее количество модулей Netscape делает доступным воспроизведение звука, общение с другими пользователями, анимацию, видео и интерактивную трехмерную виртуальную реальность. Для нас главное, что некоторые из модулей обеспечивают доступ к удаленным базам данных в Internet.

## Управляющие элементы ActiveX

Управляющие элементы Microsoft ActiveX подобны модулям Netscape, но работают по другому принципу. ActiveX базируется на ранней технологии OLE, разработанной Microsoft. Netscape поддерживает ActiveX, а также некоторые другие популярные технологии Microsoft. Естественно, Microsoft Internet Explorer совместим с ActiveX. Под контролем Netscape и Microsoft сейчас находится большая часть рынка браузеров.

## Аплеты Java

**Java** – язык подобный C++ и разработанный Sun Microsystems специально для создания клиентских Web-расширений. После того как соединение между клиентом и сервером установлено, аплет загружается на клиентский компьютер и запускается на нем. Подходящий аплет, встроенный на HTML-страницу, позволит получить удобный доступ к данным сервера. Схема работы Web-приложения базы данных с аплетом Java на клиентском компьютере приведена на рис. 16.3.

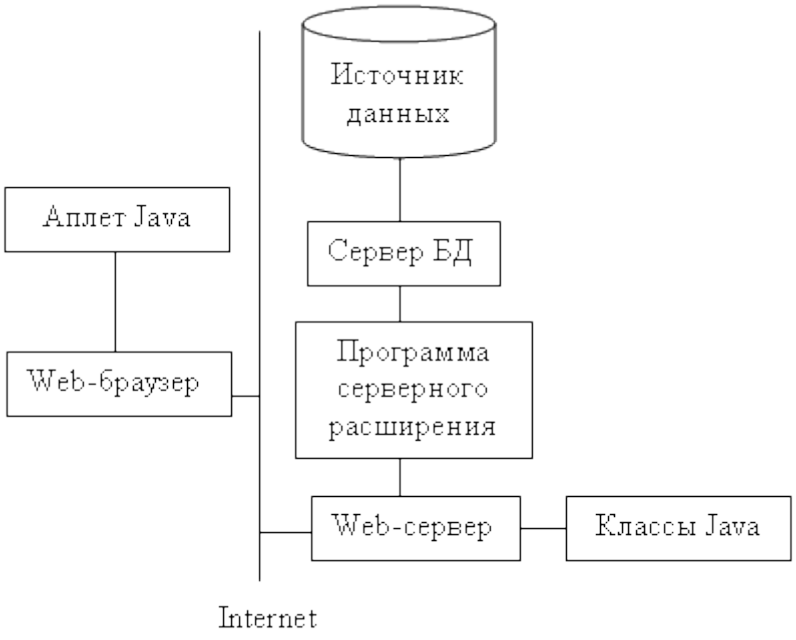


**Рис. 16.3**. Web-приложение базы данных, использующее аплет Java

Преимущество использования аплетов Java заключается в том, что они не устаревают. Так как аплеты каждый раз при использовании загружаются с сервера, клиент всегда имеет дело с последней версией аплета. Разработчикам можно не беспокоиться о возможной потере совместимости при переходе сервера на новое программное обеспечение. Надо лишь убедиться в том, что загружаемые аплеты Java совместимы с новой конфигурацией сервера. Положительный ответ будет означать, что все клиенты также совместимы.

## Сценарии

**Сценарии** (scripts) – наиболее гибкий инструмент для создания клиентских расширений. Использование языка сценариев, такого как Netscape JavaScript или Microsoft VBScript, позволяет максимально контролировать происходящее на клиентском компьютере. С их помощью можно организовать проверку достоверности ввода в поля формы, что позволит отбраковывать неправильно заполненные формы еще на клиентском компьютере. Это сэкономит время пользователей и уменьшит загрузку сети. Как и аплеты Java, сценарии встроены в страницу HTML и выполняются при открытии пользователем страницы.

## ODBC и интранет

**Интранет** – это локальная или глобальная сеть, работающая как упрощенная версия Internet. Так как вся сеть интранет принадлежит одной организации, то, как правило, отсутствует необходимость в применении комплексных мер безопасности, таких как брандмауэры. Все инструменты, разработанные для создания приложений в Internet, также подходят для создания приложений для интранет. ODBC работает в интранет точно так же, как и в Internet. При наличии нескольких различных источников данных клиенты, использующие Web-браузеры и соответствующие клиентские и серверные расширения, могут взаимодействовать с этими источниками посредством SQL-кода, передаваемого с помощью HTTP и ODBC. Драйвер ODBC переводит SQL-код в собственный язык команд базы данных и выполняет его.

## JDBC

**JDBC** (Java DataBase Connectivity – Java-интерфейс взаимодействия с базами данных) имеет как много общих черт с ODBC, так и несколько существенных отличий. Одно из отличий явствует из названия. Как и ODBC, JDBC – универсальный интерфейс к базе данных, не зависящий от источника данных на сервере. Разница в том, что клиентское приложение для JDBC может быть написано только на Java, а не на любом другом языке типа C++ или Visual Basic. Другое отличие состоит в том, что и Java, и JDBC с начала и до конца разрабатывались для использования в Internet.

**Java** – это полнофункциональный язык программирования, на котором создаются полноценные приложения для работы с базами данных для различных систем клиент/сервер. При этом приложение Java, работающее с базой данных через ODBC, очень похоже на ODBC-приложение, написанное на C++. Основная разница заключается в их работе в Internet или интранет.

Когда система функционирует в Internet, то условия ее работы отличаются от условий в системе клиент/сервер. Клиентская часть приложения, которая работает с Internet,– это браузер с минимальными вычислительными способностями. Эти способности должны быть увеличены, чтобы переложить на клиента часть работы с базой данных; и аплеты Java предоставляют такую возможность.

**Аплет** – это небольшое приложение, постоянно находящееся на сервере. Когда клиент соединяется по Internet с сервером, аплет загружается и запускается на клиентском компьютере. Аплеты Java разработаны таким образом, чтобы запускаться в **песочнице** (sandbox). "Песочница" – это определенное место в памяти клиентского компьютера, в котором выполняются аплеты Java. Аплет

не может взаимодействовать с чем-нибудь вне "песочницы". Такая архитектура позволяет защитить клиентский компьютер от потенциально опасных аплетов, которые могут получить доступ к секретной информации или нанести данным серьезный вред.

***Внимание:***
*Загружая исполняемый код из Internet, вы подвергаетесь определенной опасности. Даже Java-аплеты могут оказаться не такими уж и безобидными. Поэтому будьте осторожны при загрузке исполняемого кода с подозрительных серверов*.

Как и ODBC, JDBC передает операторы SQL от клиентской части приложения (аплета), запускаемого на компьютере клиента, к источнику данных на сервере. Также JDBC служит для передачи результатов выполнения запросов или сообщений об ошибках от источника данных обратно приложению. Польза от JDBC заключается в том, что разработчик аплетов может использовать стандартный интерфейс JDBC, не заботясь о том, какая база данных находится на сервере. JDBC выполняет все преобразования, необходимые для правильного двустороннего взаимодействия.

# SQL:2003 и XML

В этой главе…

- Использование SQL с XML
- XML, базы данных и Internet

---

Одной из самых существенных новых функциональных возможностей языка SQL:2003 является поддержка файлов XML (**extensible Markup Language** – расширяемый язык разметки), которые все больше становятся универсальным стандартом обмена данными между разнородными платформами. Для XML не имеет значения, с какой средой приложения, операционной системой или аппаратным обеспечением работает пользователь, которому предоставляются данные. Таким образом, XML позволяет построить "мост" из данных между двумя независимыми пользователями.

## Как XML связываемся, с SQL

XML, как и HTML, является языком разметки, а значит, не полнофункциональным языком, как, например, C++ или Java. Это даже не подъязык манипулирования данными, как SQL. Однако он достаточно осведомлен в отношении перемещаемых им данных. Там, где HTML имеет дело только с форматированием текста и графики в документе, XML позволяет работать со структурой содержимого документа. Однако сам XML не предназначен для прямого форматирования. Для осуществления задачи форматирования необходимо дополнить XML таблицей стилей, которая, как и в HTML, позволяет форматировать XML-документы.

SQL и XML структурируют данные двумя различными способами. Таким образом, вы можете сохранить данные и выбрать из них необходимую информацию одним из следующих способов.

- SQL представляет собой прекрасный инструмент для работы с числовыми и текстовыми данными, классифицированными по типам данных и имеющими точно описанный размер. Язык SQL был создан как стандартный инструмент для поддержки и работы с данными, хранящимися в реляционных базах данных.
- Если речь идет о произвольных данных, которые не могут быть однозначно классифицированы, то здесь лучше использовать язык XML. Движущей силой для разработчиков при создании языка XML было стремление обеспечить универсальный стандарт для перемещения данных между разнородными компьютерами и отображения их в World Wide Web.

SQL и XML как бы дополняют цели и возможности друг друга. Каждый несет в себе определенные непревзойденные преимущества в отношении собственных альянсов доменов и форм, что позволяет пользователю получить всю необходимую информацию в любое время и в любом месте.

## Тип данных XML

В SQL:2003 впервые был представлен новый для SQL тип данных – тип XML. Это означает, что согласованные между собой реализации могут хранить и работать непосредственно с данными формата XML, без их предварительного преобразования из какого-либо типа данных SQL в тип XML.

Несмотря на то что XML-данные встроены в любую поддерживающую их реализацию, они работают как тип данных, определяемый пользователем (UDT-тип, **user-defined type**). XML-данные

обеспечивают непосредственное взаимодействие языков SQL и XML, поскольку позволяют приложениям выполнять SQL-операции над содержимым XML и, наоборот, XML-операции над содержимым SQL. Вы можете использовать столбцы с XML-данными совместно со столбцами, содержащими любые предопределенные типы данных (см. главу 2), объединяя их в запросах с предложением WHERE. А ваша СУБД, в свою очередь, следуя истинным традициям реляционных баз данных, будет определять оптимальный путь для выполнения запроса, что в конце концов и произойдет.

## Когда используются XML-данные

Необходимость хранения данных в формате XML зависит от того, что вы планируете делать с этим данными. Принятие такого решения целесообразно в следующих случаях.

- Если необходимо хранить целый блок данных для последующего получения этих данных в том же полном блоке.
- Если необходимо сделать запрос для целого XML-документа. Некоторые реализации имеют расширенные возможности оператора EXTRACT, которые позволяют извлечь содержимое из XML-документа.
- Если необходимо точно ввести данные вне операторов SQL. Использование типа данных XML гарантирует истинность значений XML, а не только произвольных текстовых строк.
- Для обеспечения совместимости с будущими, еще не установленными системами хранения, которые могут не поддерживать существующие типы, например CLOB (подробнее об этом см. в главе 2).
- Для получения преимуществ в будущих оптимизациях, которые будут поддерживать только XML-данные.

## Когда XML-данные не используются

Иногда использование XML-данных не имеет никакого смысла, поскольку сегодня большинство данных в реляционных базах данных гораздо лучше работает в их текущем формате, нежели в формате XML. Для подтверждения вышесказанного приведем два примера. Итак, XML тип не используется в следующих случаях.

- При естественном разбиении данных в реляционной структуре на таблицы, строки и столбцы.
- Если необходимо обновить только некоторые части, а не весь документ.

## Преобразование данных из формата SQL в формат XML и наоборот

Для обмена данными между базами данных SQL и XML-документами различные элементы базы данных SQL должны быть преобразованы в эквивалентные элементы XML-документа и наоборот. Как вы узнаете в следующих разделах, порой преобразование просто необходимо для удачного выполнения некоторых операций.

## Преобразование наборов символов

В языке SQL поддержка наборов символов зависит от его реализации. Это означает, что приложение DB2 производства компании IBM может поддерживать наборы символов, которые не поддерживаются приложением SQL Server компании Microsoft. SQL Server, в свою очередь, поддерживает наборы символов, которые не поддерживаются приложением Oracle. Несмотря на то что большинство общих наборов символов практически всегда универсально поддерживается тем или иным приложением, использование мало распространенных символов может усложнить

перемещение вашей базы данных и приложения с одной платформы реляционной СУБД (РСУБД) на другую.

У XML нет никаких проблем совместимости с наборами символов, поскольку он поддерживает только один набор – Unicode. Этот универсальный набор символов, публикуемый консорциумом Unicode Consortium, представляет собой стандартный набор символов для компьютеров, в котором каждому письменному знаку на любом языке присвоен определенный номер. С точки зрения обмена данными между любыми реализациями языка SQL и XML это очень хорошо. Поставщики РСУБД должны описать преобразование строк из каждого набора символов в символы Unicode и обратное преобразование символов Unicode в строки, состоящие из наборов символов. К счастью, XML не поддерживает множество наборов символов, что освобождает поставщиков от большого количества проблем, связанных с бесчисленными преобразованиями символов.

## Преобразование идентификаторов

В отличие от SQL, XML более точно подходит к определению идентификаторов. Прежде чем стать частью XML-документа, символы, допустимые в SQL и недопустимые в XML, должны быть соответствующим образом преобразованы. SQL поддерживает неограниченные идентификаторы. Это означает, что все виды добавочных символов, такие как %, $ и &, будут допустимыми до тех пор, пока они заключены в двойные кавычки. Но такие символы не допустимы для XML. Кроме того, имена в XML, начинающиеся с символов XML, в любых комбинациях уже зарезервированы и, таким образом, не могут использоваться без каких-либо негативных последствий. Именно поэтому идентификаторы SQL, начинающиеся с этих символов, должны быть изменены.

При преобразовании из SQL в XML все идентификаторы конвертируются в Unicode. Любые же идентификаторы языка SQL, которые являются также допустимыми именами XML, остаются неизменными. Символы идентификатора языка SQL, не допустимые для имен XML, заменяются шестнадцатеричным кодом. Полученный результат имеет форму записи типа "_xHHHH_" или "_xHHHHHHHH_", где H– шестнадцатеричный разряд верхнего регистра. Например, символ подчеркивания "_" будет представлен как "_x005F_". Двоеточие– как "_x003A_". Эти представления являются кодами для описания в системе Unicode таких символов, как подчеркивание и двоеточие. В случае, если идентификатор SQL начинается с символов x, т или /, перед такими символами необходимо поставить префикс с кодом в форме "_xFFFF_".

Преобразовать символы из XML-формата в SQL-формат гораздо проще. Все, что для этого необходимо сделать, – это развернуть символы XML-имени в последовательность "_XFFFF_" или "_XFFFFFFFF_". Всякий раз, когда вы находите такую последовательность, заменяйте ее символами, соответствующими символам Unicode. Если же имя XML начинается с символов "_XFFFF_", игнорируйте их.

Следуя этим простым правилам, вы можете преобразовывать идентификатор SQL в XML-имя, а затем вернуться к идентификатору SQL. Однако это хорошее правило не действует для преобразования XML-имени в идентификатор SQL и наоборот.

## Преобразование типов данных

В SQL:2003 определено, что данные типа SQL преобразуются в наиболее близкую схему XML-данных. Формулировка "**наиболее близкая**" означает, что все значения, допустимые для SQL-типа, будут допустимы и для типа XML-схемы, а наименее возможные значения, не допустимые для SQL-типа, будут допустимы для типа XML-схемы. Элементы XML, такие как maxlnclusive и minlnclusive, могут ограничивать значения, допускаемые типом XML-схемы, до значений, допускаемых соответствующим SQL-типом. Например, если тип SQL-данных Ограничивает значения типа

INTEGER в диапазоне значений от -2157483648 до 2157483647, в XML значение minlnclusive может быть задано числом – 2157483648.

Ниже приведен пример такого преобразования.

```
<xsd:simpleType>
<xsd:restriction base="xsd:integer">
<xsd:maxlnclusive value="2157483647"/>
<xsd:minlnclusive value="-21574 83648"/>
<xsd:annotation>
<sqlxml:sqltype name="INTEGER"/>
</xsd:annotation>
</xsd:restriction>
</xsd:simpleType>
```

Раздел примечаний содержит информацию из описания SQL-типа, который в настоящий момент не используется XML, но позднее, при преобразовании этого документа в SQL-формат, может быть очень кстати.

## Преобразование таблиц

Вы можете преобразовать таблицу в XML-документ, а также все таблицы в схему или все таблицы в каталог. Преобразование также определяет привилегии. Пользователь, имеющий привилегию SELECT только для нескольких столбцов таблицы, может преобразовать в XML-документ только эти столбцы. В действительности преобразование порождает два документа: один, который содержит данные в таблице, а другой – XML-схему, описывающую первый документ. Ниже приведен пример преобразования SQL-таблицы в документ, содержащий XML-данные.

```
<CUSTOMER>
<row>
<FirstName>Abe</FirstName>
<LastName>Abelson</LastName>
<City>Springfield</City>
<AreaCode>714</AreaCode>
<Telephone>555-llll</Telephone>
</row>
<row>
<FirstName>Bill</FirstName>
<LastName>Bailey</LastName>
<City>Decatur</City>
<AreaCode>714</AreaCode>
<Telephone>555-2222</Telephone>
</row>
.
.
.
</CUSTOMER>
```

Основной элемент документа дал имя таблице (CUSTOMER). Каждая строка таблицы содержится в пределах элемента <row> (в данном примере таких строк две). Кроме того, каждая строка таблицы содержит последовательность элементов столбца (атрибутов), каждый из которых получает имя после связывания со столбцом в исходной таблице (столбцы FirstName, LastName, City, AreaCode, Telephone). Каждый элемент столбца содержит значение данных.

## Обработка неопределенных значений

Поскольку SQL-данные могут быть неопределенными значениями, необходимо решить, как представлять их в XML-документе. Неопределенные значения могут быть представлены либо как

нуль, либо как отсутствие всякого значения. Если элемент столбца необходимо представить как неопределенное (нулевое) значение, то он будет иметь атрибут xsi:nil="true". Это может быть сделано следующим способом:

```
<row>
<FirstName>Bill</FirstName>
<LastName>Bailey</LastName>
<City xsi:nil="true" />
<AreaCode>714</AreaCode>
<Telephone>5 55-22 22</Telephone>
</row>
```

Если элемент столбца отсутствует, то выполните следующее:

```
<row>
<FirstName>Bill</FirstName>
<Las tName>Bailey</Las tName>
<AreaCode>714</AreaCode>
<Telephone>55 5-22 22</Telephone>
</row>
```

При выборе этой опции строка содержит неопределенное значение, которое "отсутствует". С ним нет связей.

## Создание XML-схемы

При преобразовании данных из SQL в XML первый созданный документ содержит данные, а второй – информацию о схеме. В качестве примера рассмотрим схему для документа CUSTOMER (см. "Преобразование таблиц").

```
<xsd:schema>
<xsd:simpleType name="CHAR_15">
<xsd:restriction base="xsd:string">
<xsd:lenght value = "15"/>
</xsd;restriction>
</xsd:simpleType>
<xsd:simpleType name="CHAR_25">
<xsd:restriction base="xsd:string">
<xsd:lenght value = "25"/>
</xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name="CHAR_3">
<xsd:restriction base="xsd:string">
<xsd:lenght value = "3"/>
</xsd:restriction>
</xsd:simpleType>
<xsd:simpleType name-"CHAR_8">
<xsd:restriction base="xsd:string">
<xsd:lenght value = "8"/>
</xsd:restriction>
</xsd:simpleType>
<xsd:sequence>
<xsd:element name="FirstName" type="CHAR_15"/>
<xsd:element name="LastName" type="CHAR_25"/>
<xsd:element
name="City" type="CHAR_25" nillable="true"/>
<xsd:element
name="AreaCode" type="CHAR_3" nillable="true"/>
<xsd:element
name="Telephon" type="CHAR_8" nillable="true"/>
<xsd:sequence>
```

```
</xsd:schema>
```

Эта схема подходит в том случае, если при обработке неопределенных значений используется опция "нуль". Опция "отсутствие" требует несколько иного определения элемента. Например:

```
<xsd:element
name="City" type="CHAR_25 minOccurs="0"/>
```

## Получение XML результатов при использовании операторов SQL

SQL: 2003 имеет пять операторов, XMLELEMENT, XMLFOREST, XMLGEN, XMLCONCAT и XMLAGG, которые при применении к содержимому базы данных SQL дают XML-результат.

## Оператор XMLELEMENT

Оператор XMLELEMENT создает XML-элемент. Этот оператор может использоваться в операторе SELECT для помещения данных формата XML в базу данных SQL. Рассмотрим следующий пример:

```
SELECT c.LastName
XMLELEMENT (NAME "City", c.City) AS "Result"
FROM CUSTOMER C
WHERE LastName="Abelson";
```

Возвращаемый результат:

LastName Result
Abelson <City>Springfield</City>

## Оператор XMLFOREST

Оператор XMLFOREST создает **дерево (forest)** элементов из списка аргументов. Каждый аргумент оператора создает новый элемент. Ниже приведен пример использования такого оператора.

```
SELECT c. LastNartie
XMLFOREST (c.City,
c.AreaCode,
c.Telephone) AS "Result"
FROM CUSTOMER c
WHERE LastName="Abelson" OR LastName="Bailey";
```

В результате вы получите следующую информацию:

| **LastName** | **Result** |
|---|---|
| | <City>Springfield</City> |
| Abelson | <AreaCode>714</AreaCode> |
| | <Telephone>555-llll</ Telephone> |
| | <City>Decatur</City> |
| Bailey | <AreaCode>714</AreaCode> |
| | <Telephone>555-llll</ Telephone> |

## Оператор XMLGEN

Первый аргумент оператора XMLGEN – это шаблон, который содержит символы-заполнители для значений, которые будут добавлены позже. Символы-заполнители представлены в форме "{$name}". Последовательность аргументов задает значения и связанные с ними имена, которые характеризуют шаблон. Ниже приведен пример использования этого оператора.

```
SELECT c.LastName
XMLGEN ('<CUSTOMER Name="{$LASTNAME}">
<City>{$CITY}</City>
</CUSTOMERS,
c.LastName AS Name,
c City) AS "Result"
FROM CUSTOMER c
WHERE LastName="Abelson" OR LastName="Bailey";
```

Результат будет следующим:

| LastName | Result |
|---|---|
| Abelson | <CUSTOMER Name="Abelson"<br><City>Springfield</City><br></CUSTOMER> |
| Bailey | <CUSTOMER Name="Bailey"<br><City>Decatur</City><br></CUSTOMER> |

## Оператор XMLCONCAT

Оператор XMLCONCAT обеспечивает альтернативный способ создания дерева элементов. Это осуществляется путем связывания его XML-аргументов. Например:

```
SELECT c.LastName
XMLCONCAT (
XMLELEMENT (NAME "first", c.FirstName,
XMLELEMENT (NAME "last", c.LastName)
) AS "Result" FROM CUSTOMER c;
```

В результате получится следующее:

| LastName | Result |
|---|---|
| Abelson | <first>Abe</first><br><last>Abelson</last> |
| Bailey | <first>Bill</first><br><last>Bailey</last> |

## Оператор XMLAGG

**XMLAGG** – это агрегирующая функция, которая создает единственный XML-документ из других XML-документов или их отдельных фрагментов. Агрегирование содержит дерево элементов. Рассмотрим это на примере:

```
SELECT XMLELEMENT
(NAME "City",
XMLATTRIBUTES (c.City AS "name"),
XMLLAGG (XMLELEMENT (NAME "last" c.LastNa,e)
)
) AS "CityList"
FROM CUSTOMER C
GROUP BY City;
```

При обработке таблицы CUSTOMER этот запрос выведет следующее:

```
<City name="Decatur">
<last>Bailey</last>
</City>
<City name= "Philo">
<last>Stetson</last>
<last>Stetson</last>
<last>Wood</last>
</City
<City name="Springfield">
<last>Abelson</last>
</City>
```

## Преобразование не предопределенных типов данных в XML

В SQL:2003 к не предопределенным типам данных относятся домены, отдельные UDT-типы, строки, массивы и мультимножества. Для преобразования каждого из этих XML-типов данных используется соответствующий XML-код. Примеры преобразования таких типов данных рассматриваются в следующих разделах.

## Домены

Для того чтобы преобразовать SQL-домен в XML, вначале этот домен необходимо создать. Для создания домена воспользуемся оператором CREATE DOMAIN:

```
CREATE DOMAIN WestCoast AS CHAR (2)
CHECK (State IN ('CA\ 'OR1, ' WA', ' AK '));
```

Теперь создадим таблицу, которая использует этот домен:

```
CREATE TABLE WestRegion (
ClientName Character (20) NOT NULL,
State      WestCoast      NOT NULL
);
```

Ниже приведена XML-схема преобразования домена в XML.

```
<xsd:simpleType>
Name='DOMAIN.SalesWestCoast'>
<xsd:annotation> <xsd:appinfo>
<sqlxml:sqltype kind='DOMAIN'
schemaName='Sales'
typeName='WestCoast'
mappedType='CHAR_2'
finals'true'/>
<xsd:appinfo>
</xsd:annotation>
<xsd:restriction base='CHAR_2'/>
```

```
</xsd:simpleType>
```

После применения этого преобразования вы получите XML-документ примерно следующего содержания:

```
<WestRegion>
<row>
.
.
.
State>AK</State>
.
.
.
</row>
.
.
.
</WestRegion>
```

## Отдельные UDT-типы

С отдельными UDT-типами вы можете делать то же самое, что и с доменами, однако здесь необходим более строгий подход к определению типа. Например:

```
CREATE TYPE WestCoast AS Character (2) FINAL;
```

Для преобразования этого типа в XML-тип используется следующая XML-схема:

```
<xsd:simpleType>
Name='UDT.SalesWestCoast'>
<xsd:annotation>
<xsd:appinfo>
<sqlxml:sqltype kind='DISTINCT'
schemaName='Sales'
typeName='WestCoast'
mappedType='CHAR_2'
final='true'/>
<xsd:appinfo>
</xsd:annotation>
<xsd:restriction base='CHAR_2'/>
</xsd:simpleType>
```

В результате создается элемент, похожий на созданный для домена, описанного нами ранее.

## Строки

Тип ROW позволяет поместить целую строку с ценной информацией в одно лишь отдельное поле строки таблицы. Тип ROW создается как часть описания таблицы следующим образом:

CREATE TABLE CONTACTINFO (
Name CHARACTER (30)
Phone ROW (Home CHAR (13), Work CHAR (13))

Теперь воспользуемся следующей XML-схемой для преобразования этого типа:

```
<xsd:complexType Name='ROW.1'>
<xsd:annotation>
<xsd:appinfo>
<sqlxml:sqltype kind='ROW'
<sqlxml:field name='Home'
mappedType='CHAR_13'/>
<sqlxml:field name='Work'
mappedType='CHAR_13'/>
</sqlxml:sqltype>
<xsd:appinfo>
</xsd:annotation>
<xsd:sequence>
<xsd:element Name='Home' nillable='true'
Type='CHAR_13'/>
<xsd:element Name='Work' nillable='true'
Type='CHAR_13'/>
</xsd:sequence>
</xsd:complexType>
```

Такое преобразование создает для столбца следующий XML-тип:

```
<Phone>
<Home>(888)555-llll</Home>
<Work>(888)555-1212</Work>
</Phone>
```

## Массивы

Если вы хотите поместить в одно поле больше одного элемента, вместо типа ROW воспользуйтесь типом Array. Для этого примера в таблице CONTACTINFO объявим столбец Phone как массив, а затем создадим XML-схему, которая будет преобразовывать массив в XML:

Например:

CREATE TABLE CONTACTINFO (
Name CHARACTER (30)
Phone CHARACTER (13) ARRAY [4]
);

Теперь воспользуемся следующей XML-схемой для преобразования этого типа:

```
<xsd:complexType Name='ARRAY_4.CHAR_13'>
<xsd:annotation>
<xsd:appinfo>
<sqlxml:sqltype kind='ARRAY1
maxElements='4'
mappedElementType='CHAR_13'/>
</xsd:appinfo>
</xsd:annotation>
<xsd:sequence>
<xsd:element Name='element'
minOccurs='0' maxOccurs='4'
nillable='true' type='CHAR_13'/>
</xsd:sequence >
</xsd:complexType>
```

Результат будет примерно следующим:

```
<Phone>
<element>(888)555-llll</element>
<element>xsi:nil='true'/>
<element>(888)555-3434</element>
</Phone>
```

Элемент в массиве, содержащий xsi:nil='true', говорит о том, что второй телефонный номер в исходной таблице содержит неопределенное значение.

## Мультимножества

Номера телефонов из предыдущего примера могут также хорошо храниться и в мультимножествах. Для преобразования мультимножества воспользуемся следующим программным кодом:

```
CREATE TABLE CONTACTINFO (
Name CHARACTER (30)
Phone CHARACTER (13) MULTISET
);
```

Преобразуем этот тип в XML с помощью такой схемы:

```
<xsd:complexType Name='MULTISET.CHAR_13'>
<xsd:annotation>
<xsd:appinfo>
<sqlxml:sqltype kind='MULTISET'
mappedElementType='CHAR_13'/>
</xsd: appinfo>
</xsd:annotation>
<xsd:sequence>
<xsd:element Name='element'
minOccurs='0' maxOccurs='unbounded'
nillable='true' type='CHAR_13'/>
</xsd:sequence >
</xsd:complexType>
```

Результат будет примерно следующим:

```
<Phone>
<element>(888)555-llll</element>
<element>xsi:nils' true'I>
<element>(888)555-3434</element>
</Phone>
```

# Курсоры

В этой главе…

- Определение области действия курсора в операторе declare
- Открытие курсора
- Построчная выборка данных
- Закрытие курсора

---

SQL отличается от большинства наиболее популярных языков программирования тем, что в нем операции производятся одновременно с данными всех строк таблицы, в то время как процедурные языки обрабатывают данные построчно. Благодаря использованию **курсоров** в SQL становится возможным выводить, обновлять или же удалять выбранную строку в один прием, упрощая совместное использование SQL с другими языками программирования.

По существу, курсор является указателем на определенную строку таблицы. С его помощью можно вывести, обновить или удалить строку, на которую он ссылается.

Курсоры незаменимы, если требуется выбрать строки из таблицы, проверить их содержимое, а также выполнить различные операции на основании содержимого полей. Одного только SQL в данном случае недостаточно. С помощью SQL можно извлекать строки, однако для принятия решений на основе содержимого полей лучше использовать процедурные языки. Курсоры позволяют SQL по одной извлекать строки из таблицы и передавать их в процедурный код для обработки. Поместив код SQL в цикл, можно строка за строкой полностью обработать всю таблицу.

В случае использования встроенного SQL в общем все выглядит так:

```
EXEC SQL DECLARE CURSOR выражение
EXEC SQL OPEN выражение
Проверка достижения конца таблицы
Программный код
Начало цикла
Программный код
EXEC SQL FETCH
Программный код
Проверка достижения конца таблицы
Конец цикла
EXEC SQL CLOSE выражение
Программный код
```

В приведенном фрагменте кода командами SQL являются: DECLARE (объявить), OPEN (открыть), FETCH (вызвать) и CLOSE (закрыть). Каждая из этих команд детально рассматривается в этой главе.

***Совет:***
*Если для выполнения операций с выбранными строками можно обойтись обычными операторами SQL, лучше так и сделать. Используйте для построчной обработки базовый язык лишь в том случае, если обычный SQL не позволяет выполнить необходимые операции.*

## Объявление курсора

Чтобы использовать курсор, необходимо сначала объявить о его существовании СУБД. Это делается с помощью оператора DECLARE CURSOR. Фактически она не инициирует никакого действия, а

только объявляет имя курсора для СУБД и определяет запрос, с которым будет работать курсор. Синтаксис оператора DECLARE CURSOR имеет следующий вид:

```
DECLARE имя_курсора [<чувствительность>]
[<перемещаемость>]
CURSOR [<состояние_фиксации>] [<возвращаемость>]
FOR выражение_запроса
[ORDER BY порядок_сортировки]
[FOR разрешение_обновления];
```

***Примечание:***
*имя курсора однозначно определяет курсор, следовательно, оно должно отличаться от любого другого имени курсора в модуле или программе*.

Чтобы код был удобочитаемым, курсор следует назвать мнемонически, т.е. чтобы из названия следовало, для чего он предназначен. Название курсора должно быть связано либо с данными, определенными в запросе, либо с операциями, которые с этими данными выполняет программный код.

***Совет:***
*Чувствительность курсора может быть установлена в состояние SENSITIVE, INSENSITIVE или ASENSITIVE. Перемещаемость курсора может быть в состоянии SCROLL (с прокруткой) или NO SCROLL (без прокрутки). Состояние фиксации курсора может быть либо WITH HOLD (с фиксацией), либо WITHOUT HOLD (без фиксации), а возвращаемость – WITH RETURN (с возвратом) либо WITHOUT RETURN (без возврата)*.

## Выражение запроса

Выражением запроса может быть любой допустимый оператор SELECT. Оператор SELECT выводит строку, на которую указывает курсор в данный конкретный момент времени. Эти строки представляют собой область действия курсора.

***Помни:***
*Запрос не будет выполняться сразу же после выполнения оператора DECLARE CURSOR. Выборка данных обязательно начинается с оператора OPEN. Построчная обработка данных осуществляется в цикле с использованием оператора FETCH*.

## Предложение ORDER BY

В зависимости от того, что программный код должен делать с данными, иногда требуется обрабатывать выбранные данные в определенном порядке. Сортировку строк перед их обработкой можно выполнять с помощью предложения ORDER BY (упорядочить по). Его синтаксис имеет следующий вид:

```
ORDER BY спецификация_сортировки [, спецификация_сортировки]…
```

Существует множество спецификаций сортировки, каждая из которых имеет следующий синтаксис:

```
(имя_столбца) [ COLLATE BY имя_сопоставления ] [ ASC | DESC ]
```

Для сортировки по столбцу данный столбец должен присутствовать в списке выборки в выражении запроса. Сортировку по столбцам таблицы, не вошедшим в список запроса, выполнить нельзя. Например, требуется произвести операцию, не поддерживаемую SQL, со строками таблицы CUSTOMER. В данном случае можно использовать курсор:

```
DECLARE custl CURSOR FOR
SELECT CustID, FirstName,LastName, City, State, Phone
FROM CUSTOMER
ORDER BY State, LastName, FirstName
```

В этом примере оператор SELECT выводит строки, упорядоченные вначале по штату (State), затем по фамилии (LastName) и по имени (FirstName). Перед тем как перейти к первому заказчику из штата Алабама (AL), выбираются все заказчики из штата Аляска (AK). Затем все записи о заказчиках из Аляски сортируются по их фамилиям (Aaron предшествует Abbot). Там, где фамилии совпадают, сортировка производится по имени (George Aaron предшествует Нету Aaron).

Приходилось ли вам когда-нибудь делать 40 копий документа, состоящего из 20 страниц, на копировальном аппарате без сортировщика? Ужас, до чего противная работа! Необходимо организовать место для 20 кучек, которые будут соответствовать 20 страницам документа, и ходить между ними 40 раз туда и обратно, раскладывая 40 копий по этим кучкам. Такой процесс называется **сопоставлением**. Аналогичный процесс возможен и в SQL.

**Сопоставление** – это набор правил, определяющих порядок сортировки строк данных. Сортировка по алфавитному порядку установлена по умолчанию. Однако можно выбрать отличный от установленного по умолчанию порядок, по которому будут упорядочиваться данные. Для этого следует использовать необязательный оператор COLLATE BY (использовать сопоставление). Любая реализация изначально поддерживает несколько часто встречающихся сопоставлений. Можно выбрать одно из них, указав лишь, как нужно упорядочивать данные – **по возрастанию** или **по убыванию**. Для этого в конце предложения нужно добавить ключевое слово ASC или DESC соответственно.

В операторе DECLARE CURSOR можно использовать столбец, рассчитанный на основе существующих столбцов таблицы. У такого столбца нет имени, которое может использоваться в предложении ORDER BY. Чтобы иметь возможность сослаться на этот столбец, его имя нужно определить в выражении запроса оператора DECLARE CURSOR. Рассмотрим следующий пример:

```
DECLARE revenue CURSOR FOR
SELECT Model, Units, Price,
Units * Price AS ExtPrice
FROM TRANSDETAIL
ORDER BY Model, ExtPrice DESC;
```

В этом примере нет ни одного оператора COLLATE BY в предложении ORDER BY, таким образом, использовалось сопоставление по умолчанию. В четвертом столбце, ExtPrice (общая цена), находятся данные о совокупной цене изделий определенной модели. В предложении ORDER BY задается вначале сортировка по названию модели Model, затем по общей цене ExtPrice. Сортировка по столбцу ExtPrice производится по убыванию (ключевое слово DESC), т.е. вначале обрабатываются более дорогие транзакции.

В предложении ORDER BY по умолчанию используется порядок сортировки по возрастанию. Если в списке спецификаций присутствует сортировка DESC и следующая сортировка также должна производиться по убыванию, для нее следует указать ключевое слово DESC в явном виде. Например, выражение:

```
ORDER BY A, B DESC, C, D, E, F
```

…эквивалентно такому:

```
ORDER BY A ASC, B DESC, C ASC, D ASC, E ASC, F ASC
```

## Разрешение обновления

Возможны случаи, когда необходимо обновить либо удалить строки таблицы, выбираемые с помощью курсора. В других случаях требуется гарантированно исключить такое обновление или удаление. SQL предоставляет возможность осуществлять контроль с помощью предложения разрешения обновления в операторе DECLARE CURSOR. Чтобы запретить обновление и удаление в области действия курсора, используйте оператор:

```
FOR READ ONLY
```

Для разрешения обновления избранных столбцов, при том, что остальные столбцы должны быть защищены, используйте:

```
FOR UPDATE OF имя_столбца [, имя_столбца]…
```

***Помни:***
*Естественно, столбцы, внесенные в приведенный список, должны присутствовать в выражении запроса оператора DECLARE CURSOR. Если предложение разрешения (или запрета) обновления отсутствует, по умолчанию данные всех столбцов, указанных в выражении запроса, могут быть обновлены. В этом случае оператор UPDATE может обновить все столбцы строки области действия курсора, а оператор DELETE может удалить такую строку.*

## Чувствительность

Выражение запроса в операторе DECLARE CURSOR определяет строки, которые находятся в области действия курсора. Возникает вопрос: что если программный код, расположенный между операторами OPEN и CLOSE, изменяет содержимое некоторых строк так, что они перестают удовлетворять условиям запроса? А если некоторые строки вообще удалены? Продолжает ли команда курсора обрабатывать все строки, которые изначально соответствовали условиям запроса, или же изменяет свое поведение в зависимости от изменений, а может, вообще игнорирует измененные или удаленные строки?

***Помни:***
*До окончания построчной обработки в результате запроса DECLARE CURSOR данные могут находиться в полном беспорядке. Такие данные недостоверны и противоречивы. Потому рекомендуется сделать курсор нечувствительным к любым изменениям. Для этого в оператор DECLARE CURSOR нужно добавить ключевое слово INSENSITIVE. В течение времени, когда курсор остается открытым, он нечувствителен к изменениям таблицы, так что область действия курсора распространяется на строки, соответствующие исходным условиям выражения запроса. Курсор не может быть нечувствительным и обновляемым в одно и то же время. Нечувствительный курсор должен быть открыт только для чтения.*

Обычные операторы SQL, такие как UPDATE, INSERT или DELETE, работают с набором строк таблицы базы данных, а возможно, со всеми строками таблицы. Во время работы механизм транзакций SQL защищает ее от вмешательства других команд, работающих одновременно с теми же данными. При использовании же курсора данные более уязвимы. В течение всего времени, пока курсор открыт, данные могут быть изменены другими командами. Если открывается один курсор и начинается обработка таблицы, затем открывается другой курсор, а первый курсор остается активным, действия, производимые вторым курсором, могут повлиять на данные, используемые командой первого курсора. Предположим, составлены следующие запросы:

```
DECLARE Cl CURSOR FOR SELECT * FROM EMPLOYEE
ORDER BY Salary;
DECLARE C2 CURSOR FOR SELECT * FROM EMPLOYEE
```

```
FOR UPDATE OF Salary;
```

Теперь предположим, что оба курсора открыты и командой курсора C1 последовательно выбрано несколько строк, а в это время данные по зарплате обновлены с помощью C2. При этом может так получиться, что строка, уже выбранная C1, может опять появиться при последующей выборке.

***Проблемы:***
*Проблемы, возникающие в результате взаимодействий нескольких одновременно открытых курсоров или обыкновенных команд SQL и открытых курсоров, можно избежать только путем локализации транзакций. В противном случае можно нарваться на крупные неприятности. Хорошенько запомните: нельзя работать со множеством открытых курсоров.*

По умолчанию чувствительность курсора находится в состоянии ASENSITIVE. Значение ASENSITIVE зависит от реализации SQL. В одной реализации оно может быть эквивалентно SENSITIVE, а в другой – INSENSITIVE. Для того чтобы выбрать, какое значение использовать в вашей реализации, прочитайте техническую документацию к вашей системе.

## Перемещаемость

Понятие **перемещаемость** (scrollability) впервые появилось в SQL-92. В реализациях, поддерживаемых SQL-86 и SQL-89. единственным разрешенным перемещением курсора было последовательное, начиная с первой строки, выбранной выражением запроса, и заканчивая последней строкой. Ключевое слово SCROLL оператора DECLARE CURSOR в SQL-92 (и в SQL:2OO3) позволяет получить доступ к строкам в любом порядке по вашему желанию. Синтаксис оператора FETCH контролирует перемещение курсора. Оператор FETCH мы рассмотрим ниже в этой главе.

## Открытие курсора

Оператор DECLARE CURSOR определяет, какие строки включать в курсор, но фактически не инициирует никакого действия, являясь лишь объявлением. Оператор OPEN активизирует курсор. Она имеет следующий вид:

```
OPEN имя_курсора;
```

Для открытия курсора, приведенного выше в разделе, посвященном предложению ORDER BY, используйте синтаксис:

```
DECLARE revenue CURSOR FOR
SELECT Model, Units, Price,
Units * Price AS ExtPrice
FROM TRANSDETAIL
ORDER BY Model, ExtPrice DESC;
OPEN revenue;
```

***Помни:***
*До тех пор пока курсор не открыт, осуществлять выборку строк из курсора невозможно. После открытия курсора значения переменных, содержащиеся в операторе DECLARE CURSOR, становятся фиксированными, это же относится к функциям даты-времени. Рассмотрим следующий пример:*

```
DECLARE CURSOR C1 FOR SELECT * FROM ORDERS
WHERE ORDERS.Customer=:NAME
AND DueDate < CURRENT_DATE;
NAME: = 'Acme Co'; //Выражение базового языка
```

```
OPEN C1;
NAME: = 'Omega Inc.'; //Другое выражение базового языка
…
UPDATE ORDERS SET DueDate = CURRENT_DATE;
```

Оператор OPEN фиксирует значение всех переменных, приведенных в объявлении курсора, а также значения всех функций даты-времени. Поэтому второе присваивание имени переменной (NAME: = 'Omega Inc.') не влияет на выбранные курсором строки. Это значение NAME будет использоваться при последующем открытии C1. Если даже команда OPEN выполняется до полуночи, а оператор UPDATE выполняется после, значение CURRENT_DATE в операторе UPDATE вычисляется во время выполнения команды OPEN. Это справедливо даже в том случае, если в операторе DECLARE CURSOR нет функций даты-времени.

## Внутренняя фиксация (для значений даты-времени)

Похожая "фиксация" значений даты-времени имеет место при выполнении обычных операторов. Посмотрите пример кода:

```
UPDATE ORDERS SET RecheckDate = CURRENTJDATE WHERE;
```

Пусть сделано несколько заказов. Выполнение кода начинается за минуту до наступления полночи и продолжается пять или десять минут. Если выражение использует функцию CURRENT_DATE (или TIME, или же TIMESTAMP), значение последней на время выполнения выражения фиксируется. Таким образом, все строки ORDERS имеют дело с одинаковым значением текущей даты. Подобным образом обстоит дело и со значениями типа TIMESTAMP, в выражении используется только одно значение, независимо от того, сколько времени работает выражение. Ниже приведен интересный пример проявления такого правила.

```
UPDATE EMPLOYEE SET KEY CURRENT_TIMESTAMP;
```

Можно подумать, что выражение присваивает уникальное значение идентификатора каждому сотруднику, однако оно присваивает одно и то же значение в каждой строке. Итак, команда OPEN фиксирует значения даты-времени для всех команд, используемых в курсоре.

## Выборка данных из отдельных строк

Оператор DECLARE CURSOR определяет название курсора и область его действия, оператор OPEN собирает все строки таблицы, удовлетворяющие выражению запроса DECLARE CURSOR, оператор FETCH осуществляет непосредственную выборку данных. Курсор может указывать на одну строку в области действия курсора, или на область, которая находится перед первой строкой области действия, или же на область, расположенную за последней строкой области действия курсора, или на границу строк. Определить, на что именно указывает курсор, можно, используя предложение ориентации оператора FETCH.

## Синтаксис

Синтаксис оператора FETCH имеет следующий вид:

```
FETCH [[ориентация] FROM] имя_курсора
INTO целевая_спвцификация [,целевая_спецификация]…;
```

Существуют шесть вариантов ориентации:

- NEXT

- PRIOR
- FIRST
- LAST
- ABSOLUTE
- RELATIVE
- <Спецификация_простого_значения>

По умолчанию используется NEXT – единственно возможное значение параметра ориентации в SQL до версии SQL-92. Где бы ни находился курсор, он перемещается на следующую строку в порядке, установленном выражением запроса. Если курсор расположен перед первой записью, он завершает свое выполнение на первую запись. Если он указывает на запись п, то перемещается к записи 77+1. Если курсор указывает на последнюю запись, он прекращает свою работу и в системную переменную SQLSTATE помещается код отсутствия данных. Переменная SQLSTATE, а также остальные средства обработки ошибок в SQL рассматриваются в главе 20.

Целевыми спецификациями являются либо базовые переменные, либо параметры, в зависимости от того, определен ли курсор во встроенном SQL или в модуле. Количество и типы целевых спецификаций должны соответствовать количеству и типам столбцов, используемых в выражении запроса оператора DECLARE CURSOR. В случае встроенного SQL, если из строки таблицы выбирается список из пяти значений, в выражении запроса должны фигурировать пять переменных с правильно выбранными типами для хранения этих значений.

## Ориентация перемещаемого курсора

Вследствие того, что в SQL курсор может быть перемещаемым, помимо NEXT можно использовать и другие значения параметра ориентации. После перемещения с ориентацией PRIOR курсор перемещается к предыдущей строке. В случае ориентации FIRST курсор перемещается к первой записи, а в случае ориентации LAST – к последней записи выборки.

Задание ориентации в виде числа требует присутствия команд ABSOLUTE и RELATIVE. Например, FETCH ABSOLUTE 7 перемещает курсор на седьмую строку от начала выборки. FETCH RELATIVE 7 перемещает курсор на семь строк ниже его текущей позиции. FETCH RELATIVE 0 не перемещает курсор.

Оператор FETCH RELATIVE 1 производит такое же действие, что и FETCH NEXT; FETCH RELATIVE-1 действует таким же образом, что и FETCH PRIOR; FETCH ABSOLUTE 1 выбирает первую запись; FETCH ABSOLUTE 2 – вторую запись и т.д. Аналогично, FETCH ABSOLUTE – 1 предоставляет последнюю запись, а FETCH ABSOLUTE – 2 – предшествующую последней записи выборки и т.д. Оператор FETCH ABSOLUTE 0 генерирует **исключение** отсутствия данных, таким же образом действует FETCH ABSOLUTE 17, если в выборке только 16 строк. Оператор **FETCH <спецификация_простого_значения>** возвращает запись, заданную спецификацией простого значения.

## Позиционные операторы DELETE и UPDATE

Со строками, на которые в данный момент указывает курсор, можно выполнять операции удаления и обновления. Синтаксис оператора DELETE следующий:

```
DELETE FROM имя_таблицы WHERE CURRENT OF имя_курсора;
```

Если курсор не указывает на строку, генерируется ошибка, и удаление не происходит. Вот синтаксис оператора UPDATE:

```
UPDATE имя_таблицы
SET имя_столбца = значение [,имя_столбца = значение]…
WHERE CURRENT OF имя_курсора;
```

Значение, помещаемое в каждый выбранный столбец, должно являться выражением, ключевым словом или ключевым словом DEFAULT. Если при проведении операции обновления возникает какая-либо ошибка, обновление не выполняется.

## Закрытие курсора

***Совет:***
*Закончив работу с курсором, тут же закройте его. Курсоры, оставшиеся открытыми после того, как приложение закончило свою работу с ними, как правило, безобидны, но опять-таки не всегда. Кроме того, открытые курсоры используют системные ресурсы.*

По окончании работы "нечувствительного" курсора и его закрытия достаточно повторно открыть этот курсор, и он сможет работать с измененными данными.

# Постоянно хранимые модули

В этой главе…

- Сложные команды, атомарность, курсоры, переменные и состояния
- Управляющие структуры
- Создание циклов
- Использование хранимых процедур и функций
- Предоставление полномочий на выполнение
- Создание и использование хранимых модулей

---

В течение многих лет ведущие специалисты в области технологий баз данных работали над разработкой стандартов. После выхода очередного стандарта и начала его применения пользователями и программистами во всем мире сразу начинается процесс разработки следующего. Так было и в случае SQL-92. Семь лет разделяет выпуск SQL-92 и выход первого компонента SQL: 1999. Однако все эти годы продолжалась кипучая деятельность, в результате которой ANSI и ISO выпустили дополнение к SQL-92, названное SQL-92/PSM (Persistent Stored Modules – **постоянно хранимые модули**). Это дополнение послужило основой для одного из разделов стандарта SQL: 1999 с тем же названием. SQL/PSM как часть SQL:2003 определяет набор операторов, предоставляющих SQL возможности по созданию управляющих структур, обычных для наиболее мощных языков программирования. Благодаря этому стало возможным решать многие задачи только с использованием SQL, без привлечения других программных средств. Предыдущие версии SQL требовали непрерывного переключения между SQL и процедурным базовым языком.

## Составные команды

В этой книге SQL рассматривается как непроцедурный язык, который применяется в основном для обработки наборов данных, а не отдельных записей. В то же время в этой главе вы узнаете, что это положение постепенно меняется. SQL становится процедурным языком, в основных чертах все еще оставаясь средством для обработки наборов данных. Версия SQL, определяемая стандартом SQL-92, не соответствовала процедурной модели, в которой команды выполняются одна за другой в определенной последовательности. Команды SQL были одиночными и, как правило, встроенными в код программы C++ или Visual Basic. Фактически с помощью ранних версий SQL пользователи не могли создать запрос или выполнить некоторые другие операции с помощью последовательности операторов SQL. Выполнение же последовательности команд SQL вызывало значительное снижение производительности. Каждая выполняющаяся команда SQL должна послать сообщение с места работы пользователя на сервер баз данных, а тот должен отослать ответ в обратном направлении. Это увеличивает сетевой обмен и тем самым приводит к дополнительной нагрузке на сеть.

SQL: 1999 и SQL:2003 предоставляют возможность создания составных команд, образованных отдельными командами SQL, выполняемыми в один прием. Использование составных команд позволяет снизить загрузку сети, поскольку все отдельные команды, образующие составную, вместе пересылаются на сервер, возвращающий обратно один ответ.

Все команды, включенные в составную команду, должны быть расположены между ключевыми словами BEGIN и END. Например, чтобы вставить данные во множество связанных таблиц, следует использовать следующий синтаксис:

```
void main {
EXEC SQL
BEGIN
```

```
INSERT INTO students (StudentID, Fname, Lname)
VALUES (:sid,:sfname,:sid);
INSERT INTO roster (ClassID, Class, StudentID)
VALUES (:cid, rename,:sid);
INSERT INTO receivable (StudentID, Class, Fee)
VALUES (:sid,:cname,:cfee);
END;
/* Проверка значения SQLSTATE на код ошибки */
}
```

Приведенный выше небольшой фрагмент программы на языке C содержит встроенную составную команду SQL. Комментарии относятся к обработке ошибок. Если почему-то составная команда не выполнилась, в специальный параметр SQLSTATE будет помещен код ошибки. Понятно, что комментарий не может обработать ошибку. Я поместил его лишь с целью напомнить о том, что в реальной программе код обработки ошибок находится в этом месте.

## Атомарность

Составные команды вызывают ошибки, никогда не встречающиеся в случае простых команд SQL. Выполнение обычной команды SQL может завершиться успешно или неудачно. Если команда не смогла выполниться, то в базе данных не происходит изменений. В случае составных команд это не так. Посмотрите на предыдущий пример.

## Что в имени твоем?

Название дополнения SQL – постоянно хранимые модули (**persistent stored modules**) – очень метко описывает их назначение.

Да вы уже и сами могли догадаться. Если слово модуль вам неизвестно, освежите в памяти материал главы 15. По логике хранимый модуль должен являться чем-то таким, что находится в некотором месте до тех пор, пока не потребуется. Кроме того, постоянно хранимый модуль должен быть чем-то, что находится под рукой в течение определенного периода времени. Вот, собственно, и все, о чем рассказывается в этой главе.

Вы почти угадали, хотя логика немного увела вас в сторону. В этой главе описываются подобные модули. Но постоянно хранимые модули больше связаны с другими возможностями SQL. В SQL:2003 огромное количество постоянно хранимых модулей содержат необходимые функции, которые следовало включить еще в SQL-92, но этого не произошло. Такие функции значительно повышают мощность и полезность SQL, и они должны были где-то находиться. Некоторые из них действительно хранятся под своими именами (в частности, хранимые функции и хранимые процедуры). Ладно, довольно логики. Важно лишь, что эти новые средства существуют и работают.

Что если операции вставки в таблицы Students (студенты) и Roster (расписание) произошли, а в результате вмешательства постороннего пользователя операция вставки в таблицу Receivable (плата за учебу) не выполнилась. Получится, что студент будет зачислен, но счет за обучение ему выписан не будет. Такие ошибки могут слишком дорого обойтись университету. Для предотвращения развития событий по данному сценарию необходимо ввести концепцию **атомарности**. Атомарная команда является неделимой. Она либо выполняется целиком, либо не выполняется вовсе. Простые команды SQL атомарны по своей природе. Другое дело – составные команды SQL. Однако и составную команду тоже можно определить атомарной. В приведенном ниже примере составная команда SQL становится безопасной благодаря введению атомарности.

```
void main {
EXEC SQL
BEGIN ATOMIC
```

```
INSERT INTO students (StudentID, Fname, Lname)
VALUES (:sid,:sfname,:sid);
INSERT INTO roster (ClassID, Class, StudentID)
VALUES (:cid,:cname,:sid);
INSERT INTO receivable (StudentID, Class, Fee)
VALUES (:sid,:cname,:cfee);
END;
/* Проверка значения SQLSTATE на код ошибки */
}
```

Ключевые слова ATOMIC после BEGIN гарантируют выполнение всей команды полностью, а в случае возникновения ошибки происходит откат к исходному состоянию базы данных.

## Переменные

Все высокоуровневые языки программирования, такие как C или Basic, позволяют использовать **переменные**. До появления SQL/PSM переменные в SQL использовать было нельзя. Переменные являются символическими именами значений определенного типа. В составных командах можно объявить переменную и определить ее значение. В ходе выполнения составной команды переменная может использоваться. После завершения выполнения составной команды все переменные, объявленные в ней, уничтожаются. Таким образом, переменные в SQL являются локальными переменными составной команды, в которой они объявлены. Рассмотрим следующий пример:

```
BEGIN
DECLARE prezpay NUMERIC;
SELECT salary
INTO prezpay
FROM EMPLOYEE
WHERE jobtitle= 'president';
END;
```

## Курсоры

В составных командах можно использовать курсоры. Как уже упоминалось в главе 18, курсоры используются для построчной обработки данных. В составной команде можно объявить курсор, использовать его, а затем забыть о нем, так как курсор уничтожается сразу после завершения выполнения составной команды. Ниже приведен пример использования курсора.

```
BEGIN
DECLARE ipocandidate CHARACTER (30);
DECLARE cursorl CURSOR FOR
SELECT company
FROM biotech;
OPEN cursor1;
FETCH cursor1 INTO ipocandidate;
CLOSE cursor1;
END;
```

## Состояния

Все знают, что означает, когда о человеке говорят, что он дошел до "кондиции". Его состояние наутро после этого может быть тяжелым, очень тяжелым, или же все обойдется. Так и в SQL. Результат выполнения команды SQL может быть успешным, сомнительным или неправильным. Каждый из таких результатов является "состоянием" (**condition**).

Каждый раз, когда выполняется команда SQL, сервер базы данных обновляет значение параметра статуса SQLSTATE. Он представляет собой поле из пяти символов. Значение SQLSTATE содержит информацию об успешном или неудачном завершении команды. В случае возникновения ошибки значение содержит ее код.

Первые два из пяти символов значения класса SQLSTATE содержат информацию о том, выполнены ли команды SQL успешно, непонятно как или неудачно. В табл. 19.1 приведены четыре возможных результата.

**Таблица 19.1**. Значения класса sqlstate.

| Класс | Описание |
|---|---|
| 00 | Успешное завершение |
| 01 | Предупреждение |
| 02 | Не найден |
| Другое | Сгенерировано исключение |

Значение класса '00' означает, что предыдущая команда SQL выполнена успешно. В большинстве случаев это наиболее желаемый результат.

Значение класса '01' означает предупреждение. Предупреждение указывает на то, что в ходе выполнения команды SQL случилось что-то незапланированное. В данном случае СУБД не "знает", произошла ошибка или нет. Предупреждение обращает внимание разработчика на то, что требуется тщательно проверить команду SQL.

Значение класса '02' означает, что не получено никаких данных в результате выполнения команды SQL. Это может быть хорошим или плохим известием, в зависимости от обстоятельств. Иногда результат в виде пустой таблицы – как раз то, что требуется разработчику.

Любой код, отличный от значений '00', '0Г или '02', указывает на наличие ошибки в программе. Три последних символа параметра SQLSTATE содержат код ошибки. Два символа кода класса и три символа кода ошибки вместе составляют пять символов значения SQLSTATE.

## Обработка состояний

Программу можно написать таким образом, чтобы значение параметра SQLSTATE контролировалось после выполнения каждой команды SQL. Итак, используем приобретенные знания на практике.

- При получении сообщения кода класса '00' вряд ли необходимо предпринимать дополнительные действия. Продолжайте делать то, что вы планировали.
- **При получении сообщения кода класса '01' или '02' разработчик должен решить, как программа должна отреагировать**. Если это допустимые состояния программы, выполнение должно продолжиться по плану. В том случае, если такие сообщения не должны появляться, необходимо перейти к выполнению специальной процедуры, обрабатывающей подобные ситуации.
- **Получение какого-либо другого класса кода указывает на то, что в программе имеются ошибки**. В этом случае следует перейти к процедуре обработки исключения. Конкретная процедура, на которую должно перейти выполнение программы, зависит от содержимого параметра SQLSTATE. Если существует возможность множества различных состояний, для каждого следует использовать собственную процедуру обработки, так как различные

состояния часто должны обрабатываться по-разному. Некоторые ошибки несущественны или поправимы. Другие являются фатальными и приводят к аварийному завершению приложения.

## Объявление обработчиков состояний

Обработчик состояния можно поместить в составную команду. При создании обработчика вначале следует объявить состояние, которое обработчик должен обрабатывать. Объявленное состояние может быть исключением или же некоторым другим состоянием. В табл. 19.2 представлены возможные состояния, а также кратко описаны причины возникновения состояния каждого вида.

**Таблица 19.2**. Состояния, которые можно определить в обработчике.

| Состояние | Описание |
|---|---|
| SQLSTATE VALUE ' xxyyy ' | Соответствующее значение sqlstate |
| SQLEXCEPTION | Класс sqlstate, отличный от '00', ' 01' или '02' |
| SQLWARNING | Класс ' 01' |
| NOT FOUND | Класс ' 02' |

Ниже приведен пример объявления состояния.

```
BEGIN
DECLARE constraint_violation CONDITION
FOR SQLSTATE VALUE '23 000';
END;
```

Приведенный пример не является взятым из жизни, поскольку внутри структуры BEGIN…END должны располагаться и команда SQL, ответственная за возникновение особого состояния, и его обработчик.

## Действие и эффект обработчика

Как только возникает состояние, вызывающее соответствующий обработчик, выполняется действие, определенное этим обработчиком. Такое действие может быть простой или составной командой SQL. После успешного выполнения обработчика выполняется некоторое последействие, называемое эффектом. Ниже представлен список трех возможных **эффектов** обработчика.

- **CONTINUE**. Выполнение команды, следующей после команды, инициировавшей действие обработчика.
- **EXIT**. Выход из составной команды, содержащей обработчик, и выполнение следующей команды.
- **UNDO**. Отмена всех предыдущих команд составной и выполнение следующей команды.

Эффект CONTINUE лучше всего применять в том случае, если обработчик способен устранить любую проблему, вызвавшую его. Эффект EXIT применяется, если обработчику не под силу исправить проблему, но нет необходимости отменять изменения, сделанные составной командой. Эффект UNDO позволяет вернуться в состояние до выполнения составной команды. Рассмотрим следующий пример:

```
BEGIN ATOMIC
DECLARE constraint_violation CONDITION
FOR SQLSTATE VALUE '23000';
DECLARE UNDO HANDLER
FOR constraint_violation
RESIGNAL;
INSERT INTO students (StudentID, Fname, Lname)
VALUES (:sid, sfname,:slname);
INSERT INTO roster (ClassID, Class, StudentID)
VALUES (:sid, cname,:sid);
END;
```

Если выполнение какой-либо из двух команд INSERT вызывает нарушение некоторого ограничения, например, в результате добавления записи с первичным ключом, уже присутствующим в таблице, в параметр SQLSTATE помещается значение 23000 и возникает состояние constraint_violation. Обработчик такого события отменяет изменения, внесенные в таблицы командами INSERT. Команда RESIGNAL возвращает управление процедуре, вызвавшей процедуру обработчика.

После успешного выполнения обеих команд INSERT следующей выполняется команда за ключевым словом END.

***Помни:***
*Ключевое слово ATOMIC является обязательным в случае эффекта UNDO. Это не распространяется на обработчики с эффектами CONTINUE или EXIT.*

## Необрабатываемые состояния

Предположим, в предыдущем примере возникнет состояние, при котором значение SQLSTATE будет отличным от '23000'. Ваш обработчик не может обработать подобное состояние. Так как в текущей процедуре такое состояние не предусмотрено, выполняется команда RESIGNAL. После этого состояние может быть обработано на более высоком уровне. Если же и на этом уровне состояние нельзя обработать, она передается на еще более высокий уровень и т.д., пока состояние не будет обработано или пока не будет сгенерирована ошибка в основном приложении. Основная идея вышесказанного заключается в следующем: если во время выполнения команды SQL возможно появление исключений, необходимо написать обработчики всех возможных исключений. Это позволит намного упростить отладку.

## Присваивание

С появлением SQL/PSM SQL наконец-то получил возможность, которую всегда имели даже самые примитивные процедурные языки, – возможность присваивания значения переменной. Присваивание выглядит так:

```
SET переменная = значение;
```

Где **переменная** – имя переменной, а **значение** – выражение. Ниже приведено несколько примеров присваивания.

```
SET vfname = 'Brandon';
SET varea = 3.1416 *:radius *:radius;
SET vhiggsmass = NULL;
```

## Операторы управления ветвлением

Основным недостатком стандарта SQL-86, не позволяющим считать SQL полноценным процедурным языком, являлось отсутствие управляющих структур. До появления SQL/PSM строгую последовательность выполнения команд нельзя было нарушить без использования базового языка, такого как C или Basic. SQL/PSM получил в свое распоряжение аналогичные управляющие структуры, позволяя тем самым решать многие задачи без привлечения других языков программирования.

## IF…THEN…ELSE…END IF

Основной управляющей структурой является IF…THEN…ELSE…END IF. Если условие IF истинно, выполняются команды предложения THEN. В противном случае выполняются команды предложения ELSE. Например:

```
IF
vfname = 'Brandon'
THEN
UPDATE students
SET Fname = 'Brandon'
WHERE StudentID = 314159;
ELSE
DELETE FROM students
WHERE StudentID = 314159;
END IF
```

В приведенном примере, если переменная vfname содержит значение 'Brandon', в списке студентов произойдет обновление записи с идентификатором 314159– в поле Fname будет помещено значение 'Brandon1. Если же переменная содержит какое-либо другое значение, отличное от Brandon', запись с идентификатором 314159 будет удалена из таблицы Students.

Выражение IF…THEN…ELSE…END IF больше всего подходит, если в зависимости от выполнения некоторого условия существуют два варианта дальнейшего поведения программы. Однако очень часто приходится иметь дело с большим количеством вариантов. В таких случаях следует использовать оператор CASE.

## CASE…END CASE

Существуют два вида оператора CASE: простой оператор CASE и оператор CASE с поиском. В зависимости от выполнения нескольких условий выполнение программы происходит по одному из нескольких вариантов.

## Простой оператор CASE

Простой оператор CASE вычисляет одно выражение. В зависимости от его значения выполнение программы происходит одним из нескольких возможных путей. Например:

```
CASE vinajor
WHEN 'Computer Science'
THEN INSERT INTO geeks (StudentID, Fname, Lname)
VALUES(:sid,:sfname,:slname);
WHEN 'Sports Medicine'
THEN INSERT INTO jocks (StudentID, Fname, Lname)
VALUES(:sid, rsfname,:slname);
ELSE INSERT INTO undeclared (StudentID, Fname, Lname)
VALUES(:sid,:sfname,:slname);
```

```
END CASE
```

Команды предложения ELSE выполняются, если vmajor не попадает ни в одну из категорий, заданных предложениями THEN.

***Внимание:***
*Оператор ELSE является необязательным. Однако, если он отсутствует и выражение CASE не соответствует ни одному предложению THEN, SQL генерирует исключение*.

## Оператор CASE с поиском

Оператор CASE с поиском аналогичен простому оператору CASE, за исключением того, что используется несколько условных выражений, а не одно. Например:

```
CASE
WHEN vmajor
IN ('Computer Science', 'Electrical Engineering')
THEN INSERT INTO geeks (StudentID, Fname, Lname)
VALUES(:sid,:sfname,:slname);
WHEN vclub
IN ('Amateur Radio', 'Rocket1,'Computer')
THEN INSERT INTO geeks (StudentID, Fname, Lname)
VALUES(:sid,:sfname,:slname);
ELSE
INSERT INTO poets (StudentID, Fname, Lname)
VALUES(:sid,:sfname,:slname);
END CASE
```

Чтобы избежать возможности возникновения исключения, следует поместить всех студентов, которые не являются "физиками" (geeks), в таблицу poets ("лирики") – Конечно, не все "не физики" являются "лириками" – ведь есть же еще и "футболисты", не говоря уже о "химиках". Это не меняет сути дела, так как в оператор CASE всегда можно добавить еще несколько предложений WHEN.

## LOOP…END LOOP

Структура LOOP позволяет многократно выполнять некоторую последовательность команд SQL. После выполнения последней команды SQL, находящейся внутри структуры LOOP…ENDLOOP, цикл передает управление первой команде цикла, и все команды цикла выполняются еще раз. Синтаксис структуры LOOP…ENDLOOP имеет следующий вид:

```
LOOP
SET vcount=vcount+1;
INSERT INTO asteroid (AsteroidID)
VALUES (vcount);
END LOOP
```

Приведенный фрагмент кода готовит заготовку для таблицы астероидов, заполняя ее записями пока без данных, но с уникальными идентификаторами. По мере изучения астероидов эту таблицу можно будет заполнить и другими данными.

Необходимо отметить, что данный фрагмент кода имеет один существенный недостаток. Созданный им цикл является бесконечным. В нем нет кода окончания, следовательно, добавление в таблицу строк будет продолжаться до тех пор, пока СУБД будет располагать местом на диске. Затем система в лучшем случае сгенерирует исключение, а в худшем – просто аварийно завершит работу.

Для практического использования оператора LOOP необходимо иметь возможность выхода из цикла без возникновения исключения. Для этого имеется оператор LEAVE.

## LEAVE

Оператор LEAVE (покинуть) действует согласно названию. Как только выполнение программы доходит до оператора LEAVE с меткой, он передает управление команде, расположенной сразу после помеченной. Например:

```
AsteroidPreload:
SER vcount = 0;
LOOP
SET vcount = vcount+1;
IF vcount > 1000
THEN
LEAVE AsteroidPreload;
END IF;
INSERT INTO asteroid (AsteroidID)
VALUES (vcount);
END LOOP AsteroidPreload
```

Приведенный выше код создает 1000 последовательно пронумерованных записей в таблице астероидов и затем завершает цикл.

## WHILE…DO…END WHILE

Оператор WHILE предоставляет другой метод многократного выполнения последовательности команд SQL. Если условие оператора WHILE истинно, цикл продолжает выполняться. Если нет – выполнение цикла тут же прекращается. Например:

```
AsteroidPreload2:
SET vcount = 0;
WHILE
vcount < 100 0 DO
SET vcount = vcount+1;
INSERT INTO asteroid (AsteroidID)
VALUES (vcount);
END WHILE AsteroidPreload2
```

Результат работы этого кода точно такой же, как в предыдущем случае. Это еще раз подтверждает то, что в SQL существует много методов решения поставленной задачи. Забота программиста – выбрать наиболее подходящий.

## REPEAT…UNTIL..END REPEAT

Цикл REPEAT очень похож на цикл WHILE. Различие между ними заключается в том, что условие проверяется после выполнения команд цикла. Например:

```
AsteroidPreload3:
SET vcount = 0;
REPEAT
SET vcount = vcount+1;
INSERT INTO asteroid (AsteroidID)
VALUES (vcount);
UNTIL vcount = 1000
END REPEAT AsteroidPreload3
```

***Совет:***
*Хотя во всех трех приведенных выше примерах одна и та же операция с одинаковой легкостью выполнялась тремя различными методами (с помощью операторов LOOP, WHILE и REPEAT), очень часто встречаются ситуации, когда один из методов имеет явные преимущества перед другими. Поэтому лучше хорошо знать все три метода и в конкретных обстоятельствах выбрать наиболее подходящий.*

## FOR…DO…END FOR

Цикл FOR в SQL объявляет и открывает курсор, выполняет выборку строк курсора, выполняет команды тела цикла FOR для каждой строки, а затем закрывает курсор. Такой цикл позволяет построчно обрабатывать данные в SQL без обращения к базовому языку. Если реализация SQL поддерживает циклы FOR, их можно использовать в качестве альтернативы: курсорам, описанным в главе 18. Ниже приведен пример цикла FOR.

```
FOR vcount AS Cursl CURSOR FOR
SELECT AsteroidID FROM asteroid
DO
UPDATE asteroid SET Description = 'stony iron1
WHERE CURRENT OF Cursl;
END FOR
```

В данном примере происходит обновление каждой строки таблицы Asteroid путем ввода 'stony iron' в поле Description. Такой способ ввода данных очень быстрый, но не совсем правильный. Вероятно, следовало бы вначале проверить спектральные характеристики астероидов, а уже затем индивидуально вводить эту информацию в базу данных.

## LEAVE

Оператор LEAVE (покинуть) действует согласно названию. Как только выполнение программы доходит до оператора LEAVE с меткой, он передает управление команде, расположенной сразу после помеченной. Например:

```
AsteroidPreload:
SER vcount = 0;
LOOP
SET vcount = vcount+1;
IF vcount > 1000
THEN
LEAVE AsteroidPreload;
END IF;
INSERT INTO asteroid (AsteroidID)
VALUES (vcount);
END LOOP AsteroidPreload
```

Приведенный выше код создает 1000 последовательно пронумерованных записей в таблице астероидов и затем завершает цикл.

## WHILE…DO…END WHILE

Оператор WHILE предоставляет другой метод многократного выполнения последовательности команд SQL. Если условие оператора WHILE истинно, цикл продолжает выполняться. Если нет – выполнение цикла тут же прекращается. Например:

```
AsteroidPreload2:
SET vcount = 0;
```

```
WHILE
vcount < 100 0 DO
SET vcount = vcount+1;
INSERT INTO asteroid (AsteroidID)
VALUES (vcount);
END WHILE AsteroidPreload2
```

Результат работы этого кода точно такой же, как в предыдущем случае. Это еще раз подтверждает то, что в SQL существует много методов решения поставленной задачи. Забота программиста – выбрать наиболее подходящий.

## REPEAT…UNTIL..END REPEAT

Цикл REPEAT очень похож на цикл WHILE. Различие между ними заключается в том, что условие проверяется после выполнения команд цикла. Например:

```
AsteroidPreload3:
SET vcount = 0;
REPEAT
SET vcount = vcount+1;
INSERT INTO asteroid (AsteroidID)
VALUES (vcount);
UNTIL vcount = 1000
END REPEAT AsteroidPreload3
```

***Совет:***
*Хотя во всех трех приведенных выше примерах одна и та же операция с одинаковой легкостью выполнялась тремя различными методами (с помощью операторов LOOP, WHILE и REPEAT), очень часто встречаются ситуации, когда один из методов имеет явные преимущества перед другими. Поэтому лучше хорошо знать все три метода и в конкретных обстоятельствах выбрать наиболее подходящий.*

## FOR…DO…END FOR

Цикл FOR в SQL объявляет и открывает курсор, выполняет выборку строк курсора, выполняет команды тела цикла FOR для каждой строки, а затем закрывает курсор. Такой цикл позволяет построчно обрабатывать данные в SQL без обращения к базовому языку. Если реализация SQL поддерживает циклы FOR, их можно использовать в качестве альтернативы: курсорам, описанным в главе 18. Ниже приведен пример цикла FOR.

```
FOR vcount AS Cursl CURSOR FOR
SELECT AsteroidID FROM asteroid
DO
UPDATE asteroid SET Description = 'stony iron1
WHERE CURRENT OF Cursl;
END FOR
```

В данном примере происходит обновление каждой строки таблицы Asteroid путем ввода 'stony iron' в поле Description. Такой способ ввода данных очень быстрый, но не совсем правильный. Вероятно, следовало бы вначале проверить спектральные характеристики астероидов, а уже затем индивидуально вводить эту информацию в базу данных.

## ITERATE

Оператор ITERATE позволяет изменять последовательность выполнения команд в циклах SQL. Циклы SQL представлены операторами LOOP, WHILE, REPEAT и FOR. Если условие в операторах

циклов является истинным или не задано, тут же оператор ITERATE начинает следующую итерацию цикла. Если же условие итерации является ложным или неопределенным, цикл завершается и выполняются следующие за ним команды:

```
AsteroidPreload4:
SET vcount = 0;
WHILE
vcount < 1000 DO
SET vcount=vcount+1;
INSERT INTO asteroid (AsteroidID)
VALUES (vcount);
ITERATE AsteroidPreload4;
SET vpreload = 'DONE';
END WHILE AsteroidPreload4
```

Оператор ITERATE все время передает управление в начало цикла WHILE, но до тех пор, пока переменная vcount не будет равна 9999. В следующей итерации переменная vcount увеличится до 10000, выполнится оператор INSERT, после чего оператор ITERATE прекратит итерации, переменная vpreload получит значение 'DONE', а выполнение перейдет к команде, следующей за циклом.

## Хранимые процедуры

**Хранимые процедуры** (stored procedures) находятся на сервере баз данных, а не на компьютере пользователя, как это было до появления SQL/PSM. Хранимая процедура должна быть определена, после чего ее можно вызвать с помощью команды CALL. Хранение процедуры на сервере уменьшает сетевой обмен и повышает производительность. Команда CALL является единственным сообщением, передаваемым от пользователя к серверу. Ниже приведен пример создания процедуры.

```
EXEC SQL
CREATE PROCEDURE MatchScore
(IN white CHAR (20),
IN black CHAR (20),
IN result CHAR (3),
OUT winner CHAR (5))
BEGIN ATOMIC
CASE result
WHEN '1-0' THEN
SET winner='white';
WHEN '0-1 ' THEN
SET winner='black';
ELSE
SET winner='draw';
END CASE END;
```

После создания хранимой процедуры ее можно вызвать с помощью команды CALL, например:

```
CALL MatchScore ('Kasparov', 'Karpov', '1-0', winner);
```

Три первых аргумента являются входными параметрами, которые передаются процедуре MatchScore. Четвертый аргумент является выходным параметром, который процедура Match-Score использует для возврата своих результатов в вызывающую программу. В приведенном примере она возвращает значение 'white'.

## Хранимые функции

**Хранимые функции** (stored functions) аналогичны хранимым процедурам. Вместе они называются **хранимыми подпрограммами.** Они имеют ряд отличий, например, они отличаются по способу их

вызова. Хранимые процедуры вызываются в операторе CALL, а хранимые функции могут являться аргументом оператора SQL. Ниже представлен пример определения функции, сопровождаемый примером обращения к этой функции.

```
CREATE FUNCTION PurchaseHistory (CustID)
RETURNS CHAR VARYING (200)
BEGIN
DECLARE purch CHAR VARYING (200)
DEFAULT '';
FOR x AS SELECT *
FROM transaction t
WHERE t.customerID = CustID
DO
IF x.description <> ''
THEN SET purch=purch || ',';
END IF;
SET purch=purch || x.description;
END FOR
RETURN purch;
END;
```

Функция, заданная подобным образом, сводит воедино описания всех покупок, произведенных клиентом с определенным идентификационным номером, выбранным из таблицы TRANSACTIONS. Следующая команда, содержащая вызов функции PurchaseHistory, заносит в таблицу информацию о всех покупках, сделанных клиентом с идентификационным номером 314259:

```
SET customerID=3142 59;
UPDATE customer
SET history=PurchaseHistory (customerlD)
WHERE customerID=314259;
```

## Полномочия

В главе 13 рассматривались различные полномочия, предоставляемые пользователям. Владелец базы данных может предоставить другим пользователям следующие полномочия.

- На удаление записи из таблицы.
- На вставку записи в таблицу.
- На обновления записи в таблице.
- На создание ссылок на поля таблицы.
- На использование домена.

SQL/PSM добавляет к уже существующим еще один вид полномочий – полномочия на выполнение. Ниже приведены два примера.

```
GRANT EXECUTE on MatchScore to TournamentDirector;
GRANT EXECUTE on PurchaseHistory to SalesManager;
```

Эти команды предоставляют возможность руководителю шахматного турнира вызывать процедуру MatchScore, а менеджеру по продажам компании – вызывать функцию Purchase-History. Пользователи, которые не имеют полномочий на выполнение подпрограммы, не могут ее использовать.

## Хранимые модули

**Хранимые модули** (stored modules) могут содержать множество подпрограмм, т.е. процедур и (или) функций SQL. Каждый пользователь с полномочиями на выполнение модуля имеет доступ ко всем подпрограммам этого модуля. Полномочия на выполнение отдельных подпрограмм модуля предоставляться не могут. Ниже приведен пример хранимого модуля.

```
CREATE MODULE mod1
PROCEDURE MatchScore
(IN white CHAR (20),
IN black CHAR (20),
IN result CHAR (3),
OUT winner CHAR (5))
BEGIN ATOMIC
CASE result
WHEN '1-0' THEN
SET winner = 'white';
WHEN '1-0' THEN
SET winner = 'black';
ELSE
SET winner = 'draw';
END CASE
END;
FUNCTION PurchaseHistory (CustID)
RETURNS CHAR VARYING (200)
BEGIN
DECLARE purch CHAR VARYING (200)
DEFAULT '';
FOR X AS SELECT *
FROM transaction t
WHERE t.cuStomerlD = CustID
DO
IF x.description <> ''
THEN SET purch = purch | | '. ';
END IF;
SET purch = purch || x.description;
END FOR
RETURN purch;
END;
END MODULE;
```

Подпрограммы данного модуля никак не связаны между собой. Вообще говоря, подпрограммы можно разнести по разным модулям или собрать в один модуль, независимо от того, есть у них что-нибудь общее или нет.

# Обработка ошибок

В этой главе…

- Подача сигнала об ошибке
- Переход к коду обработки ошибок
- Ограничение, вызвавшее ошибку
- Ошибка какой СУБД произошла

---

Правда, было бы замечательно, чтобы каждое написанное вами приложение все время работало прекрасно? Еще бы! А если еще и выиграть 57 миллионов долларов в лотерею, что проводится в штате Орегон, то вообще было бы все круто! К сожалению, вероятность первого события не превышает вероятности второго. Те или иные состояния ошибки случаются неизбежно, поэтому полезно знать их причины. В SQL:2003 механизмом, передающим информацию об ошибке, является **параметр состояния** (или переменная базового языка) SQLSTATE (состояние SQL). С помощью этой информации можно выполнить те или иные действия, которые помогают исправить ошибку.

Скажем, предложение WHENEVER (как только) дает возможность выполнять заранее заготовленное действие, как только возникает некоторая ситуация, например, когда у параметра SQLSTATE появляется ненулевое значение. Кроме того, подробную информацию о состоянии только что выполненного оператора SQL можно найти в области диагностики. В данной главе рассказывается об этих полезных средствах и о том, как их использовать.

## SQLSTATE

Параметр SQLSTATE позволяет определить множество нештатных ситуаций. Он представляет собой строку из пяти символов, в которой могут находиться буквы в верхнем регистре от A до Z и цифры от 0 до 9. Эта строка делится на две группы – двухсимвольный код класса и трехсимвольный код подкласса.

В стандарте SQL:2003 определены все коды класса, начинающиеся с букв от A до H или с цифр от 0 до 4. Поэтому любой такой код во всех реализациях означает одно и то же. Что касается кодов классов, которые начинаются с букв от I до Z или с цифр от 5 до 9, то их значение определяется конкретными реализациями СУБД. Дело в том, что спецификация SQL не может предусмотреть все ситуации, которые могут произойти в каждой реализации. Впрочем, если говорить откровенно, разработчикам нужно пореже использовать эти нестандартные коды классов. Нестандартные коды усложняют переход с одной СУБД на другую. Лучше, конечно, вообще обойтись стандартными кодами, а нестандартные использовать только в виде исключения.

Если в параметре SQLSTATE код класса равен 00, оператор завершился успешно. А код класса, равный 01, означает уже другое: хотя оператор и завершился успешно, но вывел предупреждение. Если нет данных, значение этого кода становится равным 02. Любое другое значение кода класса, находящееся в параметре SQLSTATE, означает, что выполнение опера-гора не было успешным.

Так как после каждой операции SQL параметр SQLSTATE обновляется, проверять его можно после выполнения каждого оператора. Если значение параметра SQLSTATE – 00000 (успешное завершение), то можно приступить к выполнению следующей запланированной операции. Ну а если в нем находится что-то другое, то для обработки ситуации, возможно, придется отклониться от той обычной последовательности выполнения кода. Какое из нескольких возможных действий следует выполнить – зависит от находящихся в параметре SQLSTATE значений кодов класса и подкласса.

Чтобы параметр состояния SQLSTATE можно было использовать в программе, написанной на модульном языке (о таких программах см. в главе 15), ссылку на этот параметр надо поместить в определении процедуры. Как это делается, показано в следующем примере с процедурой NUTRIENT (питательность), которая работает с таблицей FOODS (продукты питания), имеющей столбцы Foodname (название продукта), Calories (калории), Protein (белки), Fat (жиры) и Carbohidrate (углеводы):

```
PROCEDURE NUTRIENT
(SQLSTATE,:foodname CHAR (20),:calories SMALLINT,
:rprotein DECIMAL (5.1),:fat DECIMAL (5.1),
:carbo DECIMAL (5.1))
INSERT INTO FOODS
(Foodname, Calories, Protein, Fat, Carbohidrate)
VALUES
(:foodname, rcalories,:protein,:fat,:carbo);
```

В нужном месте программы, написанной на процедурном языке, вы можете присвоить параметрам определенные значения (возможно, используя пользовательский ввод), а затем вызвать саму процедуру. Синтаксис этой операции в разных языках разный и выглядит примерно так:

```
foodname = "Okra, boiled";
calories = 29;
protein = 2.0;
fat = 0.3;
carbo = 6.0,-
NUTRIENT(state, foodname, calories, protein, fat, carbo);
```

Значение SQLSTATE возвращается переменной **state** (состояние). Ваша программа может проверять эту переменную и, в зависимости от ее значения, выполнять то или иное действие. *(Для тех, кто не знает, окра – это такое растение, которое еще называется бамией. В пищу идут недозрелые стручки, которые варят, и созревшие семена, из которых делают суррогат кофе. – Примеч. пер.)*

## Предложение WHENEVER

А зачем, собственно, знать, что оператор SQL не выполнился успешно, если с этим уже ничего не поделаешь? Дело здесь вот в чем. Если произошла ошибка, то нельзя, чтобы приложение выполнялось так, будто ничего не случилось. Нужно иметь возможность узнать об ошибке и затем что-то предпринять, чтобы ее исправить. А если исправить ее невозможно, нужно сообщить об ошибке пользователю и корректно завершить приложение. В SQL для обработки исключительных ситуаций, или, как их еще называют, исключений, имеется такой механизм, как предложение WHENEVER.

Предложение WHENEVER фактически является объявлением, поэтому его помещают в разделе объявлений SQL-приложения перед выполняемым кодом SQL. У этого предложения такой синтаксис:

```
WHENEVER состояние действие;
```

Состояниями могут быть SQLERROR (ошибка SQL) или NOT FOUND (не найден). А действием – CONTINUE (продолжать) или GOTO адрес (перейти по адресу). Если код класса в параметре SQLSTATE не равен 00, 01 или 02, то возникает состояние SQLERROR. А если параметр SQLSTATE равен 02000, то возникает состояние NOT FOUND.

Если действием является CONTINUE, то выполнение кода происходит по обычному сценарию. Но когда вместо CONTINUE задано GOTO адрес (или GO TO адрес), точка выполнения перемещается по указанному адресу в программе. Адресом перехода может быть и условное выражение, которое

проверяет значение параметра SQLSTATE и, в зависимости от результатов проверки, переводит выполнение по требуемому адресу. Вот несколько примеров предложения WHENEVER:

```
WHENEVER SQLERROR GO TO error_trap;
```

…или:

```
WHENEVER NOT FOUND CONTINUE;
```

**GO TO** – это макрокоманда. Реализация, точнее препроцессор встроенного языка, вставляет после каждого оператора EXEC SQL следующую проверку:

```
IF SQLSTATE <> '00000'
AND SQLSTATE <> '00001'
AND SQLSTATE <> '00002'
THEN GOTO error_trap;
```

Опция **CONTINUE** означает отсутствие действий, т.е. действие "игнорируй ошибку".

## Область диагностики

Хотя параметр SQLSTATE в состоянии дать информацию о том, почему неудачно завершился некоторый оператор, но такая информация все же неполная. Поэтому стандарт SQL:2003, кроме того, еще дает возможность перехватывать дополнительную информацию о состоянии и хранить ее в области диагностики. Многократное определение области диагностики работает по принципу стека LIFO (last-in-first-out – последним вошел, первым вышел). Дополнительная информация о состоянии может быть особенно полезной тогда, когда при выполнении единственного оператора SQL появилось множество ошибок. Параметр SQLSTATE сообщает только об одной из них, а область диагностики может рассказать сразу о множестве ошибок (а возможно, и обо всех).

**Область диагностики** – это структура данных СУБД, состоящая из двух компонентов.

**Заголовок**. В нем находится общая информация о последнем выполнявшемся операторе SQL.

**Информационная область**. В ней находится подробная информация о каждом коде (ошибка, предупреждение или успешное выполнение), который был сгенерирован в результате выполнения оператора.

## Область заголовка диагностики

В главе 14 мы изучали оператор SET TRANSACTION (задать транзакцию). В нем можно указать DIAGNOSTICS SIZE (размер диагностики). Этот размер является количеством информационных областей, выделяемых для хранения информации о состоянии. Если в операторе SET TRANSACTION не будет предложения DIAGNOSTICS SIZE, то СУБД выделит количество информационных областей, которое в ней установлено по умолчанию.

Заголовок области состоит из восьми элементов, которые приведены в табл. 20.1.

**Таблица 20.1**. Область заголовка диагностики.

| Поля | Тип данных |
|---|---|
| **number** (количество) | Точный числовой, масштаб 0 |
| **row_count** (количество строк) | Точный числовой, масштаб 0 |

| Поля | Тип данных |
|---|---|
| **command_functton** (командная функция) | Символьная строка переменного размера, не менее 128 символов |
| **command_function_code** (код командной функции) | Точный числовой, масштаб 0 |
| **more** (больше) | Символьная строка, длина 1 |
| **transact ions_commited** (фиксированные транзакции) | Точный числовой, масштаб 0 |
| **transactions_rolled_back** (откатанные транзакции) | Точный числовой, масштаб 0 |
| **transactions_active** (состояние транзакции) | Точный числовой, масштаб 0 |

Ниже эти элементы описаны более детально.

- В поле NUMBER хранится количество областей, заполненных диагностической информацией о текущей операции.
- В поле ROW_COUNT содержится количество строк, задействованных при выполнении оператора SQL INSERT, UPDATE или DELETE.
- Поле COMMAND_FUNCTION описывает только что выполненный динамический оператор SQL (если последний выполненный оператор SQL действительно был динамическим).
- Поле COMMAND_FUNCTION_CODE отображает номер кода для только что выполненного динамического оператора SQL (если последний выполненный оператор SQL действительно был динамическим). Каждая динамическая функция имеет соответствующий код.
- Поле MORE содержит одно из значений: 'Y' (да) или 'N' (нет). Значение 'Y' указывает на то, что записей состояния больше, чем может вместить область диагностики. Значение 'N' означает, что все сгенерированные записи состояния представлены в области диагностики. Вы можете получить необходимое количество записей, увеличив его, используя оператор SET TRANSACTIONS, однако эта возможность зависит от используемой вами реализации.
- Поле TRANSACTIONS_COMMITED содержит количество совершенных транзакций.
- Поле TRANSACTIONS_ROLLED_BACK содержит количество транзакций, которые были откатаны.
- Поле TRANSACTIONS_ACTIVE содержит значение 'Г, если транзакция в настоящее время активна, и значение '0' в противном случае. Транзакция считается допустимой для активизации, если открыт курсор или СУБД находится в ожидании отложенного параметра.

## Информационная область диагностики

В информационных областях хранятся данные по каждой отдельной ошибке, предупреждению или состоянию успешного завершения. Каждая информационная область (табл. 20.2) состоит из 26 элементов.

**Таблица 20.2**. Информационная область диагностики.

| Элемент | Тип данных |
|---|---|
| CONDITION_NUMBER (номер состояния) | Точный числовой, масштаб 0 |
| RETURNED_SQLSTATE (значение SQLSTATE) | Символьная строка, 6 символов |
| MESSAGE_TEXT (текст сообщения) | Символьная строка переменного размера, не менее 128 символов |

| Элемент | Тип данных |
|---|---|
| MESSAGE_LENGTH (длина сообщения) | Точный числовой, масштаб 0 |
| MESSAGE OCTET LENGTH (длина сообщения в октетах) | Точный числовой, масштаб 0 |
| CLASS_ORIGIN (источник класса) | Символьная строка переменного размера, не менее 128 символов |
| SUBCLASS_ORIGIN (источник подкласса) | Символьная строка переменного размера, не менее 128 символов |
| CONNECTION_NAME (имя соединения) | Символьная строка переменного размера, не менее 128 символов |
| SERVER_NAME (имя сервера) | Символьная строка переменного размера, не менее 128 символов |
| CONTRAINT_CATALOG (каталог ограничения) | Символьная строка переменного размера, не менее 128 символов |
| CONTRAINT_SCHEMA (схема ограничения) | Символьная строка переменного размера, не менее 128 символов |
| CONSTRAINT_NAME (имя ограничения) | Символьная строка переменного размера, не менее 128 символов |
| ENVIRONMENT_NAME (имя среды) | Символьная строка переменного размера, не менее 128 символов |
| CATALOG_NAME (имя каталога) | Символьная строка переменного размера, не менее 128 символов |
| SCHEMA_NAME (имя схемы) | Символьная строка переменного размера, не менее 128 символов |
| TABLE_NAME (имя таблицы) | Символьная строка переменного размера, не менее 128 символов |
| COLUMN_NAME (имя столбца) | Символьная строка переменного размера, не менее 128 символов |
| CURSOR_NAME (имя курсора) | Символьная строка переменного размера, не менее 128 символов |
| CONDITION_IDENTIFIER (идентификатор состояния) | Символьная строка переменного размера, не менее 128 символов |
| PARAMETR_NAME (имя параметра) | Символьная строка переменного размера, не менее 128 символов |
| ROUTINE_CATALOG (каталог программы) | Символьная строка переменного размера, не менее 128 символов |
| ROUTINE_SCHEMA (схема программы) | Символьная строка переменного размера, не менее 128 символов |
| ROUTINE_NAME (имя программы) | Символьная строка переменного размера, не менее 128 символов |
| SPECIFIC_NAME (специфическое имя) | Символьная строка переменного размера, не менее 128 символов |
| TRIGGER_CATALOG (каталог триггера) | Символьная строка переменного размера, не менее 128 символов |
| TRIGGER_SCHEMA (схема триггера) | Символьная строка переменного размера, не менее 128 символов |
| TRIGGER_NAME (имя триггера) | Символьная строка переменного размера, не менее 128 символов |

В элементе CONDITION_NUMBER содержится порядковый номер информационной области. Если оператор создает пять элементов состояния, которые заполняют пять информационных областей, то значение CONDITION_NUMBER для пятой такой области будет равно пяти. Чтобы получить доступ к конкретной информационной области, используйте оператор GET DIAGNOSTICS (получить диагностику) вместе с требуемым значением CONDITTONJSrUMBER. (Об операторе GET DIAGNOSTICS рассказывается ниже, в разделе "Что означает информация, возвращаемая SQLSTATE".) А в элементе RETURNED_SQLSTATE находится значение SQLSTATE, соответствующее данным этой информационной области.

Элемент CLASS_ORIGIN сообщает, откуда взято значение для кода класса, возвращаемое в параметре SQLSTATE. Если значение определено стандартом SQL, то элемент CLASS_ORIGIN равен TSO 9075'. А если оно определено реализацией СУБД, то в элементе CLASS_ORIGIN находится строка, в которой указана СУБД-источник. Элемент SUBCLASS_ORIGIN, в свою очередь, сообщает источник значения для кода подкласса, которое возвращено в параметре SQLSTATE.

Значение, находящееся в элементе CLASS_ORIGIN, является достаточно важным. Значение SQLSTATE, равное, например, '22012', относится к стандартным значениям этого параметра. Поэтому вам известно, что оно означает одно и то же во всех реализациях SQL. Однако если значение SQLSTATE равно '22500', первые два символа находятся в стандартном диапазоне и указывают на исключительную ситуацию, связанную с отсутствием данных, а последние три символа уже находятся в диапазоне, определяемом реализацией. Ну а если значение SQLSTATE равно '90001', то это значение полностью находится в диапазоне, определяемом реализацией. Одни и те же значения SQLSTATE, находящиеся в таком диапазоне, могут в разных реализациях означать совершенно различные понятия.

А где же найти описание '22500' или '90001' Для этого надо взглянуть в документацию пользователя СУБД. А какой именно СУБД? Ведь с помощью оператора CONNECT можно соединиться сразу с несколькими. Чтобы узнать, какой из них является источником ошибки, взгляните на элементы CLASS_ORIGIN и SUBCLASS_ORIGIN. В них находятся значения, которые определяют каждое приложение. Проверяя эти значения, можно определить, к какой СУБД относятся значения SQLSTATE. Значения, находящиеся в элементах CLASS_ORIGIN и SUBCLASS_ORIGIN, также определяются реализацией, но обычно содержат название компании-разработчика СУБД.

Если ошибка является нарушением ограничения, это ограничение можно определить с помощью элементов CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA и CONSTRAINT_NAME.

## Пример нарушения ограничения

Из всей информации, предоставляемой оператором GET DIAGNOSTICS, самой важной является информация о нарушении ограничения. Рассмотрим пример. Создана таблица EMPLOYEE (сотрудник) со столбцами ID (идентификатор) и Salary (зарплата):

```
CREATE TABLE EMPLOYEE
(ID CHAR(5) CONSTRAINT EmpPK PRIMARY KEY,
Salary DEC(8.2) CONSTRAINT EmpSal CHECK Salary > 0,
Dept CHAR(5) CONSTAINT EmpDept
REFERENCES DEPARTMENT);
```

Кроме того, имеется таблица DEPARTMENT (отдел) со столбцами DeptNo (номер отдела) и Budget (бюджет):

```
CREATE TABLE DEPARTMENT
(DeptNo CHAR(5),
Budget DEC(12.2) CONSTRAINT DeptBudget
CHECK(Budget >= SELECT SUM(Salary) FROM EMPLOYEE
```

```
WHERE EMPLOYEE.Dept=DEPARTMENT.DeptNo),
…);
```

А теперь посмотрите на следующий оператор INSERT:

```
INSERT INTO EMPLOYEE VALUES(:ID_VAR,:SAL__VAR,:DEPT_VAR);
```

Предположим, что вы получили значение SQLSTATE, равное '23000'. Посмотрев в документацию по SQL, вы видите, что этому значению соответствует описание "нарушение ограничения целостности". Это означает, что имеет место одна из следующих ситуаций.

- **Значение IDJVAR повторяет уже существующее значение ID**, т.е. нарушено ограничение PRIMARY KEY.
- **Значение SALJVAR отрицательное** – иначе говоря, нарушено ограничение CHECK на столбце Salary.
- **Значение DEPT_VAR не является правильным ключом, соответствующим какой-либо из строк таблицы DEPARTMENT**, так что нарушено ограничение REFERENCES в столбце Dept.
- **Значение SAL_VAR настолько большое, что у сотрудников, работающих в отделе, для которого вводятся новые данные, сумма окладов превышает значение Budget для этого отдела**. На этот раз имеется нарушение ограничения CHECK в столбце Budget таблицы DEPARTMENT. (Вспомните, что при изменении базы данных должны проверяться все связанные с ним ограничения, а не только те, которые определены в изменяемых таблицах.)

Обычно, чтобы узнать причины невыполнения оператора INSERT, приходится проводить большое количество тестов. Но на этот раз все, что нужно, можно узнать с помощью команды GET DIAGNOSTICS:

```
DECLARE ConstNameVar CHAR(18);
GET DIAGNOSTICS EXCEPTION 1
ConstNameVar = CONSTRAINT_NAME;
```

И если значение SQLSTATE равно '23000', то этот оператор GET DIAGNOSTICS присваивает переменной ConstNameVar одно из следующих значений: 'EmpPK', 'EmpSal', 'EmpDept' или 'DeptBudget'. Обратите внимание, для того, чтобы однозначно определить ограничение CONSTRAINT_NAME, еще могут понадобиться значения элементов CONSTRAINT_SCHEMA и CONSTRAINT_CATALOG.

## Ввод новых ограничений в уже созданную таблицу

Особенно полезен оператор GET DIAGNOSTICS для определения нарушаемых ограничений при изменении исходных таблиц. Например, в таблицу с помощью оператора ALTER TABLE вводят ограничения, которых не было при написании программы:

```
ALTER TABLE EMPLOYEE
ADD CONSTRAINT SalLimit CHECK(Salary < 200000);
```

Теперь, когда вы вставите данные в таблицу EMPLOYEE или обновите в ней столбец Salary и значение в этом столбце превысит 200000, значение SQLSTATE станет равно '23000'. В таких случаях можно запрограммировать вывод каких-либо полезных сообщений. Например, такого рода: "Неправильное выполнение оператора INSERT: нарушение ограничения SalLimit".

## Что означает информация, возвращаемая параметром SQLSTATE

В элементах CONNECTTON_NAME и ENVIRONMENTJVAME хранятся названия соединения и той среды, с которой было установлено соединение во время выполнения оператора SQL.

Если информация из параметра SQLSTATE относится к работе с таблицей, то эту таблицу определяют элементы CATALOG_NAME, SCHEMA_NAME и TABLE_NAME. Если появление ошибки как-то связано со столбцом таблицы, его имя помещается в элемент COLUMN_NAME. Если нештатная ситуация имеет отношение к курсору, его имя будет находиться в элементе CURSOR_NAME.

Иногда СУБД, чтобы объяснить ситуацию, создает строку текста на каком-либо языке, например английском. Такого рода информация содержится в элементе MESSAGE_TEXT. Его содержимое определяется не стандартом SQL.2003, а реализацией. Если в элементе MESSAGEJTEXT имеется сообщение, его длина в символах записывается в элемент MESSAGE_LENGTH, а длина в октетах – в элемент MESSAGE_OCTET_LENGTH. У сообщения, состоящего из обычных ASCII-символов, значения MESSAGE_LENGTH и MESSAGE_OCTET_LENGTH равны между собой. А если сообщение составлено на китайском, японском или любом другом языке, в котором для выражения символа требуется больше одного октета, то значения MESSAGE_LENGTH и MESSAGE_OCTET_LENGTH будут разными.

Чтобы получить диагностическую информацию из области заголовка, воспользуйтесь выражением:

```
GET DIAGNOSTICS переменная1 = элемент1 [, переменная2 = елемент2]…;
```

Где переменная_n – это параметр или базовая переменная; элемент_n – любое из следующих ключевых слов: NUMBER, MORE, COMMAND_FUNCTION, DYNAMIC_FUNCTION или ROW_COUNT.

А чтобы получить диагностическую информацию из информационной области, используют следующий синтаксис:

```
GET DIAGNOSTICS EXCEPTION номер-состояния
переменная1 = элемент1 [, переменная2 = элемент2]…;
```

Где переменная_n – это параметр или базовая переменная; элемент_n – любое из семнадцати ключевых слов элементов информационной области.

Эти ключевые слова приведены в табл. 20.2. И наконец, номер состояния – это значение элемента CONDITION_NUMBER информационной области.

## Обработка исключений

Если параметр состояния SQLSTATE не равен 00000, 00001 или 00002, значит, возникла исключительная ситуация, которую, возможно, нужно будет обработать.

- Обработка может состоять в возвращении управления родительской процедуре, в которой была вызвана подпроцедура – источник исключения.
- Второй способ обработки состоит в использовании предложения WHENEVER для перехода на нужную процедуру обработки или выполнения какого-либо другого действия.
- Третий способ заключается в том, чтобы обрабатывать ситуацию прямо на месте, используя для этого составной оператор SQL. Составным называется такой SQL-оператор, который состоит из более простых операторов SQL, находящихся между ключевыми словами BEGIN (начало) и END (конец).

Ниже приведен пример составного оператора, выполняющего обработку исключительных ситуаций.

```
BEGIN
DECLARE ValueOutOfRange EXCEPTION FOR SQLSTATE '73003';
INSERT INTO FOODS
(Calories)
VALUES
(:cal)
SIGNAL ValueOutOfRange;
MESSAGE 'Обрабатывается новое значение количества калорий'
EXCEPTION
WHEN ValueOutOfRange THEN
MESSAGE 'Обработка ошибки выхода количества калорий за допустимый диапазон';
WHEN OTHERS THEN
RESIGNAL;
END
```

С помощью одного или нескольких объявлений можно задать имена для всех возможных значений параметра SQLSTATE. Одним из операторов, которые могут вызвать исключительную ситуацию, является INSERT. Если значение:cal превысит максимальное значение элемента данных типа SMALLINT, параметру SQLSTATE будет присвоено значение '73003'. Сигнал о возникновении исключительной ситуации подает оператор SIGNAL (сигнализировать). Этот оператор очищает область диагностики, а также присваивает полю RETURNED_SQLSTATE этой области значение параметра SQLSTATE. Если исключения не было, выполняется обычная последовательность операторов, роль которой в нашем примере играет MESSAGE 'Обрабатывается новое значение количества калорий1. Если же это исключение все же имело место, то обычная последовательность пропускается и выполняется оператор EXCEPTION.

Как только возникает исключение **ValueOutOfRange** (значение вне диапазона), выполняется последовательность операторов, представленных в нашем примере одним – MESSAGE 'Обработка ошибки выхода количества калорий за допустимый диапазон'. Когда возникнет какое-либо другое исключение, выполняется оператор RESIGNAL (сигнализировать повторно). Он просто передает управление вызывающей родительской процедуре. В этой процедуре, возможно, есть дополнительный код обработки ошибок, с помощью которого можно обрабатывать исключительные ситуации, не связанные с выходом за пределы диапазона.

# Десять самых распространенных ошибок

В этой главе…

- Мнение, что клиенты знают, чего хотят
- Игнорирование масштаба проекта
- Учет только технических факторов
- Отсутствие обратной связи с пользователями
- Применение только своих любимых сред разработки
- Использование только своих любимых системных архитектур
- Проектирование таблиц базы данных отдельно друг от друга
- Отсутствие просмотра проекта в целом
- Отсутствие бета-тестирования
- Отказ от создания документации

---

Если уж вы читаете эту книгу, то вопросы, связанные с созданием реляционных баз данных, вас обязательно интересуют. Скажем прямо, никто не изучает SQL только ради удовольствия. Этот язык используется для создания приложений, работающих с базами данных, но перед тем как появится возможность создать какое-нибудь из этих приложений, для него уже нужно создать базу. К сожалению, многие проекты терпят неудачу еще до того, как для приложения напишут первую строку кода. Если в основе базы данных заложены неправильные принципы, то как бы хорошо ни было написано приложение, оно все равно обречено на неудачу. В последующих разделах приводятся десять самых распространенных ошибок, допускаемых при создании баз данных. Этих ошибок следует избегать.

## Мнение, что клиенты знают, чего хотят

Обычно клиенты приглашают вас спроектировать для них базу данных, когда сталкиваются с трудностями, а имеющиеся у них методы не работают. Клиенты часто уверены, что им известно, в чем состоит проблема и как ее решать. По их мнению, все, что они должны сделать, – это рассказать, что именно вам следует делать.

Однако давать клиенту в точности то, что он просит, – это рецепт заведомой неудачи. Большинство пользователей и их менеджеров не владеют знаниями и квалификацией, нужными для точного определения проблемы, поэтому у них нет шанса найти самое лучшее решение.

Перед вами стоит задача – тактично убедить клиента, что вы эксперт по анализу систем и что вам необходимо провести такой анализ, чтобы вскрыть реальную причину проблемы. Ведь реальная причина обычно скрыта за более очевидными симптомами.

## Игнорирование масштаба проекта

В начале разработки проекта ваш клиент рассказывает, что именно он ждет от нового приложения. К сожалению, клиент почти всегда забывает что-то сказать, причем его забывчивость чем-то одним, как правило, не ограничивается. В процессе работы неожиданно возникают и добавляются в проект новые требования. И если вам за проект установлена не почасовая оплата, то этот рост его масштабов может привести к тому, что когда-то прибыльный проект станет убыточным. Все, что вы обязуетесь сделать, должно быть зафиксировано в письменном виде еще до начала работы над проектом. А что касается новых требований, появляющихся во время работы над проектом, то их надо подкреплять дополнительным временем и бюджетом.

## Учет только технических факторов

Разработчики приложений часто рассматривают свои будущие проекты только с точки зрения технической выполнимости и потом оценивают требуемые усилия и время исходя из одного этого фактора. Однако на проект могут оказывать влияние и такие вещи, как максимальная стоимость, имеющиеся ресурсы, требования расписания и организационная политика. На практике они могут превратить технически выполнимый проект в сплошной кошмар. Перед тем как начинать разработку проекта, изучите все сопутствующие ему факторы. Возможно, вы решите, что нет смысла заниматься этим проектом, и намного лучше прийти к этому заключению в самом начале, чем тогда, когда на проект уже затрачены большие усилия.

## Отсутствие обратной связи с клиентами

Вначале вы, возможно, будете склонны слушать только нанявших вас менеджеров. Сами же пользователи не будут иметь ни малейшего влияния на вашу работу. С другой стороны, может существовать уважительная причина также игнорировать и менеджеров. Ведь они обычно даже представления не имеют о том, что в действительности нужно пользователям. Из этого автоматически не следует, что вы лучше самих клиентов знаете об их реальных нуждах. Конечно, сотрудники, занятые вводом данных, не очень значимы в организации, а многие менеджеры имеют лишь смутное представление о некоторых вопросах, за которые отвечают. Но если вы изолируете себя от любой из этих двух групп, то тогда наверняка создадите систему, которая не решит ничьих проблем. И пусть приложение будет прекрасно работать, но если оно решает не те проблемы, что нужно, такое приложение ни для кого не будет представлять ни малейшей ценности.

## Применение только своих любимых сред разработки

Вы, возможно, потратили месяцы или даже годы на то, чтобы стать специалистом в использовании конкретной СУБД или среды разработки приложений. Но ваша любимая среда, не важно какая, имеет как достоинства, так и недостатки. Время от времени вам будут попадаться задачи-разработки, которые предъявляют высокие требования именно к тем областям, в которых ваша любимая среда разработки находится отнюдь не на высоте. Так что вместо того, чтобы придумывать не самое лучшее решение, лучше остановиться и подумать. У вас есть два варианта. Первый состоит в том, чтобы освоить более подходящий инструмент, а затем использовать его. Второй вариант – это чистосердечно заявить своим клиентам, что их задачу лучше решать с помощью инструмента, в котором вы явно не эксперт. Затем предложите им нанять того, кто сможет продуктивно работать с этим инструментом. Такое профессиональное поведение заставит клиентов еще больше вас уважать. (Но, к сожалению, если вы работаете не на себя, а на фирму, то тогда есть опасность, что вас могут просто сократить.)

## Использование только своих любимых системных архитектур

Никто не может быть экспертом по всем вопросам. СУБД, работающие в среде дистанционной обработки данных, отличаются от тех, которые работают в средах клиент/сервер, средах с разделением ресурсов или распределенных средах. Одна или две системы, в которых вы эксперт, могут не подходить для полученного вами задания. Но все равно нужно выбрать самую лучшую архитектуру, даже если это означает передачу задания кому-либо другому. Лучше совсем не получить задание, чем получить его и сделать такую систему, которая не нужна клиенту.

## Проектирование таблиц базы данных отдельно друг от друга

В результате того, что объекты данных и их связи друг с другом определены неправильно, в базе имеются такие таблицы, благодаря которым в данных постоянно появляются ошибки, и это может свести на нет ценность любых результатов. Чтобы спроектировать добротную базу данных,

необходимо проанализировать общую схему объектов данных и тщательно проследить их связи друг с другом. Обычно существует не менее одного правильного проекта. Необходимо определить, какой из них вам подходит, учитывая при этом потребности вашего клиента, как нынешние, так и будущие.

## Отказ от консультации с другими специалистами

Никто не совершенен. Даже самый лучший проектировщик или разработчик может пропустить важные моменты, которые являются очевидными для любого, кто взглянет на ситуацию с другой точки зрения. И действительно, если вам приходится выставлять свой проект на суд общественности, это вас дисциплинирует и, возможно, помогает избежать многочисленных неприятностей, с которыми в противном случае вы бы наверняка столкнулись. Перед тем как разрабатывать приложение, обязательно покажите проект компетентным профессионалам.

## Отсутствие бета-тестирования

Приложение, работающее с базами данных, чтобы быть полезным, должно быть сложным. В то же время сложное приложение не может не иметь ошибок. Даже если вы будете проверять свое приложение всеми способами, до которых только сможете додуматься, все равно в нем останутся незамеченные сбойные участки. Бета-тестирование – это передача приложения тем людям, которые не разбираются в нем так хорошо, как вы. Вот они-то наверняка столкнутся со всеми неприятностями, которые вам никогда не встретятся по той причине, что вы слишком много знаете о своем приложении. И все ошибки, выявленные бета-тестерами, вам нужно будет исправить, причем еще до того, как продукт станет использоваться официально.

## Отказ от создания документации

Если вы думаете, что ваше приложение настолько совершенно, что за ним не придется смотреть, то сильно ошибаетесь. Единственное в этом мире, в чем можно быть абсолютно уверенным, – это то, что все течет, все изменяется. Это следует учитывать. И если тщательно не документировать, что было сделано и почему сделано именно так, а не иначе, то через шесть месяцев вы уже не сможете этого вспомнить. Кроме того, если вы перейдете в другой отдел или получите огромный выигрыш в лотерею и уволитесь, не оставив документацию по своему проекту, ваш преемник наверняка не сможет внести никаких изменений в ваше творение, чтобы оно соответствовало новым требованиям. И тогда ему, возможно, придется выбросить приложение и начать все с самого начала. Документируйте свою работу не просто в достаточной степени, а с большим запасом. Делайте документацию более подробной, чем это нужно с точки зрения здравого смысла. И если через шесть или восемь месяцев вы вернетесь к проекту, то будете рады, что вся нужная информация задокументирована.

# Десять советов по извлечению данных

В этой главе…

- Проверка структуры базы данных
- Использование тестовых баз данных
- Тщательная проверка любого запроса с оператором **join**
- Проверка запросов с подвыборками
- Использование предложения **group by** вместе с итоговыми функциями
- Внимательное отношение к ограничениям из предложения **group by**
- Использование круглых скобок в выражениях
- Защита базы данных с помощью управления полномочиями
- Регулярное резервное копирование базы данных
- Предвидение ошибок и их обработка

---

База данных может быть настоящим виртуальным кладом с информационными сокровищами, но, как и сокровища, принадлежавшие много лет назад карибским пиратам, то, что вам действительно нужно, скорее всего, зарыто и спрятано далеко от людских глаз. Чтобы откопать эту скрытую информацию, вам потребуется специальный инструмент – оператор SQL SELECT. Но даже если вы четко знаете, что вам действительно нужно, сформулировать запрос может оказаться достаточно трудным делом. Если в своем запросе вы чуть отклонитесь в сторону, результаты будут неправильными, пусть даже очень близкими к ожидаемым. Причем настолько близкими, что могут ввести вас в заблуждение. Поэтому, чтобы иметь меньше шансов быть обманутым, придерживайтесь следующих десяти принципов.

## Проверяйте структуру базы данных

Если полученные из базы данные не кажутся вам разумными, проверьте структуру этой базы. Существует много баз данных с неудачной структурой, и если вы работаете с такой базой, то вначале исправьте ее структуру, а лишь затем используйте другое "лекарство". Помните – хорошая структура является предварительным условием сохранения целостности данных.

## Испытывайте запросы на тестовой базе данных

Создайте тестовую базу данных с такой же структурой, как и база, с которой вы работаете, но имеющую в своих таблицах небольшое количество строк, которые взяты для примера. В этих строках должны быть такие данные, чтобы можно было знать наперед, каким должен быть результат ваших запросов. Отправляйте запрос в тестовую базу на выполнение и затем смотрите, соответствуют ли его результаты тем, которые вы ожидали. Если не соответствуют, то, возможно, запрос нужно будет переписать. Но если запрос очевидно правильный, то тогда, видимо, придется переделывать структуру базы данных.

Создайте несколько наборов тестовых данных, в которых обязательно должны быть "особые случаи", например, такие, как пустые таблицы или значения, взятые с крайних точек допустимых диапазонов. Пытайтесь придумать самые невероятные случаи, а затем проверьте, правильно ли ведет себя система. В ходе этой проверки нетипичных случаев вас, возможно, осенит идея и придет решение какой-либо из более обычных проблем.

## Дважды проверяйте запросы, имеющие операторы JOIN

Общеизвестно, что операторы JOIN наглядными никак не назовешь. И если какое-либо из них находится в вашем запросе, то, перед тем, как добавлять в запрос какие-либо предложения WHERE или другие усложняющие компоненты, обязательно проверьте, делает ли это предложение то, что вы от него ожидаете.

## Трижды проверяйте запросы с подвыборками

Подзапросы позволяют совмещать данные одной таблицы с данными другой, и их часто используют неправильно. Необходимо проверять, чтобы данные, получаемые внутренним оператором SELECT, были теми, которые нужны внешнему оператору SELECT для получения нужного конечного результата. А если имеется более двух уровней подвыборок, то надо быть еще более осторожным.

## Подводите итоги, используя предложение GROUP BY

Скажем, у вас есть таблица NATIONAL (Национальная бейсбольная лига), содержащая поля с фамилией игрока (поле Player), командой (поле Team) и количеством успешных ударов битой по мячу (поле Homers) для каждого игрока из Национальной лиги. Итоговые данные для всех команд можно получить, если использовать примерно такой запрос:

```
SELECT Team, SUM (Homers)
FROM NATIONAL
GROUP BY Team;
```

В результате его выполнения получается список команд, в котором за каждой из них указано общее количество успешных ударов битой, выполненных всеми ее игроками.

## Внимательно относитесь к ограничениям предложения GROUP BY

Предположим, вам нужен список самых результативных хиттеров (игроков с битой) Национальной лиги. Проанализируйте следующий запрос:

```
SELECT Player, Team, Homers
FROM NATIONAL
WHERE Homers >= 2 0
GROUP BY Team;
```

В большинстве реализаций этот запрос возвращает сообщение об ошибке. Обычно в списке выборке могут появляться только те столбцы, которые применяются для группирования, или столбцы, используемые в итоговой функции. С учетом этого следующий запрос уже вполне работоспособный:

```
SELECT Player, Team, Homers
FROM NATIONAL
WHERE Homers >= 20
GROUP BY Team, Player, Homers;
```

Так как все столбцы, которые вы хотите отобразить, указаны в предложении GROUP BY, то запрос выполнится успешно и выведет те результаты, которые вам нужны. Благодаря этой формулировке полученный список будет сперва отсортирован по полю Team, затем по полю Player и Homers.

## Используйте круглые скобки с ключевыми словами AND, OR и NOT

Когда AND и OR используются вместе, SQL иногда не обрабатывает выражение в том порядке, который вы ожидаете. Чтобы полученный результат был тем, на который вы рассчитываете, используйте круглые скобки. Несколько лишних щелчков на кнопках – это достаточно низкая цена, которую следует заплатить за более достоверные результаты. Кроме того, круглые скобки помогают

сделать так, чтобы ключевое слово NOT применялось именно необходимому к терму или выражению.

## Управляйте полномочиями на получение данных

Многие люди не пользуются средствами безопасности, которые имеются в их СУБД. Они не хотят возиться с этими возможностями, считая, что с их данными не может случиться ничего плохого. Не играйте с огнем, иначе останетесь на пепелище. Установите и поддерживайте безопасность для всех объектов баз данных, имеющих хотя бы малейшую ценность.

## Регулярно выполняйте резервное копирование своих баз данных

Если ваш жесткий диск был разрушен скачком напряжения в сети, огнем, землетрясением или в результате минометного обстрела, то получить данные с этого диска будет нелегко. Поэтому чаще делайте резервные копии и убирайте носители в сухое прохладное место. Что считать безопасным местом, зависит от того, насколько важными являются ваши данные. Им может быть несгораемый сейф, находящийся в той же комнате, что и компьютер, или другое здание. Удобнее всего для этих целей использовать бетонный бункер, выдолбленный в скале и укрепленный настолько, что может выдержать ядерное нападение. Какой уровень безопасности нужен вашим данным – это уже решать вам.

## Предусмотрите достойный выход из ошибочных ситуаций

Делаете ли вы одноразовые запросы с консоли или вставляете запросы в приложение, но время от времени SQL вместо нужных вам результатов выдает сообщение об ошибке. Работая с консоли, вы на основе полученного сообщения можете решить, что же вам делать, а затем выполнить нужное действие. Что касается приложения, то здесь другая ситуация. Пользователь приложения, вероятно, не знает, какое действие является правильным. Поэтому организуйте в своем приложении всеохватную обработку ошибок. Это нужно, чтобы не пропустить ни одной мыслимой ошибки, которая только может возникнуть. Создание кода обработки ошибок потребует больших усилий, но это лучше, чем заставлять пользователя недоуменно смотреть на "зависший" экран.

# Приложение А. Зарезервированные слова SQL:2003.

Зарезервированные слова SQL:2003.

| | | |
|---|---|---|
| ABS | COLLATE | DETERMINISTIC |
| ALL | COLUMN | DISCONNECT |
| ALLOCATE | COMMIT | DISTINCT |
| ALTER | CONDITION | DOUBLE |
| AND | CONNECT | DROP |
| ANY | CONSTRAINT | DYNAMIC |
| ARE | CONVERT | EACH |
| ARRAY | CORR | ELEMENT |
| AS | CORRESPONDING | ELSE |
| ASENSITIVE | COUNT | END |
| ASYMMETRIC | COVAR_POP | END-EXEC |
| AT | COVAR_SAMP | ESCAPE |
| ATOMIC | CREATE | EVERY |
| AUTHORIZATION | CROSS | EXCEPT |
| AVG | CUBE | EXEC |
| BEGIN | CUME_DIST | EXECUTE |
| BETWEEN | CURRENT | EXISTS |
| BIGINT | CURRENT_COLLATION | EXP |
| BINARY | CURRENT_DATE | EXTERNAL |
| BLOB | CURRENT_DEFAULT_TRANSFORM_GROUP | EXTRACT |
| BOOLEAN | CURRENT_PATH | FALSE |
| BOTH | CURRENT_ROLE | FETCH |
| BY | CURRENT_TIME | FILTER |
| CALL | CURRENT_TIMESTAMP | FLOAT |
| CALLED | CURRENT_TRANSFORM_GROUP_FOR_TYPE | FLOOR |
| CARDINALITY | CURRENT_USER | FOR |
| CASCADED | CURSOR | FOREIGN |
| CASE | CYCLE | FREE |
| CAST | DATE | FROM |
| CEIL | DAY | FULL |
| CEILING | DEALLOCATE | FUNCTION |
| CHAR | DEC | FUSION |
| CHAR_LENGTH | DECIMAL | GET |
| CHARACTER | DECLARE | GLOBAL |
| CHARACTER_LENGTH | DAFAULT | GRANT |
| CHECK | DELETE | GROUP |
| CLOB | DENSE_RANC | GROUPING |
| CLOSE | DEREF | HAVING |
| COALESCE | DESCRIBE | HOLD |
| | | |
| HOUR | NONE | REGR_SYY |
| IDENTITY | NORMALIZE | RELEASE |
| IN | NOT | RESULT |
| INDICATOR | NULL | RETURN |
| INNER | NULLIF | RETURNS |
| INOUT | NUMERIC | REVOKE |
| INSENSITIVE | OCTET_LENGHT | RIGHT |

| | | |
|---|---|---|
| INSERT | OF | ROLLBACK |
| INT | OLD | ROLLUP |
| INTEGER | ON | ROW |
| INTERSECT | ONLY | ROW_NUMBER |
| INTERSECTION | OPEN | ROWS |
| INTERVAL | OR | SAVEPOINT |
| INTO | ORDER | SCOPE |
| IS | OUT | SCROLL |
| JOIN | OUTER | SEARCH |
| LANGUAGE | OVER | SECOND |
| LARGE | OVERLAPS | SELECT |
| LATERAL | OVERLAY | SENSITIVE |
| LEADING | PARAMETER | SESSION_USER |
| LEFT | PARTITION | SET |
| LIKE | PERCENT_RANK | SIMILAR |
| LN | PERCENTILE_DISC | SMALLINT |
| LOCAL | POSITION | SOME |
| LOCALTIME | POWER | SPECIFIC |
| LOCALTIMESTAMP | PRECISION | SPECIFICTYPE |
| LOWER | PREPARE | SQL |
| MATCH | PRIMARY | SQLEXEPTION |
| MAX | PROCEDURE | SQLSTATE |
| MEMBER | RANGE | SQLWARNING |
| MERGE | RANK | SQRT |
| METHOD | READS | START |
| MIN | REAL | STATIC |
| MINUTE | RECURSIVE | STDDEV_POP |
| MOD | REF | STRDDEV_SAMP |
| MODIFIERS | REFERENCES | SUBMULTISET |
| MODULE | REFERENCING | SUBSTRING |
| MONTH | REGR_AVGX | SUM |
| MULTISET | REGR_AVGY | SYMMETRIC |
| NATIONAL | REGR_COUNT | SYSTEM |
| NATURAL | REGR_INTERCEPT | SYSTEM_USER |
| NCHAR | REGR_R2 | TABLE |
| NCLOB | REGR_SLOPE | TABLESAMPLE |
| NEW | REGR_SXX | THEN |
| NO | REGR SXY | TIME |
| TIMESTAMP | UNION | VAR_SAMP |
| TIMEZONE_HOUR | UNIQUE | VARCHAR |
| TIMEZONE_MINUTE | UNKNOWN | VARYING |
| TO | UNNEST | WHEN |
| TRAILING | UPDATE | WHENEVER |
| TRANSLATE | UPPER | WHERE |
| TRANSLATION | USER | WIDTH_BUCKET |
| TREAT | USING | WINDOW |
| TRIGGER | VALUE | WITH |
| TRIM | VALUES | WITHING |
| TRUE | VAR POP | WITHOUT |
| | | YEAR |

# Приложение Б. Словарь терминов.

## A

**API (Application Programmer's Interface** – интерфейс прикладного программиста). Стандартное средство взаимодействия приложения и базы данных или другого системного ресурса.

## C

**CODASYL DBTG**. Сетевая модель базы данных. ***Обратите внимание:*** *в данном случае термин "сетевая" относится не к соединениям по сети, а к структуре данных (т.е. подразумевается не иерархическая, а сетевая структура)*.

## I

**Internet**. Всемирная компьютерная сеть.

**IPX/SPX**. Один из протоколов локальных сетей.

## J

**Java**. Платформенно-независимый компилируемый язык, специально предназначенный для разработки Web-приложений.

**JavaScript**. Язык сценариев, который позволяет программировать Web-страницы, написанные с помощью HTML.

**JDBC (Java DataBase Connectivity** – Java-интерфейс взаимодействия с базами данных). Стандартный интерфейс между аплетом или приложением, написанными на Java, и базой данных. Стандарту JDBC предшествовал стандарт ODBC.

## N

**NetBEUI**. Один из протоколов локальных сетей.

## O

**ODBC (Open DataBase Connectivity** – открытый интерфейс доступа к базам данных). Стандартный интерфейс между базой данных и приложением, которое пытается получить доступ к данным базы. ODBC определяется международным (ISO) и американским (ANSI) стандартами.

**Oracle**. Система управления реляционными базами данных, распространяемая компанией Oracle Corporation.

## R

**RAD-инструмент (rapid application development tool** – инструмент быстрой разработки приложений). Патентованная графически ориентированная альтернатива языку SQL. Существует множество таких инструментов.

## S

**SEQUEL**. Подъязык данных, созданный фирмой IBM. Предшественник SQL.

**SQL**. Основной стандартный подъязык данных. Специально предназначен для создания реляционных баз данных, манипуляции и управления ими. Его последней версией является SQL:2OO3.

**SQL, встроенный (SQL, embedded).** SQL-код, встроенный в программу, написанную на базовом языке.

**SQL, динамический (SQL, dynamic)**. Средство создания приложений, которое не требует, чтобы во время компиляции были известны все элементы данных.

**SQL, интерактивный (SQL, interactive)**. Взаимодействие с базой данных в реальном времени.

**SQL/DS**. Система управления реляционными базами данных, распространяемая фирмой IBM Corporation.

## T

**TCP/IP** (**Transmission Control Protocol/Internet Protocol** – протокол управления передачей/протокол Internet). Сетевой протокол, используемый в Intenet и интранет.

## W

**World Wide Web ("Всемирная паутина")**. Часть Internet, для просмотра которой требуется графический пользовательский интерфейс. Доступ к Web выполняется с помощью приложений, называемых **Web-браузерами**, а информация находится на **Web-сереерах.**

## X

**XML (extensible Markup Language – расширяемый язык разметки).** Широко используемый язык разметки, который применяется для обмена данными между системами на разных платформах.

## A

**Администратор базы данных (DBA – Database Administrator)**. Человек, который отвечает за функционирование, целостность и сохранность базы данных.

**Аномалия вставки (insertion anomaly)**. Несогласованность в многотабличной базе данных. Иногда появляется при вставке строки в одну из таблиц этой базы.

**Аномалия модификации (modification anomaly)**. Несогласованность данных, которая иногда возникает при модификации (вставке, удалении или обновлении), выполняемой в одной из таблиц базы данных.

**Аномалия обновления (update anomaly)**. Несогласованность данных, которая иногда возникает в базе данных при обновлении строки одной из таблиц.

**Аномалия удаления (deletion anomaly)**. Несогласованность в многотабличной базе данных. Иногда появляется при удалении строки в одной из таблиц этой базы.

**Аплет (applet)**. Небольшое приложение, написанное на языке Java и находящееся на Web-сервере. Предназначено для загрузки и выполнения на Web-клиенте, соединенном с этим сервером.

**Атомарный (atomic)**. Неспособный к делению на части.

**Атрибут (attribute)**. Компонент структурированного типа или отношения.

## Б

**База данных (database)**. Самоописательный набор интегрированных записей.

**База данных корпорации (database, enterprise)**. База данных, в которой находится информация, используемая во всей корпорации.

**База данных организации (database, organizational)**. База данных, в которой находится информация, используемая во всей организации.

**База данных, персональная (database, personal)**. База данных, предназначенная для использования одним человеком на единственном компьютере.

**База данных рабочей группы (database, workgroup)**. База данных, предназначенная для использования внутри организации каким-либо ее отделом или рабочей группой.

**Базовая переменная (host variable)**. Переменная приложения, написанного на процедурном базовом языке, и доступная встроенному коду SQL.

**Брандмауэр (firewall)**. Программное обеспечение (или комбинация программного и аппаратного обеспечения) для изоляции сети интранет от Internet и контроля трафика между ними.

## В

**Виртуальная таблица (virtual table)**. Представление.

**Владелец схемы (schema owner)**. Пользователь, указанный при создании схемы в качестве ее владельца.

**Вложенный запрос (nested query)**. Оператор, в котором содержится один или множество подзапросов.

**Внешний ключ (foreign key)**. Столбец или набор столбцов в таблице из базы данных, который ссылается на первичный ключ из другой таблицы той же базы данных.

**Выражение со значением (value expression)**. Выражение, в котором комбинируется не менее двух значений.

**Выражение со значением даты-времени (value expression, datetime)**. Выражение со значением, в котором используются данные типа DATE, TIME, TIMESTAMP или INTERVAL.

**Выражение со значением типа записи (row value expression)**. Заключенный в круглые скобки список значащих выражений, которые отделены друг от друга запятыми.

**Выражение со значением, строковое (value expression, string)**. Выражение со значением, в котором символьные строки комбинируются с помощью оператора конкатенации.

**Выражение со значением, условное (value expression, conditional)**. Выражение со значением, аргументы которого, в зависимости от истинности или ложности некоторых условий, имеют те или иные значения.

**Выражение со значением, числовое (value expression, numeric)**. Выражение со значением, в котором числовые значения комбинируются с помощью операторов сложения, вычитания, умножения или деления.

## Д

**Дескриптор (descriptor)**. Область памяти, используемая для передачи информации между процедурным кодом приложения и кодом динамического SQL того же приложения.

**Домен (domain)**. Набор всех значений, допустимых для элемента базы данных.

**Драйвер (driver)**. Интерфейсная часть СУБД, которая прямо стыкуется с базой данных. Драйверы входят в состав **серверной части** (**back end**).

## З

**Запись (record)**. Представление некоторого физического или умозрительного объекта.

**Запрос (query)**. Вопрос, который вы задаете о данных, находящихся в базе.

**Зарезервированные слова (reserved words)**. Слова, имеющее в SQL специальное значение, которые нельзя использовать в качестве имен переменных или каким-то другим не предназначенным для них способом.

## И

**Иерархическая модель базы данных (hierarchical database model)**. Модель организации данных в древовидной структуре.

**Избыточность данных (data redundancy)**. Хранение одних и тех же данных одновременно в нескольких местах базы.

**Индекс (index)**. Таблица с указателями, которые применяются, чтобы быстро находить строки в таблице данных.

**Информационная схема (information schema)**. Системные таблицы, в которых хранятся **метаданные** (**metadata**) базы данных.

**Источник данных (data source)**. Местонахождение данных, используемых приложением базы данных. Источником данных может быть СУБД или файл данных.

**Итоговая функция (aggregate function)**. Функция, которая выдает единственный результат после обработки целого набора табличных строк. Также называется **функцией набора** (**setfunction**).

## К

**Каталог (catalog)**. Поименованная совокупность схем.

**Кластер (cluster)**. Поименованная совокупность каталогов.

**Клиент (client)**. Рабочая станция отдельного пользователя, на которой находится клиентская часть (**front end**) СУБД, отображающая информацию на экране и реагирующая на ввод, выполняемый пользователем.

**Клиентская часть (front end)**. Часть СУБД, которая непосредственно взаимодействует с пользователем.

**Концептуальное представление (conceptual view)**. Схема базы данных.

**Курсор (cursor)**. Средство SQL, позволяющее определить набор строк, упорядочить их, а также выбрать текущую строку в этом наборе.

## Л

**Логические связки (logical connectives).** Используются для получения более сложных предикатов за счет объединения или изменения логических значений простых предикатов.

## М

**Масштаб (scale)**. Количество цифр в дробной части числового элемента данных.

**Менеджер драйверов (driver manager)**. Компонент ODBC-совместимого интерфейса базы данных. На машинах с операционной системой Windows менеджер драйверов – это библиотека динамической компоновки, которая связывает источники данных с соответствующими драйверами.

**Метаданные (metadata)**. Данные о структуре данных, хранящихся в базе.

**Модуль Netscape (Netscape plug-in).** Программный компонент, загружаемый с Web-сервера на Web-клиент, для расширения функциональных возможностей браузера.

**Модульный язык (module language)**. Форма SQL, при которой операторы SQL размещаются в модулях, вызываемых прикладной программой, написанной на базовом языке.

## Н

**Нормализация (normalization)**. Прием, который уменьшает или устраняет возможность появления в базе данных аномалий модификации.

## О

**Область диагностики (diagnostics area)**. Структура данных, управляемая СУБД. В ней хранится подробная информация о последнем выполнявшемся операторе SQL, а также обо всех ошибках, которые произошли при его выполнении.

**Объединение (join)**. Реляционный оператор, который объединяет данные из множества таблиц в одну таблицу.

**Объект (object)**. Однозначно определяемый предмет.

**Объектная целостность (entity integrity)**. Свойство таблицы из базы данных. Состоит в том, что таблица полностью соответствует моделируемому ею объекту из реального мира.

**Ограничение (constraint)**. Требование, предъявляемое к данным базы.

**Ограничение, задержанное (constraint, deferred)**. Ограничение, которое не применяется, пока не будет определено как **немедленное (immediate)** или пока не будет выполнен оператор COMMIT, завершающий транзакцию.

**Одновременный доступ (concurrent access)**. Ситуация, когда с одними и теми же табличными строками из базы данных одновременно работают не менее двух пользователей.

**Отношение (relation)**. Двумерный массив строк и столбцов, причем у каждого из его элементов может быть только одно значение, а строки друг друга не повторяют.

## П

**Параметр (parameter)**. Переменная в приложении, написанном на модульном языке SQL.

**Первичный ключ (primary key)**. Столбец или набор столбцов в таблице базы данных, который однозначно идентифицирует каждую строку в этой таблице.

**Плоский файл (flat file)**. Набор записей с данными, который имеет минимальную структуру.

**Подзапрос (subquery)**. Запрос, находящийся внутри другого запроса.

**Подтип (subtype)**. Тип данных является подтипом другого типа данных, если каждое значение первого является также значением второго.

**Подъязык данных (data sublanguage)**. Подмножество "полноценного" компьютерного языка, предназначенное специально для обработки данных. SQL является подъязыком данных.

**Последовательность сопоставления (collating sequence)**. Способ упорядочения для символов в символьном наборе. Все последовательности сопоставления, заданные для наборов с латинскими символами (a, b, c), задают естественное упорядочение (a, b, c…). Они отличаются способами упорядочения специальных символов (**+, -, <,?** и т.д.), а также для цифр и букв по отношению друг к другу.

**Предикат (predicate)**. Утверждение, которое может быть истинным или ложным.

**Представление (view)**. Компонент базы данных, который ведет себя как таблица, но самостоятельно не существует.

**Преобразование (mapping)**. Преобразование данных из одного формата в другой.

**Процедурный язык (procedural language)**. Язык программирования, на котором решение задачи записывается в виде последовательности действий.

**Псевдоним (alias)**. Краткий заменитель имени таблицы.

**Публикация базы данных (database publishing)**. Действие, в результате которого содержимое базы данных становится доступным в Internet или интранет.

## Р

**Распределенная обработка данных (distributed data processing)**. Система, в которой данные обрабатываются множеством серверов.

**Реализация (implementation)**. Отдельная реляционная СУБД, работающая на конкретной аппаратной платформе.

## С

**Сервер базы данных (database server)**. Серверная часть в системе **клиент/сервер** (**client/server system**).

**Серверная часть (back end)**. Часть СУБД, которая непосредственно взаимодействует с базой данных.

**Сетевая модель базы данных (network database model)**. Способ организации базы данных, при котором каждый из элементов данных (узел) разрешается связывать с любым другим. Используется для минимизации избыточности данных.

**Сеть интранет (intranet)**. Сеть, в которой используется аппаратное и программное обеспечение World Wide Web, но доступная пользователям, работающим в одной организации.

**Система дистанционной обработки (teleprocessing system)**. Мощный центральный процессор, соединенный с множеством терминалов.

**Система клиент/сервер (client/server system)**. Многопользовательская система, в которой "центральная машина" (сервер) соединяется с множеством интеллектуальных пользовательских рабочих станций (клиентов).

**Составной ключ (composite key)**. Ключ, состоящий не менее чем из двух столбцов таблицы.

**Ссылочная целостность (referential integrity)**. Состояние, при котором все таблицы базы данных согласованы друг с другом.

**Столбец (column)**. Компонент таблицы, в котором находится один из ее атрибутов.

**Строка (row)**. Последовательность пар (**имя поля, значение**).

**Структурированный тип (structured type)**. Тип, определяемый пользователем, представленный как перечень определений атрибутов и методов. Структурированный тип не базируется на отдельном предопределенном исходном типе.

**СУБД (DBMS)**. Система управления базами данных.

**Супертип (supertype)**. Тип данных является супертипом другого типа данных, если каждое значение второго является также значением первого.

**Схема (schema)**. Структура всей базы данных. Информация, которая описывает схему, является **метаданными** (**metadata**) базы данных.

## Т

**Таблица (table)**. Отношение.

**Таблица трансляции (translation table)**. Инструмент для преобразования символьных строк из одного набора символов в другой.

**Тип данных (data type)**. Совокупность значений, способных представлять данные.

**Тип коллекций (collection type)**. Тип данных, которые позволяют хранить множество объектов в одном поле строки таблицы.

**Тип ссылки (reference type)**. Тип данных, значения которых являются ссылками на другие данные.

**Тип, определяемый пользователем (user-defined type)**. Тип, характеристики которого определяются пользователем.

**Точность (precision)**. Максимальное количество цифр, которое может иметь числовой элемент данных.

**Транзакция (transaction)**. Последовательность операторов SQL, результат выполнения которой доступен другим транзакциям только после полного выполнения всех операторов этой транзакции.

**Транзитивная зависимость (transitive dependency)**. Один атрибут отношения зависит от второго атрибута, который, в свою очередь, зависит от третьего.

**Триггер (trigger)**. Небольшой код, который определяет реакцию СУБД на некоторые операторы SQL.

## У

**Управляющий элемент ActiveX (ActiveX control)**. Повторно используемый программный компонент, который можно встраивать в приложение, уменьшая таким образом время разработки. ActiveX использует фирменную технологию Microsoft. Такие компоненты могут использоваться только в средах разработки в операционных системах Windows.

**Утверждение (assertion)**. Ограничение, указанное не в операторе CREATE TABLE, а в CREATE ASSERTION. Обычно применяется к более чем одной таблице.

## Ф

**Файловый сервер (file server)**. Серверный компонент системы с разделением ресурсов. Не содержит никаких программ управления базой данных.

**Функциональная зависимость (functional dependency)**. Связь между атрибутами отношения.

**Функция значения (value function)**. Функция, которая выполняет операцию с одной символьной строкой, числом или значением типа даты-времени.

**Функция набора (set function)**. Функция, которая в результате обработки целого набора табличных строк выдает единственный результат. Также называется **итоговой функцией** (**aggregate function**).

**Функция-мутатор (mutator function)**. Функция, связанная с типом, определяемым пользователем, и имеющая два параметра, зависящих от некоторого атрибута этого типа. Первый параметр (результат) имеет тот же тип, что и тип, определяемый пользователем, а второй параметр – тот же тип, что и определяющий атрибут.

## Ц

**Целостность домена (domain integrity)**. Свойство столбца из таблицы базы данных. Состоит в том, что все элементы данных из этого столбца находятся в его домене.

## Я

**Ядро базы данных (database engine)**. Часть СУБД, которая непосредственно взаимодействует с базой данных и является одним из компонентов **серверной части (back end).**

**Язык манипулирования данными (DML – Data Manipulation Language)**. Часть SQL, отвечающая за непосредственную работу с данными базы.

**Язык определения данных (DDL – Data Definition Language).** Часть SQL, которая используется для определения, изменения и уничтожения структур базы данных.

**Язык разметки гипертекста (HTML – HyperText Markup Language)**. Стандартный язык форматирования Web-документов.

**Язык управления данными (DCL – Data Control Language)**. Часть SQL, отвечающая за защиту базы данных.