ВЕДЬ ЭТО ТАК ПРОСТО!

9-е издание

SQL для чайников®

Издательство ДИАЛЕКТИКА

Изучите все новинки последней версии стандарта SQL

Начните создавать реляционные базы данных

Защитите свою базу данных от несанкционированного доступа

Аллен Тейлор

SQL

для чайников®

SQL

by Allen G. Taylor

SQL

Аллен Тейлор

для чайников®

Москва ◆ Санкт-Петербург
2020

ББК 32.973.26-018.2.75
Т30
УДК 681.3.07

Компьютерное издательство "Диалектика"

Перевод с английского *А.П. Сергеева*
Под редакцией *В.Р. Гинзбурга*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Тейлор, Аллен

Т30 SQL для чайников, 9-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2020. — 544 с. : ил. — Парал. тит. англ.

ISBN 978-5-907144-81-1 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание
Аллен Тейлор

SQL для чайников,

9-е издание

Подписано в печать 03.10.2019. Формат 70×100/16
Гарнитура Times.
Усл. печ. л. 43,86. Уч.-изд. л. 24,5
Тираж 500 экз. Заказ № 8171

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907144-81-1 (рус.)
ISBN 978-1-119-52707-7 (англ.)

Оглавление

Содержание

Об авторе

Аллен Тейлор — ветеран компьютерной индустрии с тридцатилетним стажем, автор более 40 книг по компьютерной тематике. Читает лекции по базам данных и компьютерным технологиям, а также ведет онлайн-курсы по разработке баз данных. Если вам интересно, чем занимается Аллен в настоящее время, посетите его сайт `www.allengtaylor.com`. Можете связаться с ним и по электронной почте, написав письмо по адресу `allen.taylor@ieee.org`.

Введение

Добро пожаловать в мир баз данных, где царит SQL — язык структурированный запросов! Существует множество систем управления базами данных (СУБД), предназначенных для разных платформ. Различия между этими системами довольно велики, однако их объединяет одно: все они поддерживают SQL как средство доступа к данным и управления ими. Если вы знаете этот язык, то сможете создавать реляционные базы данных и извлекать из них полезную информацию.

Цель книги

Системы управления базами данных играют важнейшую роль в большинстве организаций. Люди часто думают, что создание и обслуживание таких систем — удел профессионалов в области баз данных, которым открыты знания, недоступные простым смертным. Книга, которую вы держите в руках, развеет этот миф. Прочитав ее, вы:

- узнаете об истории баз данных;
- получите представление о структуре СУБД;
- откроете для себя основные функциональные компоненты SQL;
- научитесь создавать базы данных;
- сможете защищать базы данных от потенциальных опасностей;
- научитесь работать с данными;
- узнаете, как извлечь из базы данных нужную информацию.

Цель книги — научить вас создавать реляционные базы данных и извлекать из них нужную информацию средствами SQL. Этот язык принят в качестве международного стандарта и повсеместно применяется для создания и обслуживания реляционных баз данных. В новом издании книги рассмотрена последняя версия стандарта: SQL:2016.

В книге не будет говориться о том, как проектировать базы данных. Предполагается, что у вас уже есть подходящий проект базы данных, поэтому будет показано, как его реализовать с помощью SQL. Если же вы считаете, что существующий проект недостаточно хорош, то обязательно исправьте его до

того, как приступите к реализации. Чем раньше вы обнаружите недостатки в проекте, тем меньше усилий затратите на их исправление.

Для кого предназначена эта книга

Если вам нужно хранить данные в СУБД или извлекать их из нее, то практическое знание SQL поможет вам делать это эффективнее. Чтобы применять SQL, не нужно быть программистом и знать такие языки, как Java, C или Basic. Синтаксис SQL сходен с синтаксисом английского языка.

Если же вы программист, то сможете внедрить SQL-код в свои программы. Он привнесет в обычные языки программирования мощный аппарат манипулирования данными. В книге будет показано, что именно нужно знать, чтобы реализовать в своих программах богатый набор возможностей, предоставляемых SQL.

Пиктограммы, используемые в книге

Расположенные на полях книги пиктограммы привлекут ваше внимание к важной информации. В книге вам встретится несколько видов пиктограмм.

Советы помогут вам сэкономить немало времени и уберегут от множества проблем.

Прислушайтесь к информации, помеченной данной пиктограммой, — это ценные сведения, которые пригодятся вам позже.

Эту информацию не следует игнорировать, иначе не избежать больших неприятностей.

Данной пиктограммой помечены технические детали, которые могут быть интересны, но не особо существенны для понимания рассматриваемой темы.

Что дальше

Базы данных — самый прекрасный инструмент для отслеживания важной информации. Если вы их освоите и заставите с помощью SQL выполнять свои распоряжения, то перед вами откроются широчайшие возможности. Сотрудники будут обращаться к вам, когда им потребуется найти нужные для работы данные. Руководство станет прислушиваться к вашим советам. Молодежь будет просить у вас автограф. И что важнее всего, вы поймете, причем во всех деталях, как функционируют информационные потоки в вашей организации.

Ждем ваших отзывов!

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

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

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

Наши электронные адреса:

E-mail: `info@dialektika.com`

WWW: `http://www.dialektika.com`

Знакомство с SQL

В ЭТОЙ ЧАСТИ...

- Основы реляционных баз данных
- Основы SQL
- Компоненты SQL

Глава 1

Основы реляционных баз данных

В ЭТОЙ ГЛАВЕ...

- Обработка данных
- Что такое база данных
- Что такое СУБД
- Эволюция моделей баз данных
- Что такое реляционная база данных
- Трудности проектирования баз данных

SQL (Structured Query Language) — это стандартный язык, предназначенный для создания баз данных, добавления новых и поддержки имеющихся данных, а также для извлечения нужной информации. Разработанный в 1970-х годах компанией IBM, SQL (произносится как "эс-кью-эл") превратился за прошедшие годы в отраслевой стандарт. Официально он был утвержден ISO (International Organization for Standardization — Международная организация по стандартизации).

Существуют различные типы баз данных, и определяющим фактором при отнесении базы данных к тому или иному типу является то, какая модель данных используется в каждом конкретном случае. SQL был создан для работы с теми базами, которые следуют *реляционной модели*. Недавно в международный стандарт SQL были включены элементы *объектной модели*, в результате чего появились гибридные структуры, называемые *объектно-реляционными базами данных*. В этой главе будет проведено сравнение реляционной модели

с другими основными моделями и будет дан обзор ключевых особенностей реляционных баз данных.

Впрочем, прежде чем рассказывать об SQL, нужно дать определение понятию *база данных*. Развитие компьютерных технологий изменило способы хранения и обработки информации, а также значение этого термина.

Обработка данных

Сегодня с помощью компьютеров люди решают множество задач, которые раньше решались с помощью других инструментов. Документы теперь создают и исправляют не на пишущих машинках, а на компьютерах. Калькуляторы также заменены компьютерами — лучшим средством выполнения математических расчетов. Компьютеры пришли на смену миллионам листов бумаги, папок и стеллажей для документов и являются основными хранилищами важной информации. В сравнении со старыми инструментами компьютеры выполняют намного больше работы, причем быстрее и, главное, с более высокой точностью. Однако за все приходится платить. Пользователи компьютеров больше не имеют прямого физического доступа к своим данным.

Как только компьютеры внезапно перестают работать, у пользователей сразу же закрадываются сомнения: а действительно ли компьютеризация — благо? Раньше папкам с документами угрожало лишь падение на пол, и, чтобы все вернулось "на круги своя", достаточно было просто нагнуться, собрать выпавшие листы и снова сложить их в папку. Если не считать землетрясений и других природных катаклизмов, то стеллажи с папками никогда не "удалялись", кроме того, они никогда не выдавали никаких сообщений об ошибке. А вот крах жесткого диска — это совсем иное дело: потерянные биты и байты "собрать" невозможно. Отказы оборудования, вызванные механическими, электрическими и человеческими факторами, могут безвозвратно лишить вас бесценных данных. Единственное, что может спасти в таких ситуациях, — как можно более частое резервное копирование данных. Можно также хранить данные в облаке, где они будут резервироваться провайдером облачного хранилища.

Высокая скорость и точность компьютеров только тогда пойдут на пользу, когда будут приняты необходимые меры по защите данных от случайной потери.

При хранении важных данных возникают четыре главные задачи.

- Операции по сохранению данных должны выполняться быстро и легко, так как заниматься ими, скорее всего, вам придется часто.

- Носитель, предназначенный для хранения данных, должен быть надежным. Вряд ли вам понравится новость о пропаже большей части ваших данных.
- Получение данных должно быть быстрым и легким, независимо от количества сохраняемых элементов.
- Необходим способ относительно простого поиска нужной в данный момент информации среди тонн ненужной.

Базы данных соответствуют всем этим четырем критериям. Если приходится хранить более десятка элементов данных, то, вероятнее всего, у вас возникнет желание хранить их именно в базе данных.

Что такое база данных

Понятие *база данных* вошло в широкий обиход довольно-таки поздно и при этом потеряло большую часть своего первоначального смысла. Для кого-то база данных является произвольным набором информационных элементов. Другие же определяют это понятие более строго.

В этой книге мы определим *базу данных* как самоописательный набор интегрированных записей. Технология баз данных полностью компьютеризирована и дополнена языками программирования типа SQL.


ЗАПОМНИ!

Запись является представлением некоего физического или абстрактного объекта. Например, вы собираетесь сохранять данные о своих клиентах. Каждому из них в базе данных выделяется отдельная запись. В каждой записи имеется набор *атрибутов*, таких как имя, адрес и номер телефона. Сами же имена, адреса и другие значения, соответствующие этим атрибутам, представляют собой *данные*.

База данных состоит как из данных, так и из *метаданных*. Метаданные — это данные, которые описывают структуру записей, находящихся в базе. Зная, как хранятся данные, их можно извлечь. Поскольку описание структуры данных находится в самой базе, она является *самоописательной*. База данных является *интегрированной*, так как хранит не только элементы данных, но и существующие между ними связи.

В базе данных метаданные хранятся в области, которая называется *словарем данных*. В нем описываются таблицы, столбцы, индексы, ограничения и другие компоненты, из которых состоит база данных.

Размер и сложность базы данных

Базы данных бывают любых размеров, начиная с простого набора из нескольких записей и заканчивая огромными системами с миллионами записей. Большинство баз данных попадает в одну из трех категорий, в зависимости от размера самой базы данных, масштаба оборудования, на котором она выполняется, и масштаба обслуживающей ее организации.

- **Персональная база данных** предназначена для использования одним человеком на одном компьютере. У такой базы обычно достаточно простая структура и относительно небольшой размер.
- **База данных рабочей группы** используется сотрудниками одного отдела или членами одной рабочей группы в пределах одной организации. Такая база обычно больше персональной и, конечно же, сложнее. Она должна обеспечивать работу сразу нескольких пользователей, которым одновременно нужен доступ к одним и тем же данным.
- **База данных организации** может достигать громадных размеров и полностью моделировать информационный поток на крупном предприятии.

Что такое СУБД

Система управления базами данных (СУБД) — это набор программ, используемых для создания, 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, применяемый для создания такой структуры и работы с данными, которые в ней хранятся.

Сегодня на рынке представлено множество СУБД. Одни из них работают только на больших и мощных машинах, другие — на персональных компьютерах, ноутбуках и планшетах. Требования времени выражаются в том, чтобы СУБД могли работать на нескольких платформах или в сетях, содержащих различные типы компьютеров. В то же время наблюдается четко выраженная тенденция к хранению данных в дата-центрах и даже в *облаке*, которое может представлять собой как общедоступную облачную среду, управляемую большими компаниями (такими, как Amazon, Google или Microsoft), так и частную среду, обслуживаемую самой организацией, которая хранит данные в собственной интрасети.

В наши дни *облако* стало модным словечком, которое получило широкое распространение в профессиональных кругах. Оно представляет собой коллекцию вычислительных ресурсов, доступных в Интернете или локальной сети. Единственное, что отличает вычислительные ресурсы в облаке от аналогичных ресурсов в физическом дата-центре, — это то, что ресурсы в облаке доступны через браузер, а не прикладную программу, которая получает непосредственный доступ к этим ресурсам.


ЗАПОМНИ!

СУБД, которая выполняется на платформах различных типов, как больших, так и малых, называется *масштабируемой*.

ЦЕННОСТЬ НЕ В ДАННЫХ, А В ИХ СТРУКТУРЕ

Много лет назад один умник заявил, что если разложить тело человека на компоненты, такие как атомы углерода, водорода, кислорода и азота (плюс незначительное количество других), то его стоимость будет составлять всего 97 центов. Думаю, вы понимаете, что это совершеннейшая глупость. Люди состоят не из изолированных групп атомов. Наши атомы комбинируются в энзимы, белки, 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. Аналогично и структура баз данных позволяет интерпретировать данные, по отдельности кажущиеся бессмысленными. Именно структура выявляет закономерности и тенденции, скрытые в данных. Неструктурированные данные, как и неупорядоченные атомы, имеют малую ценность или совсем не имеют ее.

Каким бы ни был тип компьютера, хранящего базу данных, а также независимо от того, подключена ли машина к сети, поток информации между базой данных и пользователем, в принципе, один и тот же. На рис. 1.1 показано, как пользователь взаимодействует с базой данных с помощью СУБД. Последняя скрывает технические детали хранения данных, благодаря чему приложению приходится иметь дело только с логическими их характеристиками, а не со способами хранения данных в компьютере.

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 набор записей, представленных в определенном формате одна за другой, т.е. фактически это список записей. По компьютерным меркам плоский файл очень прост. В нем нет метаданных со структурной информацией, а есть лишь сами данные.

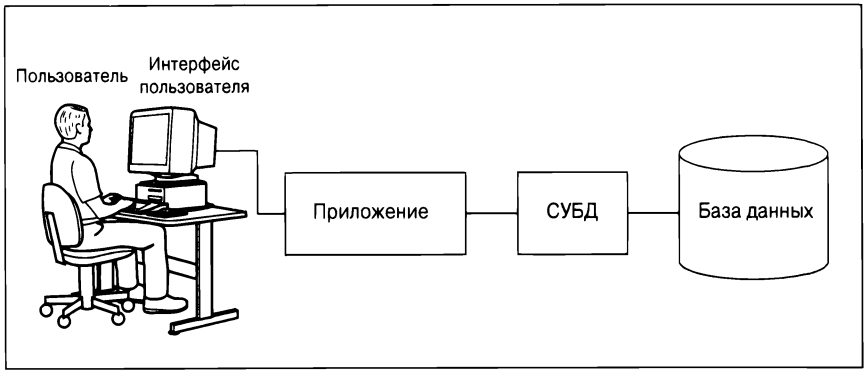


Рис. 1.1. Схема информационной системы, работающей на основе СУБД

Если бы вам нужно было сохранить в виде плоского файла имена и адреса клиентов вашей компании, у него была бы примерно такая структура:

```
Иван Ивченко      630136   ул. Киевская, 11         Новосибирск
Карина Новикова   400086   ул. Латвийская, 39       Волгоград
Александр Цыба    410064   ул. Перспективная, 25    Саратов
Денис Данилкин    129110   пр. Мира, 21             Москва
Семен Шевченко    02000    ул. Космическая, 12      Киев
Тимур Зинкевич    185013   ул. Спортивная, 4        Петрозаводск
Линда Смит        444      Бульвар Сансет           Лос-Анджелес
Стив Джонс        2424     Пятая авеню              Нью-Йорк
```

Как видите, в таком файле нет ничего, кроме данных. Каждое поле имеет фиксированную длину (к примеру, длина поля имени 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, предназначенная для обработки данных из файлов, должна функционировать на очень глубоком уровне детализации. Приложение должно точно знать, где и как в файле хранятся данные. Что касается небольших систем, то в них плоские файлы

работают прекрасно. Но чем крупнее система плоских файлов, тем труднее с ней работать.


СОВЕТ

Использование базы данных вместо системы плоских файлов позволяет избежать дублирования операций при работе с данными. Несмотря на то что файлы базы данных более громоздкие, СУБД могут работать на разных аппаратных платформах и под управлением разных операционных систем. Кроме того, базы данных упрощают процесс написания приложений, 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" сама 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. С другой стороны, когда используется СУБД, такой код в приложениях вообще не нужен.

Кроме того, если в приложении имеется код для работы с данными из плоских файлов, то он, как правило, работает только под управлением определенной операционной системы. Перенос приложения в другую систему — довольно сложный процесс, так как приходится менять весь код, связанный с функциями конкретной системы. А вот перенос на другую платформу аналогичного приложения СУБД проходит намного проще — с меньшим количеством проблем и головной боли.

Модели баз данных

Первые базы данных (которые появились в далекие 1950-е) были структурированы в соответствии с иерархической моделью. Основной их недостаток заключался в избыточности данных и негибкости структуры, что существенно усложняло модификацию таких баз данных. В качестве решения основной проблемы иерархической модели появились базы данных, которые подчинялись сетевой модели. В сетевых базах данных избыточность минимальна, но за это приходится платить сложностью структуры.

Несколькими годами позже доктор Э.Ф. Кодд из компании IBM разработал *реляционную* модель баз данных, которая отличалась минимальной избыточностью и прозрачностью структуры. Для работы с реляционными базами данных

и был изобретен SQL. Со временем реляционные базы данных отправили своих предшественниц (иерархические и сетевые базы данных) на свалку истории.


ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Нельзя не упомянуть о появлении так называемых баз данных NoSQL, которые лишены структуры реляционных баз данных и не используют SQL. В этой книге мы не будем их рассматривать. При желании сведения о них можно найти в Интернете.

Реляционная модель

Доктор Кодд впервые сформулировал реляционную модель баз данных в 1970 году, 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 изменять структуру, не внося корректив в приложения, чего не скажешь о приложениях, созданных на основе старых структур. Предположим, к примеру, что в таблицу базы данных были добавлены один или несколько новых столбцов. В таком случае нет необходимости изменять какое-либо из уже написанных приложений, обрабатывающих данную таблицу. Это потребуется сделать, только если вы изменили столбцы, с которыми работают приложения.


ВНИМАНИЕ!

Естественно, если вы удалили столбец, к которому обращается существующее приложение, то какая бы модель базы данных ни использовалась, вы столкнетесь с проблемами. Один из лучших способов устроить аварийное завершение приложения, работающего с базой данных, — запросить у него информацию, которой нет в базе.

Компоненты реляционной базы данных

Гибкость реляционных баз данных объясняется тем, что их данные хранятся в таблицах, которые в значительной степени независимы одна от другой. Можно добавлять данные в таблицу, 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 ним. (Об отношениях родительских и дочерних таблиц будет идти речь в главе 5.) Далее будет показано, из чего состоят таблицы и как они связаны с другими элементами реляционной базы данных.

Отношения

В праздничные дни родственники собираются за общим столом. В базах данных также есть "родственные" отношения, однако каждое из них имеет собственный "стол" — таблицу. Любая реляционная база данных включает одно или несколько отношений.

Отношение — это двухмерный массив строк и столбцов, причем строки не дублируют друг друга. Каждая ячейка в массиве может содержать только одно значение, и никакие из двух строк не могут быть одинаковыми.

Большинство людей знакомы с *двухмерными массивами*. Например, это электронные таблицы, с которыми можно работать в приложениях типа Microsoft Excel. Другой пример двухмерного массива — это статистика на оборотной стороне именной карточки бейсболиста Главной лиги. В такой карточке имеются столбцы с указанием года, команды, количества игр, результативных ударов, заработанных очков и других статистических показателей. Строки соответствуют годам, на протяжении которых игрок выступал в Главной лиге. Эти данные можно хранить в таблице, которая имеет такую же простую структуру. На рис. 1.2 показана таблица реляционной базы данных, содержащая статистику по одному из игроков. В действительности такая таблица может хранить статистику для всей команды и даже для всей лиги.

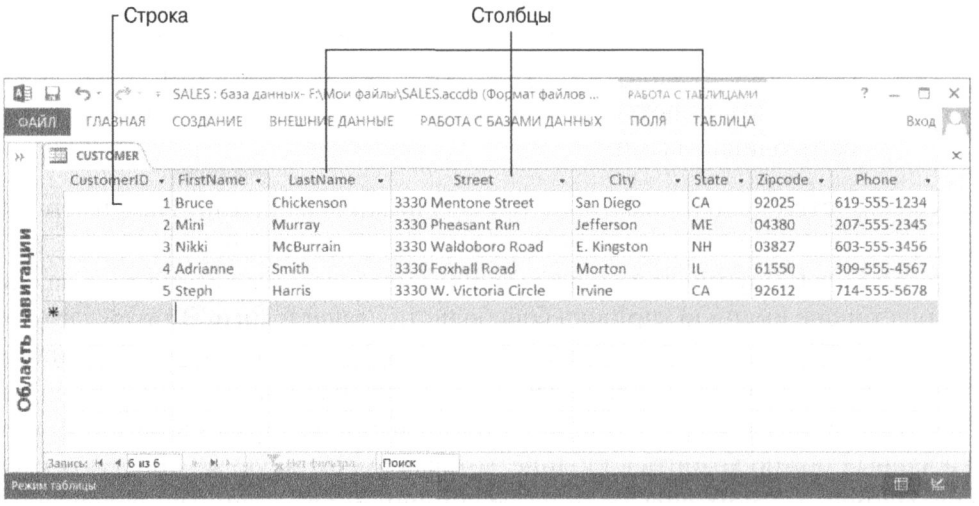
Player	Year	Team	Game	At Bat	Hits	Runs	RBI	2B	3B	HR	Walk	Steals	Bat. Avg.
Roberts	1988	Padres	5	9	3	1	0	0	0	0	1	0	.333
Roberts	1989	Padres	117	329	99	81	25	15	8	3	49	21	.301
Roberts	1990	Padres	149	556	172	104	44	36	3	9	55	46	.309

Рис. 1.2. Таблица со статистикой игрока

Столбцы в массиве являются *самосогласованными*. Это значит, что в каждой строке они имеют один и тот же смысл. Если в одной строке столбец содержит фамилию игрока, то в остальных строках того же столбца должны быть именно фамилии. Порядок расположения строк и столбцов в массиве не имеет значения. Что касается СУБД, то для нее безразлично, 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 базы данных представляет один из ее атрибутов. 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 строка таблицы (также называемая *записью* или *кортежем*) содержит данные по одному клиенту. А каждый столбец содержит один *атрибут*, например номер клиента, его имя, улицу, на которой он живет, город, почтовый индекс или номер телефона. Столбцы и строки такой таблицы показаны на рис. 1.3.



CustomerID	FirstName	LastName	Street	City	State	Zipcode	Phone
1	Bruce	Chickenson	3330 Mentone Street	San Diego	CA	92025	619-555-1234
2	Mini	Murray	3330 Pheasant Run	Jefferson	ME	04380	207-555-2345
3	Nikki	McBurrain	3330 Waldoboro Road	E. Kingston	NH	03827	603-555-3456
4	Adrianne	Smith	3330 Foxhall Road	Morton	IL	61550	309-555-4567
5	Steph	Harris	3330 W. Victoria Circle	Irvine	CA	92612	714-555-5678

Рис. 1.3. Каждая строка базы данных содержит запись, а каждый столбец содержит один атрибут

ЗАПОМНИ!

Понятие *отношение* в модели базы данных соответствует понятию *таблица* в базе данных, основанной на этой модели.

Представления

Мне часто представляется один прекрасный пейзаж: вид на Йосемитскую долину весенним вечером на выезде из туннеля Уовона. Золотистый свет заливает отвесную поверхность скалы Эль-Капитан, вдали сияет пик Хаф-Доум, а водопад "Фата невесты" (Бридалвейл) сверкает серебристым каскадом, в то время как по небу бежит стайка легких облаков. У баз данных тоже имеются представления, пусть и не столь живописные. Их красота заключена в реальной пользе, которую они приносят при работе с данными.

Таблицы могут содержать достаточно много строк и столбцов. Иногда вас могут интересовать все эти данные, а иногда — нет. В какой-то момент вам, возможно, потребуется просмотреть содержимое только некоторых из столбцов, имеющихся в таблице, или только строк, которые удовлетворяют определенному условию. Или же вас могут заинтересовать некоторые столбцы из одной таблицы и некоторые — из другой, связанной с ней. Чтобы исключить

данные, которые вам в данный момент не нужны, можно создать *представление*, являющееся подмножеством базы данных. В представлении могут находиться фрагменты одной или нескольких таблиц.


ЗАПОМНИ!

Представления иногда называют *виртуальными таблицами*. Для приложения или пользователя они выглядят точно так же, как и обычные таблицы. Однако представления сами по себе не существуют. Они всего лишь позволяют просматривать нужные данные, не будучи при этом структурной частью базы данных.

Предположим, вы работаете с базой данных, в которой существуют 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 индекс) и `Phone` (телефон). А столбцами второй таблицы являются `InvoiceNumber` (номер счета-фактуры), `CustomerID` (идентификатор клиента), `Date` (дата), `TotalSale` (сумма сделки), `TotalRemitted` (сумма перечисления) и `FormOfPayment` (форма оплаты).

Коммерческий директор хочет видеть на экране только такие реквизиты клиентов, как имя, фамилия и номер телефона. Если на основе таблицы `CUSTOMER` создать представление, в котором содержатся лишь три столбца с названной информацией, то директор будет просматривать только то, что ему нужно. Данные же из тех столбцов, которые его не интересуют, он видеть не будет. На рис. 1.4 показан пример представления для коммерческого директора.

В то же время менеджер по продажам, возможно, захочет видеть имена, фамилии и номера телефонов всех клиентов с почтовым индексом в диапазоне 90000–93999 (Южная и Центральная Калифорния). Эту работу выполняет представление, которое накладывает ограничение на получаемые с его помощью записи, а также на выводимые с его помощью столбцы. На рис. 1.5 иллюстрируется получение представления для менеджера по продажам.

Финансовому менеджеру, вполне вероятно, требуется просматривать имена и фамилии клиентов из таблицы `CUSTOMER`, а также значения столбцов `Date`, `TotalSale`, `TotalRemitted` и `FormOfPayment` из таблицы `INVOICE`, причем только из таких записей, в которых 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 финансового менеджера.

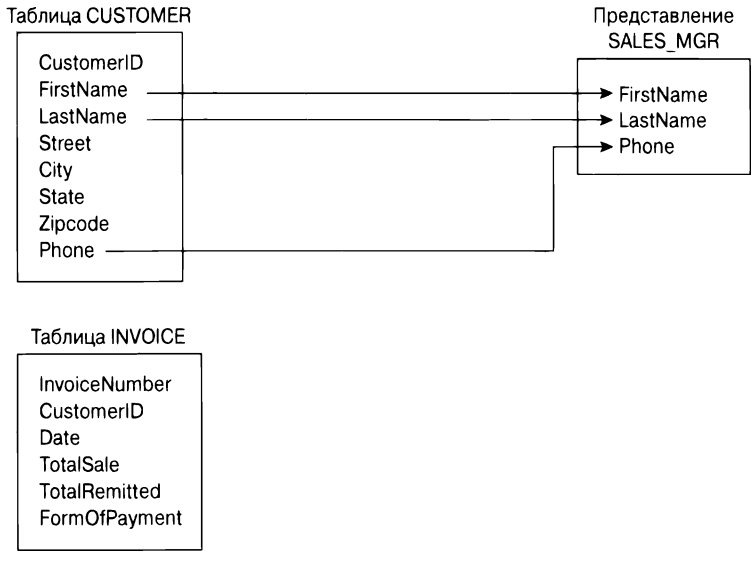


Рис. 1.4. Представление для коммерческого директора, получаемое из таблицы `CUSTOMER`

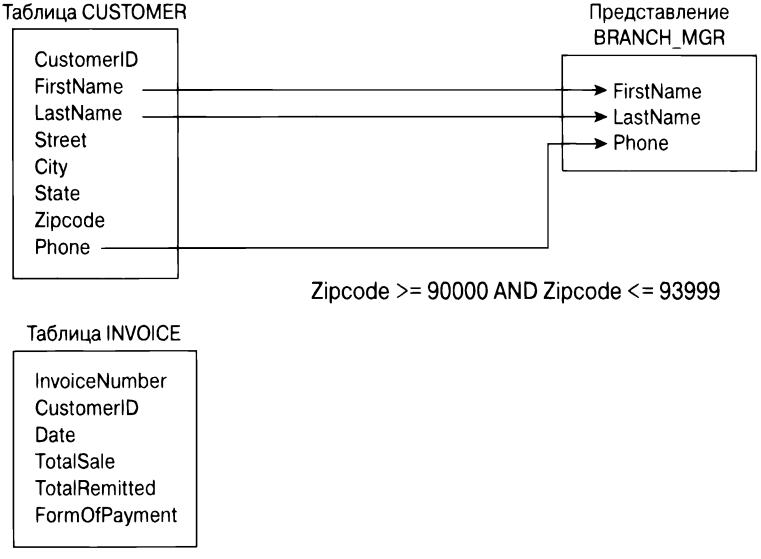


Рис. 1.5. В представление для менеджера по продажам включаются только некоторые записи из таблицы `CUSTOMER`

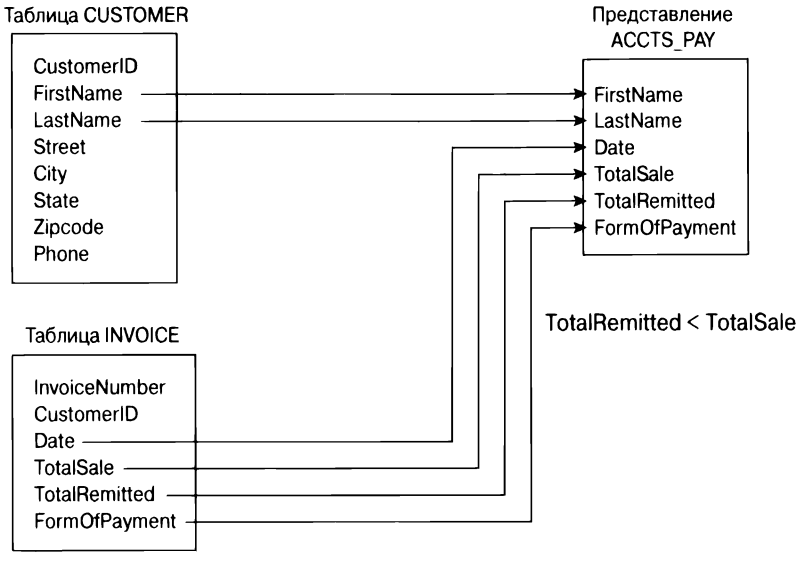


Рис. 1.6. Представление для финансового менеджера собирает данные из двух таблиц

Представления полезны потому, что позволяют получать и выводить информацию из базы данных в определенном формате, не оказывая при этом влияния на хранящиеся данные. О том, как создать представление с помощью SQL, речь пойдет в главе 6.

Схемы, домены и ограничения

База данных — это не просто набор таблиц. В ней имеются и другие структуры, которые помогают поддерживать целостность данных на нескольких уровнях. *Схема* базы данных содержит общую структуру таблиц. *Домен* табличного столбца определяет, какие значения можно в нем хранить. *Ограничения* в таблице базы данных используют для того, чтобы не позволить никому (в том числе самому владельцу) сохранить в ней некорректные данные.

Схемы

Структура всей базы данных — это ее *схема*, или *концептуальное представление*. Такую структуру иногда называют *полным логическим представлением* базы данных. Схема, по сути, — это и есть метаданные, и она является частью хранилища данных. Сами же метаданные, которые описывают структуру базы данных, хранятся в таблицах, похожих на те таблицы, в которых находятся обычные данные. Метаданные — это тоже данные, и в этом их прелесть.

Домены

Атрибут отношения (т.е. столбец таблицы) может допускать конечное количество значений. Множество всех таких значений называют *доменом* атрибута.

В качестве примера предположим, что вы являетесь автодилером и торгуете новинкой — спортивным купе Curarri GT 4000. Данные об автомобилях, находящихся на складе, вы храните в базе данных, точнее — в таблице, которая называется `INVENTORY` (склад). Один из столбцов этой таблицы вы назвали `Color` (цвет), и в нем находятся данные о цвете каждого автомобиля. Эта модель автомобиля поступает только в четырех цветах: малиновый, черный, белый и металлик. Эти четыре цвета и представляют собой домен атрибута `Color`.

Ограничения

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

Предотвратить ввод в столбец неправильных данных можно, применяя к нему жесткие ограничения. Любое значение, входящее в домен столбца, должно удовлетворять всем ограничениям, существующим для этого столбца. Как уже говорилось в предыдущем разделе, доменом столбца является множество всех значений, которые в нем допустимы. Ограничение дополнительно сужает это множество.

Возвращаясь к примеру с автомобилями, можно применить к таблице ограничение, при котором в столбец `Color` можно будет вводить только одно из четырех вышеперечисленных значений. Если после этого оператор ввода попытается добавить в столбец некорректное значение, например "темно-зеленый", система его не примет. Ввод данных нельзя будет продолжить до тех пор, пока оператор не введет в поле `Color` допустимое значение.

Вы спросите: "А что произойдет, если компания решит представить в качестве модели второй половины года темно-зеленую версию Curarri GT 4000?" Ответ: программисты, занимающиеся обслуживанием баз данных, никогда не останутся без работы. Такое случается регулярно и требует обновления структуры базы данных. Только разработчики, которые знают, как модифицировать структуру базы данных, окажутся в состоянии предотвратить катастрофу.

Объектная модель бросает вызов реляционной

Реляционная модель оказалась весьма успешной в самых разных прикладных областях. Однако "беспроблемной" ее назвать нельзя. Ее ограниченность стала проявляться с ростом популярности объектно-ориентированных языков программирования, таких как C++, Java и C#, которые позволяют решать более

сложные задачи, чем обычные языки программирования. Это достигается благодаря таким возможностям, как расширяемость типов данных, инкапсуляция, наследование, полиморфизм методов, а также возможность создания составных объектов.

В книге не будет объясняться весь объектно-ориентированный жаргон (хотя в процессе изложения материала некоторые термины все же будут затронуты). Достаточно сказать, что со многими из этих концепций классическая реляционная модель не совсем хорошо стыкуется. В результате были разработаны СУБД, работающие на основе объектной модели. Несмотря на то что ООП (объектно-ориентированное программирование) стало очень популярным, а объектно-ориентированные базы данных — нет, идея создания последних по-прежнему жива.

Объектно-реляционная модель

Разработчики баз данных, как и все остальные люди, постоянно ищут лучшие решения. Они размышляют: "Не правда ли, было бы здорово воспользоваться преимуществами объектно-ориентированных СУБД и при этом рассчитывать на их совместимость с реляционной моделью, которую мы знаем и любим?" Такого рода размышления и привели к созданию гибридной объектно-реляционной модели. Объектно-реляционные СУБД расширяют реляционную модель настолько, что в ней стало возможным поддерживать объектно-ориентированное моделирование данных. Объектно-ориентированные модели были добавлены в международный стандарт SQL для того, чтобы поставщики реляционных СУБД могли преобразовать свои продукты в объектно-реляционные СУБД, сохранив при этом совместимость со стандартом. Таким образом, если стандарт SQL-92 описывает чисто реляционную модель баз данных, то SQL:1999 — объектно-реляционную модель. Стандарт SQL:2003 включает еще больше объектно-ориентированных свойств, и последующие версии SQL развиваются именно в этом направлении.

В книге описывается международный стандарт ISO/IEC SQL. (Если вам интересно, то аббревиатура "IEC" означает "International Electrotechnical Commission" — Международная электротехническая комиссия.) Этот стандарт предписывает главным образом использование реляционной модели баз данных. В книге также рассмотрены объектно-ориентированные расширения, которые были введены в стандарте SQL:1999, и ряд дополнительных расширений, включенных в более поздние версии. Объектно-ориентированные возможности нового стандарта SQL позволяют разработчикам решать с помощью баз данных задачи, которые не по зубам старой, чисто реляционной парадигме. Производители СУБД в своих продуктах активно задействуют описанные

в стандарте ISO объектно-ориентированные функции. Одни из них обкатаны уже давно, а другим обкатка еще только предстоит.

Вопросы проектирования баз данных

База данных является представлением физической или концептуальной структуры, такой, например, как статистика выступлений всех бейсбольных клубов Главной лиги. Точность представления зависит от уровня детализации, достигнутой в проекте базы данных. Объем приложенных к такому проекту усилий зависит от типа информации, которую вы собираетесь извлекать из базы данных. Чрезмерная детализация — это напрасная трата усилий, времени и места на жестких дисках. А слишком малая детализация может свести ценность базы данных к нулю.


СОВЕТ

Определите, какая степень детализации вам нужна сейчас, а какая может понадобиться в будущем, и воплотите именно этот уровень детализации в своем проекте. Впрочем, не удивляйтесь, если позже придется приводить проект в соответствие с постоянно изменяющимися условиями реального мира.


ЗАПОМНИ!

Современные СУБД, оснащенные привлекательным графическим интерфейсом и интуитивно понятными средствами проектирования, могут навеять потенциальному разработчику ложное чувство безопасности. В таких системах проектирование базы данных кажется сравнимым с созданием простой электронной таблицы. Но это не тот случай. Проектирование базы данных — довольно сложный процесс. При неправильном подходе вы получите базу, которая со временем будет становиться все хуже и хуже. Часто проблема не проявляется до тех пор, пока вы не потратите достаточно времени и усилий на ввод данных. И к тому времени, когда проблема себя проявит, она станет достаточно серьезной. В большинстве случаев решение сводится к тому, чтобы полностью перепроектировать базу и заново ввести все данные. Единственным положительным моментом является лишь то, что к тому времени, когда второй вариант базы данных будет полностью готов, вы наконец поймете, что действительно стали специалистом в области баз данных.

Глава 2

Основы SQL

В ЭТОЙ ГЛАВЕ...

- Что такое SQL
- Заблуждения, связанные с SQL
- Стандарты SQL
- Инструкции и ключевые слова SQL
- Представление чисел, символов, дат, времени и других типов данных
- Пустые значения и ограничения
- Использование SQL в среде "клиент/сервер"
- Сетевые возможности SQL

SQL — гибкий язык, который можно использовать самыми разными способами. Он является самым распространенным инструментом для работы с реляционными базами данных. В этой главе вы узнаете, что такое SQL, в частности, чем он отличается от других компьютерных языков. Затем вы ознакомитесь с инструкциями и типами данных, которые поддерживаются в стандарте SQL. Кроме того, будут раскрыты такие понятия, как *пустые (неопределенные) значения* и *ограничения*. И наконец, вы получите представление о том, как применять SQL в среде "клиент/сервер", а также в Интернете и корпоративных сетях.

Что такое SQL

Первое, что нужно уяснить насчет SQL, — данный язык не является *процедурным*, в отличие от Python, C, C++, C# и Java. Чтобы решить задачу с помощью одного из процедурных языков, приходится писать *программу*, которая выполняет одну за другой указанные операции, пока ее работа не будет завершена. Процедура может быть линейным алгоритмом или содержать ветвление, но в любом случае программист задает порядок выполнения.

В противоположность этому SQL является *непроцедурным языком*. Чтобы с его помощью решить задачу, сообщите SQL, *что именно* вам нужно, как будто вы говорите с джинном из лампы Аладдина. И при этом не требуется указывать, *каким образом получить* то, что вы хотите. СУБД сама решит, как лучше выполнить ваш запрос.

Однако миллионы программистов (и вы, возможно, один из них) привыкли решать задачи процедурным путем, поэтому в последние годы оказывалось немалое давление, чтобы дополнить SQL определенными процедурными возможностями. По этой причине теперь в спецификации SQL имеются такие признаки процедурного языка, как блоки `BEGIN`, условные инструкции `IF`, функции и процедуры. Благодаря новым средствам программы можно хранить на сервере, чтобы к ним могли обращаться (причем неоднократно) многие пользователи.

Чтобы вы поняли, что имелось в виду под фразой "сообщите SQL, что именно вам нужно", предположим, что имеется таблица `EMPLOYEE` с данными о служащих и нужно извлечь из нее все строки, соответствующие всем опытным сотрудникам. Опытным можно считать каждого, кто старше 40 лет или кто получает более 100 тысяч долларов в год. Нужную нам выборку можно получить с помощью следующего запроса:

```
SELECT * FROM EMPLOYEE WHERE AGE > 40 OR SALARY > 100000;
```

При выполнении этой инструкции из таблицы `EMPLOYEE` извлекаются все строки, в которых значение столбца `AGE` (возраст) больше 40 или значение столбца `SALARY` (зарплата) превышает 100 000. SQL сам знает, каким образом следует отбирать информацию. Драйвер базы данных, проведя соответствующий анализ, принимает решение, как лучше выполнить запрос. Все, что от вас требуется, — указать, какие именно данные вам нужны.


ЗАПОМНИ!

Запрос — это вопрос, который вы задаете базе данных. Если какие-либо ее данные удовлетворяют условиям запроса, то SQL возвращает их вам.

В современных реализациях SQL отсутствуют многие простые программные конструкции, которые являются фундаментальными для большинства других языков. В реальных приложениях, как правило, требуются хотя бы некоторые из этих конструкций, поэтому SQL в действительности представляет собой *подъязык* данных. Даже при наличии расширений, добавленных в 1999, 2003, 2005, 2008, 2011 и 2016 годах, для создания полноценного приложения все равно необходимо комбинировать SQL с одним из процедурных языков, таким, например, как C++.

Извлекать информацию из базы данных можно одним из следующих способов.

- **С помощью *интерактивного запроса*, вводя инструкцию SQL и читая на экране результаты ее выполнения.** Такие запросы с клавиатуры (не подлежащие сохранению) имеют смысл тогда, когда требуется получить быстрый ответ на конкретный вопрос. Для решения какой-либо текущей задачи вам могут понадобиться определенные данные, которые до этого никогда не были нужны; возможно, после этого они вам никогда больше не понадобятся. Введите с клавиатуры соответствующий SQL-запрос, и через определенное время на экране появится результат.
- **С помощью программы, которая извлекает из базы данных информацию, а затем создает на основе этих данных отчет, выводимый на экран или на печать.** Сложный SQL-запрос, который, возможно, еще пригодится в будущем, можно поместить прямо в программу, что позволит многократно использовать его в дальнейшем. Таким образом, запрос формулируется всего один раз. Как вставлять SQL-код в программы, написанные на другом языке, вы узнаете в главе 16.

Немного истории

SQL, как и теория реляционных баз данных, появился в одной из исследовательских лабораторий компании IBM. В начале 1970-х годов ученые из IBM разработали первые реляционные СУБД (или РСУБД) и создали подъязык данных, предназначенный для работы в этих системах. Пробная версия данного подъязыка была названа *SEQUEL* (*S*tructured *E*nglish *QU*Ery *L*anguage — структурированный английский язык запросов). Однако когда пришло время официально выпускать язык запросов в качестве продукта, оказалось, что это название уже зарегистрировано в качестве торговой марки другой компанией. Маркетологи IBM решили дать выпускаемому продукту имя, хотя и отличающееся

от SEQUEL, но явно созвучное ему. Так и появилось название "SQL" (произносится как "эс-кью-эл").

О работе, которая велась в IBM над реляционными базами данных и языком запросов, в информационной отрасли хорошо знали, причем еще до того, как компания представила в 1981 году реляционную СУБД SQL/DS. К тому времени компания Relational Software, Inc. (ныне Oracle Corporation) уже выпустила свою первую реляционную СУБД. Эти первоначальные продукты сразу 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 СУБД выпустили собственные версии SQL, причем эти реализации охватывали все основные функциональные возможности продуктов IBM, расширенные так, чтобы раскрыть преимущества именно своих СУБД. В результате, несмотря на то что почти все поставщики использовали варианты одного языка запросов, платформенная совместимость была слабой.


ЗАПОМНИ!

Реализация — это конкретная СУБД, работающая на конкретной аппаратной платформе.

Вскоре началось движение за создание общепризнанного стандарта SQL, которого могли бы придерживаться все. В 1986 году Американский национальный институт стандартов (American National Standards Institute — ANSI) выпустил официальный стандарт: SQL-86. Он был обновлен той же организацией в 1989 году и получил название "SQL 89", а затем, в 1992 году, был назван "SQL-92". Теперь поставщики СУБД, выпуская новые версии своих продуктов, старались приблизить свои реализации к стандарту. Эти усилия привели к тому, что мечта о настоящей переносимости SQL стала намного ближе к реальности.


ЗАПОМНИ!

Последней полной версией стандарта SQL на данный момент является SQL:2016 (ISO/IEC 9075-X:2016). В книге описан язык, определяемый именно этим стандартом. Любая конкретная реализация SQL в определенной степени отличается от стандарта. Так как полный стандарт SQL является слишком общим, от современных реализаций не стоит ждать точного соответствия ему. Однако поставщики СУБД сейчас работают над тем, чтобы их системы все же соответствовали основной части стандарта SQL. Полные спецификации стандарта ISO/IEC доступны для покупки в Интернете по адресу `www.iso.org/search.html?q=iso%209075`, но вам вряд ли стоит их покупать, если вы, конечно, не собираетесь в ближайшее

время создавать собственную СУБД по стандарту ISO/IEC. Описание стандарта имеет свою специфику и рассчитано не на новичков, а на тех, кто имеет солидный опыт программирования.

Инструкции SQL

SQL состоит из ограниченного количества инструкций, специально предназначенных для управления данными. Одни из этих инструкций служат для определения данных, другие — для их обработки, остальные — для администрирования. Об инструкциях определения и обработки данных рассказывается в главах 4–12, а об инструкциях администрирования — в главе 14.

Чтобы соответствовать стандарту SQL:2016, реализация должна включать все основные возможности. Кроме того, в ее состав могут входить и расширения основного набора (которые также описаны в спецификации SQL:2016). В табл. 2.1 приведен список инструкций SQL:2016 (как основного набора, так и расширенного).

Таблица 2.1. Инструкции SQL:2016

ADD	DEALLOCATE PREPARE	FREE LOCATOR
ALLOCATE CURSOR	DECLARE	GET DESCRIPTOR
ALLOCATE DESCRIPTOR	DECLARE LOCAL TEMPORARY TABLE	GET DIAGNOSTICS
ALTER DOMAIN	DELETE	GRANT PRIVILEGE
ALTER ROUTINE	DESCRIBE INPUT	GRANT ROLE
ALTER SEQUENCE GENERATOR	HOLD LOCATOR	HOLD LOCATOR
ALTER TABLE	DISCONNECT	INSERT
ALTER TRANSFORM	DROP	MERGE
ALTER TYPE	DROP ASSERTION	OPEN
CALL	DROP ATTRIBUTE	PREPARE
CLOSE	DROP CAST	RELEASE SAVEPOINT
COMMIT	DROP CHARACTER SET	RETURN
CONNECT	COLLATION	REVOKE

CREATE	DROP COLUMN	ROLLBACK
CREATE ASSERTION	DROP CONSTRAINT	SAVEPOINT
CREATE CAST	DROP DEFAULT	SELECT
CREATE CHARACTER SET	DROP DOMAIN	SET CATALOG
CREATE COLLATION	DROP METHOD	SET CONNECTION
CREATE DOMAIN	DROP ORDERING	SET CONSTRAINTS
CREATE FUNCTION	DROP ROLE	SET DESCRIPTOR
CREATE METHOD	DROP ROUTINE	SET NAMES
CREATE ORDERING	DROP SCHEMA	SET PATH
CREATE PROCEDURE	DROP SCOPE	SET ROLE
CREATE ROLE	DROP SEQUENCE	SET SCHEMA
CREATE SCHEMA	DROP TABLE	SET SESSION AUTHORIZATION
CREATE SEQUENCE	DROP TRANSFORM	SET SESSION CHARACTERISTICS
CREATE TABLE	DROP TRANSLATION	SET SESSION COLLATION
CREATE TRANSFORM	DROP TRIGGER	SET TIME ZONE
CREATE TRANSLATION	DROP TYPE	SET TRANSACTION
CREATE TRIGGER	DROP VIEW	SET TRANSFORM GROUP
CREATE TYPE	EXECUTE IMMEDIATE	START TRANSACTION
CREATE VIEW	FETCH	UPDATE
DEALLOCATE DESCRIPTOR		

Ключевые слова

Помимо инструкций, специальное значение в SQL имеют и некоторые другие слова. Наряду с инструкциями они зарезервированы для конкретных целей. Эти слова нельзя использовать в качестве имен переменных или любым другим способом, для которого они не предназначены. Совершенно очевидно,

почему таблицам, столбцам и переменным нельзя давать имена из списка ключевых слов. Представьте себе, какая путаница возникнет, например, из-за такой инструкции:

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

Этим все сказано. Полный список ключевых слов SQL приведен в приложении А.

Типы данных

В разных реализациях SQL поддерживаются различные исторически сложившиеся типы данных. В спецификации SQL признаны только семь предопределенных общих типов:

- числовой;
- двоичный;
- строковый;
- логический;
- дата/время;
- интервальный;
- XML.

В каждом из этих типов может существовать несколько подтипов (целые числа, числа с плавающей запятой, символьные строки, двоичные строки, строки больших объектов и т.п.). Кроме встроенных (предопределенных) типов, в SQL поддерживаются коллекции, а также сложные и определяемые пользователем типы — всех их мы рассмотрим далее.


СОВЕТ

Если вы используете реализацию SQL, в которой поддерживаются типы данных, не описанные в спецификации, то для лучшей переносимости вашей базы данных старайтесь не пользоваться ими. Прежде чем создавать и использовать пользовательский тип, убедитесь, что в СУБД, на которую вам, возможно, захочется перейти в будущем, также поддерживаются определяемые пользователем типы.

Целочисленные типы

Как вы, возможно, поняли из названия, *целочисленные типы* позволяют точно выразить значение числа. К этой категории относятся следующие шесть типов:

- INTEGER;
- SMALLINT;
- BIGINT;
- NUMERIC;
- DECIMAL;
- DECFLOAT.

Тип INTEGER

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


ЗАПОМНИ!

Точностью числа называется максимальное количество цифр, которое оно может содержать.

Тип SMALLINT

Тип SMALLINT (малое целое) также предназначен для целых значений, но его точность в конкретной реализации не может быть больше точности типа INTEGER. В то же время во многих реализациях SMALLINT и INTEGER являются одним и тем же типом.

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

Тип BIGINT

Тип данных BIGINT (большое целое) определяется как тип, точность которого не может быть меньше, чем точность данных типа INTEGER. Предел точности данных типа BIGINT зависит от конкретной реализации SQL.

Тип NUMERIC

В данных типа NUMERIC (числовой), кроме целой части, может быть и дробная. Для этих данных можно указать точность и масштаб. Точность, как вы помните, — это максимально возможное количество цифр в числе.


ЗАПОМНИ!

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

При использовании типа NUMERIC у вас есть возможность указать нужные значения точности и масштаба. Если они не определены явно (т.е. задано просто NUMERIC), будут использоваться значения, действующие по умолчанию. Если вы укажете NUMERIC (*p*), то получите требуемую точность (*p*), а значение масштаба будет взято по умолчанию. Выражение NUMERIC (*p*, *s*) позволяет конкретно задать и точность, и масштаб. При определении полей вместо параметров *p* и *s* необходимо ввести нужные значения точности и масштаба соответственно.

Для примера предположим, 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 999 999,999999. Если же вы установите для столбца тип данных NUMERIC (10), то максимальное значение чисел, которые смогут находиться в этом столбце, составит уже только 9999,999999. Параметр (10) определяет максимально возможное для числа количество цифр. Если же для столбца будет указан тип данных NUMERIC (10, 2), то в столбце смогут находиться числа с максимальным значением 99 999 999,99. В этом случае, хотя и остается всего десять цифр, справа от десятичной запятой будут находиться только две из них.


СОВЕТ

Тип данных NUMERIC предназначен для таких значений, как 595,72. Точность этого значения равна 5 (общее количество цифр), а масштаб — 2 (количество цифр справа от десятичной запятой). Для подобных чисел подходит тип данных NUMERIC (5, 2).

Тип DECIMAL

Тип данных DECIMAL (десятичный) подобен NUMERIC. В нем может быть дробная часть, и для него можно указать точность и масштаб. Тип DECIMAL отличается от типа 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 столбец, тип которого определен как NUMERIC (5, 2), нельзя поместить числа, большие 999,99. Если же тип столбца — DECIMAL (5, 2), то в него всегда можно поместить значения как меньшие 999,99, так и (если точность реализации позволяет) большие 999,99.


СОВЕТ

Если в ваших данных имеются дробные части, то применяйте тип NUMERIC или DECIMAL, а если данные всегда состоят из целых чисел, то используйте тип INTEGER, SMALLINT или BIGINT. Для достижения максимальной переносимости следует использовать тип NUMERIC

(а не DECIMAL), потому что поле, тип которого вы определите как NUMERIC (5, 2), будет во всех системах иметь один и тот же диапазон значений.

Тип данных DECFLOAT

Тип данных DECFLOAT появился в спецификации SQL:2016. В отличие от типов REAL и DOUBLE PRECISION, которые представляют числа с плавающей запятой, точнее — их двоичное приближение, тип данных DECFLOAT сочетает точность целочисленного типа данных DECIMAL и быстродействие типа данных FLOAT. Это важно для бизнес-приложений, в которых приближения недопустимы.

Числа с плавающей запятой

Некоторые величины имеют такой большой диапазон возможных значений, что компьютер с фиксированным размером регистра не способен представить все эти значения в точности (*размер регистра* может быть 32, 64 и 128 бит). Обычно в таких случаях точность не слишком важна, и достаточно иметь приближенное значение. Для работы с такими данными стандарт SQL определяет три приближенных числовых типа: REAL, DOUBLE PRECISION и FLOAT.

Тип REAL

Тип данных REAL (действительное число) позволяет хранить числа одинарной точности с плавающей запятой (точность зависит от конкретной реализации). Вообще-то, точность определяется используемым оборудованием. К примеру, 64-разрядная машина дает бо́льшую точность, чем 32-разрядная.


ЗАПОМНИ!

Число с плавающей запятой — это число, имеющее десятичную запятую. Десятичная запятая "плавает", т.е. появляется в разных частях числа, в зависимости от его значения. В качестве примеров чисел с плавающей запятой можно привести 3,1, 3,14 и 3,14159 (нетрудно догадаться, что все они представляют собой значения числа p, но с разной точностью).

Тип DOUBLE PRECISION

Тип данных DOUBLE PRECISION (двойная точность) позволяет представлять числа двойной точности с плавающей запятой, точность которых, опять-таки, зависит от реализации. Арифметика двойной точности в основном применяется в научных расчетах, и при этом для разных научных приложений требуется разная точность.

В некоторых системах тип DOUBLE PRECISION обеспечивает и для мантиссы, и для экспоненты строго в два раза бо́льшую вместимость, чем тип REAL. Если вы забыли, о чем вас учили в средней школе, то напомню, что любое число можно представить в виде *мантиссы*, умноженной на число 10 в степени, показателем которой является *экспонента*. Например, число 6626 можно записать в виде 6,626E3. Число 6,626 является мантиссой, которую вы умножаете на число 10, возведенное в степень 3 (в данном случае число 3 является экспонентой).

Вы не получите никакого выигрыша, если будете с помощью приближенного числового типа данных представлять целые числа (такие, как 6626 или 6626000). Целочисленные типы работают так же, но занимают в памяти меньше места. Однако для очень маленьких или, наоборот, очень больших чисел, таких как 6,626E–34, необходимо использовать приближенный числовой тип. Для таких величин не подходят целочисленные типы. В одних системах тип DOUBLE PRECISION дает чуть больше двойной вместимости мантиссы и чуть меньше двойной вместимости экспоненты, чем тип REAL. В других системах тип DOUBLE PRECISION дает для мантиссы вместимость, в два раза бо́льшую, чем тип REAL, а для экспоненты — ту же, что и REAL. В этом случае точность удваивается, а диапазон — нет.


ЗАПОМНИ!

Спецификация SQL не определяет жестко, что означает тип DOUBLE PRECISION. Она только требует, чтобы точность числа типа DOUBLE PRECISION была больше, чем точность числа типа REAL. Хоть это ограничение и очень слабое, вероятно, оно является наилучшим из возможных, если учитывать те немалые различия, которые могут иметь место в аппаратной конфигурации системы.

Тип FLOAT

Используйте тип данных FLOAT (с плавающей запятой), если существует вероятность переноса вашей базы данных на аппаратную платформу с размерами регистров, отличающимися от размеров регистров платформы, для которой вы первоначально делали проект. Работая с типом данных FLOAT, можно задавать нужную точность, например FLOAT (5). Если оборудование поддерживает указанную точность на базе аппаратной одинарной точности, значит, ваша система использует арифметику с одинарной точностью. Если указанная точность требует арифметики с двойной точностью, то система будет использовать ее.


СОВЕТ

Использование типа FLOAT вместо REAL или DOUBLE PRECISION облегчает перенос баз данных на другие системы, так как тип данных FLOAT позволяет строго определять точность. Точность же данных типа REAL или DOUBLE PRECISION зависит от применяемого оборудования.

Если вы не знаете, какие типы нужно выбирать — точные (NUMERIC/DECIMAL) или приближенные (FLOAT/REAL), — то выбирайте точные. Им требуется меньше системных ресурсов, к тому же они дают точные, а не приблизительные результаты. Типы данных с плавающей запятой, как правило, используются в случае, если диапазон возможных значений достаточно большой, но ведь этот факт вы, скорее всего, сможете определить для себя заранее.

Символьные строки

В базах данных хранятся данные множества типов, в том числе графические изображения, звуки и анимация. Не исключено, что вслед за ними появятся и запахи. Можете представить на своем экране трехмерное, размером 1920×1080 пикселей, 24-битовое изображение большого куска пиццы, в то время как через вашу супермультимедийную плату воспроизводится фрагмент запаха, записанный в местной пиццерии? Подобные эффекты могут лишь вызывать раздражение — по крайней мере до тех пор, пока в систему нельзя будет вводить и данные вкусовых ощущений. Увы, возможно, придется ждать очень долго, пока запах и вкус станут стандартными типами данных в SQL. А пока что типами данных, которые вам придется использовать чаще всего, не считая, конечно, числовых, являются типы символьных строк.

Существуют три главных типа символьных данных:

- фиксированные символьные данные (CHARACTER или CHAR);
- переменные символьные данные (CHARACTER VARYING или VARCHAR);
- данные для больших символьных объектов (CHARACTER LARGE OBJECT или CLOB).

Кроме того, существуют еще три разновидности этих типов данных:

- NATIONAL CHARACTER;
- NATIONAL CHARACTER VARYING;
- NATIONAL CHARACTER LARGE OBJECT.

Тип CHARACTER

Если вы определяете для столбца тип данных CHARACTER или CHAR, то количество символов, которое будет в нем находиться, можно указать, используя синтаксис CHAR (*x*), где *x* и является нужным вам количеством. Если, например, тип данных столбца был определен как CHAR (16), то максимальная длина любых данных, которые можно будет ввести в этот столбец, равна 16 символам. Если аргумент не указан явно (т.е. нет значения в скобках), то SQL считает, что длина поля равна одному символу. Если вы вводите в поле типа

CHARACTER заданной длины меньше символов, чем может в нем поместиться, 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 дополнялось пробелами. Этот тип данных позволяет сохранять именно то количество символов, которое ввел пользователь. Для типа CHARACTER VARYING не существует значения по умолчанию. Чтобы указать этот тип данных, используйте синтаксис CHARACTER VARYING (*x*) или VARCHAR (*x*), где *x* — максимально разрешенное количество символов.

Тип CHARACTER LARGE OBJECT

Тип данных CHARACTER LARGE OBJECT (CLOB) впервые появился в стандарте SQL:1999. Он предназначен для хранения громадных символьных строк, которые для типа CHARACTER слишком велики. Данные типа CLOB представлены во многом так же, как и обычные символьные строки, но на действия, которые можно с ними проводить, наложен ряд ограничений.

Значения типа CLOB нельзя использовать в выражениях PRIMARY KEY (первичный ключ) и FOREIGN KEY (внешний ключ), а также в предикате UNIQUE. Более того, эти данные нельзя использовать для сравнения, за исключением операций равенства или неравенства. Из-за того что у данных типа CLOB слишком большие размеры, они, как правило, всегда остаются на сервере. Вместо них на стороне клиента используется специальный тип данных, именуемый *локатором CLOB*. Значение такого параметра-локатора идентифицирует конкретный большой символьный объект (находящийся на сервере).


ЗАПОМНИ!

Предикат — это инструкция, которая возвращает только логические значения True (истина) или False (ложь).

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

Во многих языках используются специфические символы, которые отсутствуют в других языках. Например, в немецком языке имеются символы, которых нет в английском языке, а набор символов русского языка очень сильно отличается от набора символов английского языка. Если вы установите в своей системе английский язык по умолчанию, то все равно сможете использовать и другие символьные наборы, поскольку 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 задаваемый вами набор символов отличается от того, который используется по умолчанию.

Набор символов можно указывать при определении столбца в таблице. В случае необходимости в каждом столбце можно использовать отдельный набор символов. В следующем примере инструкция создания таблицы задает 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 предназначен для символов из набора, используемого в данной реализации по умолчанию. Столбец LANGUAGE_3 поддерживает символы национального набора данной реализации, а столбец LANGUAGE_2 — греческие символы. И наконец, столбец LANGUAGE_4 предназначен для символов кандзи (азиатские наборы символов теперь доступны во многих СУБД).

Двоичные строки

Двоичные строки — нововведение стандарта SQL:2008. С учетом того, что двоичные данные являются основой цифровых компьютеров начиная с 1930-х годов (ЭВМ Атанасова — Берри), понимание важности двоичных данных в SQL кажется немного запоздавшим (хотя лучше позже, чем никогда). Существуют три двоичных типа данных: BINARY, BINARY VARYING и BINARY LARGE OBJECT.

Тип BINARY

Определив тип столбца как BINARY, можно указать и количество байтов (октетов), хранимых в столбце. Для этого используется синтаксис BINARY (*x*), где *x* — количество байтов. Если, например, вы определяете тип столбца как BINARY (16), то бинарная строка должна иметь длину 16 байтов. Ввод байтов в поле типа BINARY начинается с байта 1.

Тип BINARY VARYING

Используйте тип BINARY VARYING или VARBINARY в случае переменной длины двоичной строки. Чтобы определить этот тип данных, используйте синтаксис BINARY VARYING (*x*) или VARBINARY (*x*), где *x* — максимально разрешенное количество байтов. Минимальный размер строки — 0, а максимальный — *x*.

Тип BINARY LARGE OBJECT

Тип данных BINARY LARGE OBJECT (большой двоичный объект) или BLOB используется для огромных двоичных строк, которые слишком велики для типа BINARY. Примерами таких двоичных строк являются файлы изображений и аудиофайлы. Большие двоичные объекты ведут себя практически так же, как и двоичные строки, но SQL накладывает множество ограничений на действия, которые можно выполнять с ними.

Прежде всего, тип BLOB нельзя использовать в выражении PRIMARY KEY, FOREIGN KEY или в предикате UNIQUE. Кроме того, тип BLOB не участвует в операциях сравнения, кроме равенства или неравенства. Большие двоичные объекты действительно бывают весьма велики, поэтому приложения обычно не передают объекты типа BLOB в базы данных или из них. А чтобы манипулировать данными типа BLOB, они используют специальный клиентский тип данных, называемый *локатором BLOB*. Локатор — это параметр, значение которого идентифицирует двоичный объект большого размера.

Логические значения

Тип данных BOOLEAN (булев) поддерживает два определенных логических значения, True (истина) и False (ложь), а также неопределенное значение Unknown (неизвестное). Если любое из первых двух (определенных) значений сравнить с NULL или с Unknown, то результатом будет Unknown.

Значения даты и времени

В SQL определены пять типов данных, связанных с датой и временем (их называют *темпоральными данными*). Некоторые из них довольно сильно перекрывают друг друга, поэтому в отдельных реализациях все пять типов могут и не поддерживаться.

Если в какой-то реализации не полностью поддерживаются все пять типов темпоральных данных, то при переносе в нее баз данных из других реализаций могут возникнуть проблемы. В таком случае необходимо выяснить, какие способы представления даты и времени существуют в этих двух реализациях — исходной и той, на которую нужно перейти.

Тип DATE

Тип DATE (дата) предназначен для хранения значений даты в следующем порядке: год, месяц, день. Значение года занимает четыре цифры, месяца и дня — по две. Значения этого типа могут представлять любую дату, начиная

с 0001 года и заканчивая 9999 годом. Длина значения типа DATE равна десяти позициям, как, например, для даты 1957-08-14.

Тип TIME WITHOUT TIME ZONE

Тип TIME WITHOUT TIME ZONE (время без учета часового пояса) предназначен для хранения значений времени в следующем порядке: час, минута, 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 58,436 секунды 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 TIME ZONE без дробной части занимает восемь позиций (включая двоеточия), а с дробной частью — девять позиций (вместе с десятичной запятой) плюс количество цифр дробной части. Этот тип данных задается или с помощью синтаксиса TIME, 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* указывается количество цифровых позиций справа от десятичной точки. В предыдущем абзаце был показан пример данных типа TIME WITHOUT TIME ZONE (3).

Тип TIMESTAMP WITHOUT TIME ZONE

В данных типа TIMESTAMP WITHOUT TIME ZONE (метка времени без учета часового пояса) хранится информация как о дате, так и о времени. У данных этого типа такие же значения длины и такие же ограничения, как и у данных типа DATE и TIME WITHOUT TIME ZONE, если не считать одного различия: 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 — дата (один пробел служит в качестве разделителя) и восемь позиций — время. 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 вариантов синтаксиса: TIMESTAMP WITHOUT TIME ZONE или TIMESTAMP WITHOUT TIME ZONE (*p*), где вместо *p* указывается количество позиций, предназначенных для цифр дробной части. Значение параметра *p* не может быть отрицательным, а его максимум зависит от конкретной реализации SQL.

Тип TIME WITH TIME ZONE

Тип данных TIME WITH TIME ZONE (время с учетом часового пояса) совпадает с типом TIME WITHOUT TIME ZONE, за исключением того, что в нем дополнительно имеется информация о разнице между местным и всемирным координированным временем (Universal Time Coordinated — UTC), "правопреемником" среднего времени по Гринвичу (Greenwich Mean Time — GMT). Значение этой разницы может находиться в диапазоне от –12:59 до +13:00. Наличие такой дополнительной информации приводит к тому, что цифры времени занимают 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 (метка времени с учетом часового пояса) аналогичен типу данных TIMESTAMP WITHOUT TIME ZONE, за исключением того, что в нем дополнительно имеется информация о разнице между местным и всемирным временем. Эта дополнительная информация занимает шесть позиций после метки времени. Для поля с данными часового пояса и без дробной части требуется 25 позиций, а для поля с дробной частью — 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 иногда требуется определить интервал между двумя датами или значениями времени.

SQL поддерживает два разных 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 датами в пределах одного месяца. Нельзя смешивать вычисления, в которых используется интервал "год–месяц", с вычислениями, в которых применяется интервал "день–время", потому что в месяцах разное количество дней (28, 29, 30 или 31).

Тип XML

XML — это аббревиатура от "eXtensible Markup Language" (расширяемый язык разметки). Этот язык устанавливает набор правил для добавления разметки к данным. Разметка структурирует данные способом, который позволяет отразить их смысл. XML обеспечивает возможность переноса данных между различными платформами.

Тип данных `XML` характеризуется древовидной структурой, а значит, корневой узел может иметь дочерние узлы, которые, в свою очередь, могут иметь собственные дочерние узлы. Тип `XML`, появившийся в стандарте SQL:2003, был расширен в SQL:2005 и усовершенствован в SQL:2008. Стандарт 2005 года определил для типа `XML` пять параметризованных подтипов, сохранив его (родителя) в первоначальном простом виде. Значения типа `XML` могут существовать как экземпляры двух или более типов, потому что одни подтипы являются подтипами других подтипов. (Их можно было бы назвать "подподтипами" или даже "подподподтипами". К счастью, в стандарте SQL:2008 был определен стандартный способ ссылки на подтипы.)

Основными модификаторами типа `XML` являются `SEQUENCE`, `CONTENT` и `DOCUMENT`, а дополнительными — `UNTYPED`, `ANY` и `XMLSCHEMA`. Древовидная структура, иллюстрирующая иерархические отношения между подтипами, приведена на рис. 2.1.

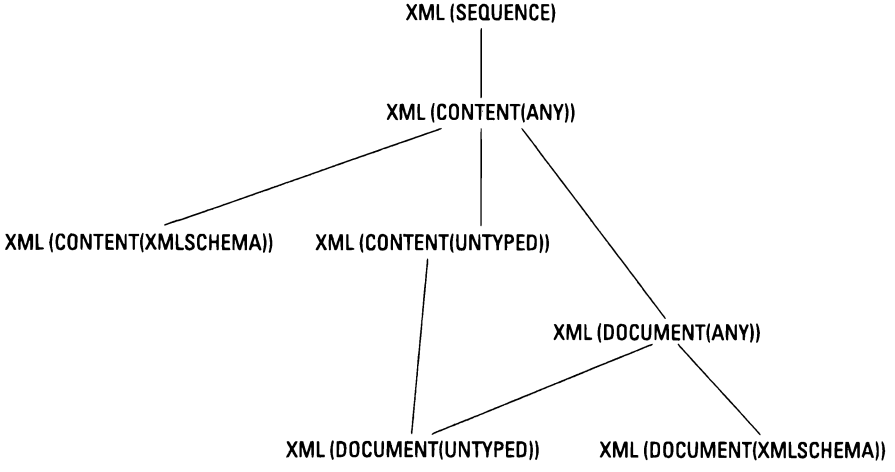


Рис. 2.1. Отношения между подтипами типа `XML`

Вот краткий обзор типов `XML`, с которыми имеет смысл ознакомиться. Не расстраивайтесь, если не все будет понятно сразу; более подробная информация по этим типам приведена в главе 18. Типы перечислены, начиная с самого простого и заканчивая самым сложным.

» XML(SEQUENCE). Любое значение в XML является либо пустым (NULL), либо последовательностью XQuery. Таким образом, любое XML-значение является экземпляром типа XML(SEQUENCE). XQuery — это язык запросов, специально предназначенный для извлечения информации из XML-данных. Это основной тип XML.


ЗАПОМНИ!

XML(SEQUENCE) среди всех типов XML имеет меньше всего ограничений. Он может включать даже плохо сформированные значения XML. Остальные типы не обладают такой "толерантностью".

» XML(CONTENT(ANY)). Этот тип чуть более ограничивающий, чем XML(SEQUENCE). Любое значение XML, являющееся либо пустым, либо узлом документа XQuery, либо дочерним от этого узла, является экземпляром данного типа. Любой экземпляр типа XML(CONTENT(ANY)) одновременно принадлежит к типу XML(SEQUENCE). Значения этого типа не обязательно хорошо сформированы. Они могут являться промежуточными результатами запроса, которые позже будут приводиться к хорошо сформированному виду.

» XML(CONTENT(UNTYPED)). Этот тип еще более ограничивающий, чем XML(CONTENT(ANY)). Таким образом, любое значение типа XML(CONTENT(UNTYPED)) также является экземпляром типа XML(CONTENT(ANY)) и типа XML(SEQUENCE). Любое XML-значение, которое является либо пустым (NULL), либо непустым значением типа XML(CONTENT(ANY)), представляет собой узел *D* документа XQuery, причем такой, что следующие утверждения справедливы для каждого узла-элемента XQuery, содержащегося в XQuery-дереве *T* с корнем в *D*:

- свойство type-name имеет тип xdt:untyped;
- свойство nilled равно False;
- для каждого узла-атрибута, содержащегося в дереве *T*, свойство type имеет тип xdt:untypedAtomic;
- для каждого узла-атрибута, содержащегося в дереве *T*, свойство type принимает значение свойства type-name.

» XML(CONTENT(XMLSCHEMA)). Это еще один подтип XML(CONTENT(ANY)), помимо XML(CONTENT(UNTYPED)), а также подтип XML(SEQUENCE). Любое XML-значение, которое является либо пустым, либо непустым значением типа XML(CONTENT(ANY)), также представляет собой узел *D* документа XQuery, причем каждый узел-элемент XQuery, содержащийся в XQuery-дереве *T* с корнем в *D*, также является:

- действительным в соответствии с XML-схемой *S* (XML Schema), или

- действительным согласно XML-пространству имен *N* в XML-схеме *S*, или
- действительным согласно компоненту E XML-схемы *S* при глобальном объявлении элементов, или
- значением типа XML(CONTENT(XMLSCHEMA)), дескриптор типа которого включает зарегистрированный дескриптор XML-схемы *S* и (если пространство имен *N* задано) XML-пространство имен URI пространства *N* или (если компонент *E* задан) XML-пространство имен URI компонента *E* и NCName (non-colonized name — имя без двоеточий) компонента *E*.

» XML(DOCUMENT(ANY)). Это еще один подтип типа XML(CONTENT(ANY)) с дополнительным ограничением: экземпляры этого типа должны быть узлами документа, которые имеют ровно один узел-элемент XQuery, нулевое или положительное количество узлов-комментариев XQuery и нулевое или положительное количество узлов-инструкций обработки XQuery.

» XML(DOCUMENT(UNTYPED)). Любое значение, являющееся либо пустым (NULL), либо непустым значением типа XML(CONTENT(UNTYPED)), т.е. узлом документа XQuery, свойство children которого имеет ровно один узел-элемент XQuery, нулевое или положительное количество узлов-комментариев XQuery и нулевое или положительное количество узлов-инструкций обработки, представляет собой значение типа XML(DOCUMENT(UNTYPED)). Все экземпляры типа XML(DOCUMENT(UNTYPED)) одновременно являются экземплярами типа XML(CONTENT(UNTYPED)). Более того, все экземпляры типа XML(DOCUMENT(UNTYPED)) одновременно являются экземплярами типа XML(DOCUMENT(ANY)). Тип XML(DOCUMENT(UNTYPED)) является самым строгим (ограничивающим) среди всех остальных подтипов, разделяя все их ограничения. Любой документ, определяемый типом XML(DOCUMENT(UNTYPED)), одновременно является экземпляром всех остальных подтипов XML.

Тип ROW

Тип данных ROW (запись) впервые появился в SQL:1999. Его нельзя назвать простым для понимания, и в процессе изучения SQL вы, возможно, так никогда с ним и не столкнетесь. В конце концов, с 1986 по 1999 годы люди прекрасно обходились без него.

Примечательно, что тип данных ROW нарушает правила нормализации, которые доктор Кодд изложил в теории реляционных баз данных. Эти правила будут подробно описаны 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 позволяет объявить целую строку данных, находящуюся в одном поле одной строки таблицы, — другими словами, объявить запись, вложенную в другую запись.

Нормальные формы, введенные доктором Коддом, определяют характеристики реляционных баз данных. Включение типа ROW в стандарт SQL стало первой попыткой выхода SQL за пределы стандартной реляционной модели.

Проанализируйте следующую инструкцию SQL, в которой для персональной адресной информации (улица, город, штат, почтовый код) используется 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 установлен, 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 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 задавать только один раз — в определении типа ROW.

Типы коллекций

После того как с выходом стандарта SQL:1999 была разрушена реляционная строгость, стало возможным использование типов данных, которые нарушают первую нормальную форму. Теперь поля получили возможность содержать не один объект, а целую их коллекцию. Тип ARRAY (массив) впервые появился уже в SQL:1999, а тип MULTISET (мультимножество) — в SQL:2003.

Две коллекции можно сравнить между собой только в том случае, если они относятся к одному и тому же типу данных (либо ARRAY, либо MULTISET) и имеют сравнимые типы элементов. Поскольку массивы имеют определенный порядок элементов, соответствующие элементы массивов можно сравнивать. Мультимножества такого порядка не имеют, однако их тоже можно сравнить, но только если в обоих сравниваемых мультимножествах существует нумерация и она у них совпадает.

Тип ARRAY

Тип данных ARRAY (массив) нарушает первую нормальную форму (1НФ), но иным образом, чем тип ROW. Тип ARRAY, относящийся к типу коллекций, в действительности не является отдельным типом данных в том же смысле, что и тип CHARACTER или NUMERIC. Тип ARRAY просто позволяет одному из других типов иметь множество значений в одном поле таблицы. Например, вашей организации важно поддерживать контакт со своими клиентами, где бы они ни находились: на работе, дома или в дороге. Поэтому данные о клиенте могут включать и несколько телефонных номеров. Это можно реализовать, объявив атрибут Phone (телефон) массивом, как показано в следующей инструкции.

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

Выражение ARRAY [3] позволяет хранить в таблице CUSTOMER (клиент) до трех телефонных номеров для каждого клиента. Три телефонных номера являются примером повторяющейся группы. Если следовать классической теории реляционных баз данных, то такие группы недопустимы, однако это лишь один из нескольких примеров нарушения правил в стандарте SQL:1999. Когда доктор Кодд впервые опубликовал свои правила нормализации, то ради целостности данных он пожертвовал функциональной гибкостью. В стандарте SQL:1999 утраченная гибкость частично была восстановлена, но за счет определенного 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* позицией в массиве.

Массив — это упорядоченная коллекция значений, а их количество называют *мощностью* массива. Любой SQL-массив может характеризоваться мощностью в диапазоне от нуля до некоторого объявленного максимума. Это означает, что мощность столбца, имеющего тип массива, может меняться от строки к строке. Массив может быть атомарно нулевым, и тогда его мощность будет выражена значением `null`. Однако не путайте нулевой массив с пустым, мощность которого равна нулю. Массив, который содержит только элементы `null`, будет иметь положительную мощность (больше нуля). Например, мощность массива с пятью элементами `null` равна пяти.

Если мощность массива меньше объявленного максимума, то неиспользуемые ячейки в таком массиве считаются несуществующими. Нельзя считать, что они содержат значения `null`, — этих элементов вообще нет.

Чтобы получить доступ к отдельным элементам массива, достаточно заключить их индексы в квадратные скобки. Например, если у вас есть массив `Phone`, то выражение `Phone [3]` будет указывать на третий элемент массива.

Стандарт SQL:1999 позволил определять мощность массива с помощью функции `CARDINALITY`, а SQL:2011 — максимальную мощность массива с помощью функции `ARRAY_MAX_CARDINALITY`. Это очень полезный инструмент, поскольку он позволяет создавать программные процедуры общего назначения, которые применимы к массивам с различными максимальными мощностями. Программные процедуры, использующие жестко запрограммированные максимальные мощности, работают только с массивами, которые характеризуются заданными значениями максимальной мощности. Их придется переписывать для массивов с любой другой максимальной мощностью.

Несмотря на то что в стандарте SQL:1999 появился тип данных `ARRAY` и стал возможным доступ к отдельным элементам массива, в стандарте не была предусмотрена возможность удалять элементы из массива. Это упущение было исправлено в стандарте SQL:2011 благодаря функции `TRIM_ARRAY`, которая позволяет удалять элементы с конца массива.

Тип MULTISET

Тип данных `MULTISET` (мультимножество) — это неупорядоченная коллекция. Невозможно обеспечить адресацию конкретных элементов мультимножества, поскольку им не назначаются определенные порядковые позиции.

Типы REF

Типы `REF` не входят в ядро SQL. Это значит, что СУБД, не использующая эти типы данных, все равно считается соответствующей стандарту SQL. Тип `REF` не считается отдельным типом данных в том смысле, как это понимается для типов `CHARACTER` и `NUMERIC`. Тип `REF` — это размещаемый в строке таблицы

указатель на элемент данных, на данные типа ROW или на данные абстрактного типа. Выполняя операцию разыменования указателя, можно получить значение, на которое он ссылается.

Не стоит волноваться, если вам что-то пока непонятно, — в этом вы не одиноки. Использование типов REF требует глубокого знания принципов объектно-ориентированного программирования. В книге мы не будем погружаться в эти дебри. Поскольку типы REF не входят в ядро SQL, вам лучше совсем их не использовать. Если вы стремитесь к максимальной переносимости между разными СУБД, то не следует выходить за рамки ядра SQL.

Пользовательские типы

Пользовательские типы (user-defined type — UDT) — это еще один пример новых возможностей, вошедших в стандарт SQL:1999 из мира объектно-ориентированного программирования. Как программист на SQL вы больше не ограничены теми типами данных, которые определены в спецификации языка. У вас есть возможность создавать собственные типы, используя принципы построения абстрактных типов данных (в таких объектно-ориентированных языках программирования, как C++).

Одним из самых важных преимуществ UDT является то, что их можно использовать для устранения нестыковок между SQL и той языковой оболочкой, в которую инкапсулирован SQL (т.е. языком реализации приложения баз данных). Застарелая проблема SQL заключается в том, что его предопределенные типы данных могут не соответствовать типам, используемым в языках-оболочках, в которые встроены инструкции SQL. Теперь же программист баз данных может создавать в SQL такие типы данных, которые поддерживают это соответствие.

Типы UDT имеют инкапсулированные атрибуты и методы. Пользователю доступны определения атрибутов и результаты выполнения методов, но конкретные механизмы этого выполнения от него скрыты. Доступ к атрибутам и методам можно еще больше ограничить, указав, что они являются открытыми, закрытыми или защищенными.

- ***Открытые* (public)** атрибуты или методы доступны всем пользователям UDT.
- ***Закрытые* (private)** атрибуты или методы доступны только самому типу UDT.
- ***Защищенные* (protected)** атрибуты или методы доступны только самому типу UDT и его подтипам.

Таким образом, в SQL тип UDT ведет себя во многом так же, как и класс из объектно-ориентированного языка программирования. Существуют две разновидности определяемых пользователем типов: индивидуальные и структурированные.

Индивидуальные типы данных

Индивидуальный тип данных — это более простая форма из двух определяемых пользователем типов. Его характерная особенность состоит в том, что он выражается как один конкретный тип данных. Он создается на основе одного из ранее определенных типов данных, называемых *исходными*. Разные индивидуальные типы, которые созданы на основе одного исходного типа, отличаются друг от друга, поэтому непосредственно сравнивать их между собой нельзя. К примеру, индивидуальные типы можно использовать, чтобы различать валюты разных стран. Рассмотрим следующее определение типа:

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

В результате этого определения на основе уже существующего типа `DECIMAL` создан новый тип данных `USdollar`, предназначенный для долларов США. Аналогичным образом можно создать другой 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.

```
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 экземпляры первого типа нельзя сравнивать с экземплярами второго, равно как и с

экземплярами типа DECIMAL. Теперь в SQL, как и в реальном обменном пункте, можно конвертировать доллары в евро, но для этого потребуется специальная операция преобразования (CAST). И только после такого преобразования можно выполнять операции сравнения экземпляров разных типов.

Структурированные типы

Вторая разновидность определяемого пользователем типа — структурированный тип. Он представляет собой перечень определений атрибутов и методов, а не базируется на каком-то одном конкретном исходном типе.

Конструкторы

При создании структурированного UDT-типа СУБД автоматически создает для него функцию-конструктор с именем, совпадающим с именем UDT-типа. Задача конструктора состоит в инициализации атрибутов UDT значениями по умолчанию.

Мутаторы и наблюдатели

При создании структурированного UDT-типа СУБД также автоматически создает для него *функцию-мутатор* и *функцию-наблюдатель*. Первая изменяет значение атрибута структурированного типа, а вторая, наоборот, извлекает значение атрибута структурированного типа. Вы можете включать функции-наблюдатели в запросы SELECT, чтобы извлекать нужные значения из баз данных.

Подтипы и супертипы

Между двумя структурированными типами могут существовать иерархические отношения. Например, тип MusicCDudt (музыка) имеет подтипы RockCDudt (рок) и ClassicalCDudt (классика). Для этих двух подтипов MusicCDudt является супертипом. RockCDudt является *собственным подтипом* MusicCDudt при условии, что не существует никакого другого подтипа MusicCDudt, который был бы супертипом для RockCDudt. Если RockCDudt имеет подтип HeavyMetalCDudt (хеви-метал), то HeavyMetalCDudt также является подтипом MusicCDudt, но не собственным.

Структурированный тип, не имеющий супертипа, называется *максимальным супертипом*, а структурированный тип, не имеющий подтипов, — *листовым* (т.е. конечным) подтипом.

Пример структурированного типа

Структурированные UDT-типы могут быть созданы следующим способом.

```
/* Создаем UDT-тип с именем MusicCDudt */
CREATE TYPE MusicCDudt AS
/* Задаем атрибуты */
Title          CHAR(40),
```

```
Cost           DECIMAL(9, 2),
SuggestedPrice DECIMAL(9, 2)
/* Вводим 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.

```
CREATE TABLE METALSKU (
   Album    HeavyMetalCDudt,
   SKU      INTEGER
) ;
```

Теперь можно добавить строки в новую таблицу.

```
BEGIN
   /* Объявляем временную переменную */
   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
```

Пользовательские типы, созданные на базе типов коллекций

Выше было показано, как можно создать определяемый пользователем тип из уже существующего типа (см. пример создания типа `USDollar` из типа `DECIMAL`). Эта возможность впервые была реализована в SQL:1999, а стандарт SQL:2011 пошел еще дальше, и теперь можно создать новый UDT-тип на базе типа коллекции. Это расширение позволяет разработчику определять методы, применимые к массиву целиком, а не только к его отдельным элементам, как это разрешалось делать в стандарте SQL:1999.

Перечень типов данных

В табл. 2.2 перечислены различные типы данных и показаны примеры литералов, соответствующих 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
BINARY (1)	'01100011'
VARBINARY (4)	'011000111100011011100110'
BLOB (1000000)	'1001001110101011010101010101...'
BOOLEAN	'TRUE'
DATE	DATE '1957-08-14'
TIME (2) WITHOUT TIME ZONE[1]	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' WITHOUT TIME ZONE
TIMESTAMP WITH TIME ZONE (0)	TIMESTAMP '1957-08-14 12:46:02-08:00' WITH TIME ZONE
INTERVAL DAY	INTERVAL '4' DAY
XML(SEQUENCE)	<Client>Иван Сергеев</Client>
ROW	ROW (Street VARCHAR (25), City VARCHAR (20), Stat CHAR (2), PostalCode VARCHAR (9))
ARRAY	INTEGER ARRAY [15]
MULTISET	К типу MULTISET литералы не применимы
REF	Это не тип, а указатель
USER DEFINED TYPE	Тип валюты, созданный на основе DECIMAL

[1] *Аргумент задает количество цифр в дробной части числа.*

Ваша реализация SQL может не поддерживать все типы данных, перечисленные в этом разделе. Более того, в ней могут поддерживаться нестандартные типы, которые здесь не описаны. (Вы уже поняли, о чем идет речь, и знаете, что именно нужно искать в справочной системе.)

Пустые значения

Если в поле базы данных находятся какие-то данные, то в этом поле существует определенное значение. Если же поле не содержит никаких данных, то говорят, что у него пустое, или отсутствующее, значение (его еще называют *неопределенным*). Следует иметь в виду, что:

- пустое значение (`null`) в числовом поле — не то же самое, что число 0;
- пустое значение в символьном поле — не то же самое, что пустая строка.

Нуль и пустая строка являются определенными значениями, а наличие пустого значения в поле говорит о том, что в него вообще не занесено какое бы то ни было значение, т.е. в общем случае оно неизвестно.

Существует множество ситуаций, в которых поле может иметь пустое значение. Ниже приведены некоторые из этих случаев и даны примеры каждого из них.

- Значение существует, но вам оно пока неизвестно. До того как астрономы нашли четкое доказательство существования жизни за пределами Солнечной системы в галактике Млечный путь, вы записали пустое значение в поле `NUMBER` строки `Lifeforms` (формы жизни) таблицы `Exoplanets` (экзопланеты).
- Значение пока что не существует. Установим неопределенное значение в поле `TOTAL_SOLD` (всего продано) строки `"SQL для чайников"` таблицы `BOOKS` (книги), так как первые данные о продажах за квартал еще не поступили.
- Значение для данного поля строки неприменимо. Имеет смысл установить неопределенное значение в поле `SEX` (пол) строки `C3PO` таблицы `EMPLOYEE` (наемный работник), так как `C3PO` — это андроид, у которого нет пола.
- Значение выходит за пределы установленного диапазона. В поле `SALARY` (зарплата) строки `Илон Маск` таблицы `EMPLOYEE` вы

установили неопределенное значение, так как для этого поля вы сами задали тип `NUMERIC (8, 2)`, а оклад, предусмотренный в контракте Илона Маска, превышает 999 999,99 доллара.


СОВЕТ

Поле может иметь пустое значение по самым разным причинам, так что не торопитесь с выводами относительно того, что именно означает конкретное пустое значение.

Ограничения

Ограничения устанавливаются на данные, вводимые кем-либо в таблицу базы данных. Например, известно, что значения, вводимые в некоторый числовой столбец, должны находиться в пределах заданного диапазона. Если же кто-то вводит число, не попадающее в этот диапазон, то такой ввод будет ошибочным. От подобных ошибок и защищает установка ограничения для столбца — разрешается вводить в него только значения из заданного диапазона.

Традиционно сложилось так, что если приложение использует базу данных, то оно и накладывает на эту базу ограничения. Однако в последних версиях СУБД у вас есть возможность устанавливать ограничения на данные непосредственно из СУБД. Этот подход дает ряд 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 для этого достаточно добавить соответствующее предложение в инструкцию `CREATE`.

Об ограничениях и *утверждениях*, которые тоже являются ограничениями, но применяются к нескольким таблицам, мы подробно поговорим в главе 5.

Использование SQL в архитектуре "клиент/сервер"

SQL — это подъязык данных, который применяется в автономных или многопользовательских системах. Особенно хорошо SQL зарекомендовал себя в рамках архитектуры "клиент/сервер". В такой системе пользователи работают на множестве клиентских машин, подключенных к одному или нескольким серверам. При этом пользователи могут иметь доступ — через локальную сеть или другие каналы связи — к базе данных, расположенной на сервере, с которым соединены их компьютеры. В приложении, работающем на клиентском

компьютере, создаются инструкции SQL. Та часть СУБД, которая находится на клиентском компьютере, передает эти инструкции на сервер по каналу связи, соединяющему сервер с клиентом. А другая часть СУБД, которая находится на сервере, интерпретирует и выполняет полученную инструкцию SQL, а затем по каналу связи отправляет результаты назад клиенту. В SQL-запросе можно закодировать очень сложные операции, а затем на сервере декодировать их и выполнить. При такой организации работы пропускная способность канала связи используется наиболее эффективно.

Если в системе "клиент/сервер" вы получаете данные с помощью SQL-инструкций, то по каналу связи от сервера на клиентский компьютер будут передаваться только нужные вам данные. И наоборот, простая система с разделением ресурсов и с минимальным "интеллектом" сервера вынуждена гонять туда-сюда по каналу связи огромные массивы данных — и все это ради того, чтобы вы смогли получить крохотный объем нужной информации. Не стоит и говорить, что такие пересылки избыточных данных могут сильно замедлить работу. Архитектура "клиент/сервер", дополняя возможности SQL, позволяет в малых, средних и крупных сетях получить хорошую производительность при умеренных расходах.

Сервер

Сервер не делает ничего, пока не получит запрос от клиента. Он находится в состоянии ожидания. 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 наискорейший доступ к данным и оперативную их передачу. Но так как им приходится обрабатывать трафик, идущий одновременно от множества клиентских машин, они должны быть оборудованы очень быстрыми многоядерными процессорами.

Что такое сервер

Сервер (в данном случае *сервер баз данных*) является той частью системы "клиент/сервер", где находится база данных. Кроме того, на нем находится серверная часть СУБД, которая интерпретирует инструкции, полученные от клиентов, а затем преобразует их в операции, выполняемые в базе данных. После этого серверные программы форматируют результаты выполнения запроса и отправляют эти результаты клиенту, от которого пришел данный запрос.

Что делает сервер

Задача сервера баз данных относительно проста и понятна. Все, что ему нужно делать, — это читать, интерпретировать и выполнять инструкции,

поступающие по сети от клиентов. Эти инструкции должны быть написаны на одном из существующих подъязыков данных.

Подъязык нельзя считать полноценным языком — он выполняет только часть функций языка. Подъязык данных занят только обработкой данных. В нем имеются операции добавления, обновления, удаления и извлечения данных, но нет таких управляющих структур, как циклы, локальные переменные, функции или операции ввода-вывода на различные устройства. Из имеющихся сегодня подъязыков данных самым известным является SQL, и он уже стал промышленным стандартом. SQL вытеснил патентованные подъязыки данных на всех классах машин. С появлением версии SQL:1999 стандарт SQL пополнился новыми возможностями, которые отсутствуют у традиционных подъязыков. Однако SQL все еще не является полноценным языком программирования общего назначения. Поэтому, чтобы создать приложение, работающее с базой данных, его следует использовать совместно с языком-оболочкой.

Клиент

Клиентская часть системы "клиент/сервер" 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, которая используется на сервере, или даже идентична ей. Клиент отличается от сервера в основном программным обеспечением.

Что такое клиент

Основная задача клиента состоит в поддержке пользовательского интерфейса. С точки зрения пользователя клиентской машиной является его компьютер, а приложением — пользовательский интерфейс. Пользователь может даже и не знать, что в процессе участвует сервер. Обычно сервер не виден — он может находиться в другой комнате или в другом здании. Но кроме пользовательского интерфейса, в состав клиента входит еще и прикладная программа, а также клиентская часть СУБД. Приложение выполняет нужное вам задание, например оформляет заказы. Клиентская часть СУБД выполняет команды приложения и обменивается с сервером данными и инструкциями 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 может обрабатывать данные, поступающие по каналу связи или от других компьютеров сети. Весь "интеллект"

приложения сосредоточен в клиентской части СУБД. Эта клиентская часть как раз и интересует разработчиков. Ведь в серверной части всего лишь реализована монотонная, механическая обработка запросов, поступивших от клиента.

Использование SQL в Интернете и локальной сети

Работа с базами данных по Интернету (в наши дни — в "облаке") или в локальной сети кардинально отличается от работы в традиционной системе "клиент/сервер". Основное различие сосредоточено в клиентской части. В традиционной системе "клиент/сервер" большинство функций СУБД выполняется на клиентской машине. А в базе данных, работающей по Интернету, основная часть системы (если не вся) находится на сервере. В клиентской же части может не быть ничего, кроме веб-браузера и его расширения (например, это может быть встраиваемый модуль Firefox или элемент управления ActiveX). Таким образом, "интеллект" системы в этом случае концентрируется на сервере. Такая концентрация имеет несколько преимуществ:

- достигается дешевизна клиентской части системы (браузера);
- используется стандартный пользовательский интерфейс;
- клиентскую часть легко поддерживать;
- используется стандартная связка "клиент — сервер";
- имеются стандартные средства вывода мультимедийных данных.

В число главных недостатков работы с базами данных по Интернету входят проблемы защиты и целостности данных.

- Чтобы защищать информацию от несанкционированного доступа или повреждения, на веб-сервере и в клиентском браузере необходимо 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, существуют клиентские и серверные расширения, предназначенные для решения этих проблем. Благодаря этим расширениям Интернет вполне подходит для установки приложений, работающих с базами данных. Архитектура локальных сетей подобна используемой в Интернете, но вопрос

защиты в них не является предметом такого сильного беспокойства. Поскольку в организациях, где функционирует корпоративная сеть, физически контролируются все клиентские компьютеры, а также серверы и сама локальная сеть, соединяющая эти компоненты, система не в такой мере открыта для хакеров. Впрочем, и во внутренних сетях есть еще над чем работать: предметом особого внимания остаются ошибки, допускаемые при вводе данных, и не теряет актуальности проблема нарушения синхронизации в базах данных.

Глава 3

Компоненты SQL

В ЭТОЙ ГЛАВЕ...

- Создание баз данных
- Обработка данных
- Защита баз данных

SQL — это язык, предназначенный для работы с информацией, хранящейся в реляционных базах данных. И хотя поставщики СУБД предлагают свои реализации SQL, развитие самого языка определяется и контролируется стандартом ISO/IEC (переработанным в 2016 году). Все реализации в большей или меньшей степени отличаются от стандарта. Но как можно более точное следование стандарту — главное условие для работы базы данных и связанных с ней приложений на нескольких (а не только на одной) платформах.

SQL не является языком программирования общего назначения, но обладает рядом достаточно мощных средств. Все необходимые действия по созданию, изменению, поддержке базы данных и обеспечению ее безопасности выполняются с помощью трех подъязыков, входящих в состав SQL.

- **Язык определения данных (DDL).** Это часть SQL, которая используется для создания (полного определения) базы данных, изменения ее структуры и удаления после того, как база становится ненужной.
- **Язык манипулирования данными (DML).** Предназначен для поддержки базы данных. С его помощью можно точно указать, что именно нужно сделать с данными, находящимися в базе: добавить, изменить, удалить или извлечь.
- **Язык управления данными (DCL).** Предназначен для защиты базы данных от различных повреждений. При правильном

использовании DCL обеспечивает безопасность базы, а степень защищенности зависит от используемой реализации. Если реализация не обеспечивает достаточной защиты, то довести безопасность до нужного уровня необходимо при разработке приложения.

Задача этой главы — ознакомить вас с языками DDL, DML и DCL.

Язык определения данных

Язык определения данных (Data Definition Language — DDL) представляет собой ту часть SQL, которая используется для создания, изменения и удаления основных элементов реляционной базы данных. В число этих элементов могут входить таблицы, представления, схемы, каталоги, кластеры и, возможно, не только они. В данном разделе рассматривается иерархия вложенности, которая связывает между собой эти элементы, а также описываются инструкции, выполняемые над элементами базы данных.

В главе 1 уже упоминались таблицы и схемы, а также говорилось о том, что *схема* — это определение общей структуры, в состав которой входят таблицы. Таким образом, таблицы и схемы в реляционной базе данных являются двумя элементами *иерархии вложенности*. Эту иерархию можно представить следующим образом:

- таблицы состоят из строк и столбцов;
- схемы состоят из таблиц и представлений;
- схемы находятся в каталогах.

Сама же база данных состоит из каталогов. В некоторых источниках базу данных называют *кластером*. Мы еще поговорим о кластерах в разделе, посвященном использованию каталогов.

Когда "Просто сделай это!" — не лучший совет

Предположим, вы собрались создать базу данных для своей организации. Окрыленный перспективой создать полезную и ценную структуру, очень важную для будущего вашей компании, вы садитесь за компьютер и начинаете вводить SQL-инструкции `CREATE`. Не так ли?

Ну ладно, не совсем так. Фактически это было бы предпосылкой к возникновению последующих проблем. Множество проектов баз данных запутывается с самого начала, поскольку волнение и энтузиазм мешают хладнокровному планированию. Даже обладая четким представлением о структуре своей базы данных, *отобразите ее на бумаге*, а потом уже садитесь за клавиатуру.

Вот где разработка базы данных начинает напоминать шахматы. В сложной шахматной партии вы можете увидеть ход, который кажется выигрышным. Но сделав его поспешно, вы в итоге обнаруживаете, что он ведет к поражению. Поэтому гроссмейстеры советуют новичкам (в шутку) играть, пряча руки под стол. Это воспрепятствует нервному желанию сделать очевидный ход побыстрее. Если вы будете изучать позицию чуть дольше, то, вероятно, найдете лучший ход или даже предупредите блестящий встречный ход, который может сделать ваш противник. Создание базы данных без глубокого предварительного анализа может привести к структуре, которая в лучшем случае окажется неоптимальной. В худшем случае она будет способствовать нарушению целостности данных. Прятать руки под стол, вероятно, не поможет, но попробуйте все же взять карандаш и нарисовать план базы данных на бумаге.

При планировании базы данных следуйте приведенным ниже советам.

- Перечислите все таблицы.
- Определите столбцы, которые должна содержать каждая таблица.
- Присвойте каждой таблице *первичный ключ*, который гарантирует уникальность (первичные ключи обсуждаются в главах 4 и 5).
- Удостоверьтесь, что у каждой таблицы в базе данных есть по крайней мере один столбец, совпадающий со столбцом хотя бы в одной другой таблице базы данных. Эти общие столбцы послужат логическими связками, которые позволят вам устанавливать отношения между данными одной таблицы с соответствующими данными другой.
- Приведите каждую таблицу как минимум в *третью нормальную форму* (3НФ), чтобы предотвратить аномальные операции вставки, удаления и модификации (нормализацию базы данных мы обсудим в главе 5).

Закончив проектирование базы данных на бумаге и проверив его правильность, можно переходить к проектированию на компьютере. Это можно сделать путем ввода SQL-инструкций `CREATE`. Но, скорее всего, для создания элементов базы данных вы воспользуетесь графическим интерфейсом своей СУБД. С его помощью вводимые вами команды будут незаметно преобразованы в надлежащие SQL-инструкции средствами самой СУБД.

Создание таблиц

Таблица базы данных представляет собой двухмерный массив, состоящий из строк и столбцов. Создать таблицу можно с помощью инструкции `CREATE TABLE`, указав имя и тип данных каждого столбца.

После того как таблица будет создана, ее можно заполнять данными (впрочем, загружать данные — это дело инструкций DML, а не DDL). Если требования изменяются, то поменять структуру уже созданной таблицы можно с помощью инструкции `ALTER TABLE`. Со временем таблица может стать бесполезной или устареть. И когда окажется, что "час пробил", уничтожить таблицу можно будет с помощью инструкции `DROP`. Различные варианты инструкций `CREATE`, `ALTER` и `DROP` и составляют основу языка DDL.

Предположим, вы проектируете базу данных и не хотите, чтобы ее таблицы со временем (по мере обновления имеющихся в них данных) стали слишком громоздкими. Для поддержки целостности данных обычно принимается решение нормализовать таблицы.

Нормализация, которая сама по себе является широким полем для исследования, — это способ создания такой структуры таблиц базы данных, в которой обновление данных не создавало бы аномалий. В каждой создаваемой таблице столбцы соответствуют атрибутам, которые тесно связаны между собой.

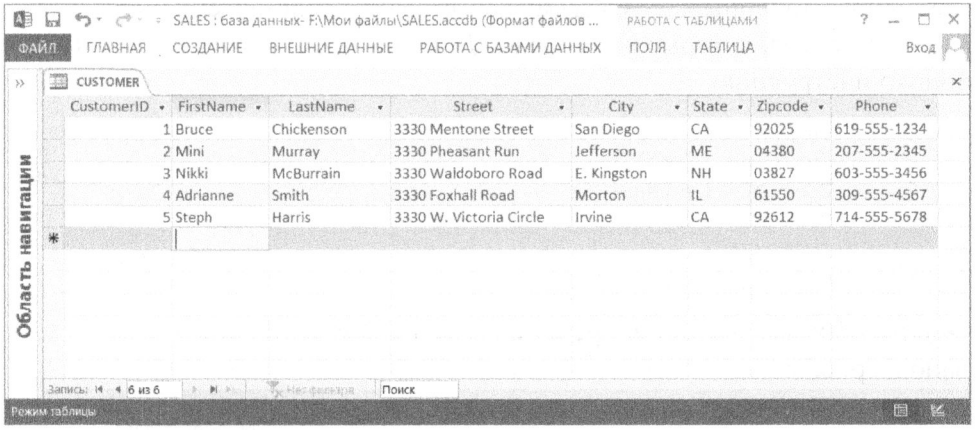
К примеру, можно создать таблицу `CUSTOMER` (клиент) с такими атрибутами: `CUSTOMER.CustomerID` (идентификатор клиента), `CUSTOMER.FirstName` (имя), `CUSTOMER.LastName` (фамилия), `CUSTOMER.Street` (улица), `CUSTOMER.City` (город), `CUSTOMER.State` (штат), `CUSTOMER.Zipcode` (почтовый индекс) и `CUSTOMER.Phone` (телефон). Все эти атрибуты имеют отношение к описанию клиента, а не какого-либо другого объекта. В них находится относительно постоянная информация о клиентах вашей организации.

В большинстве СУБД предусмотрены графические инструменты для создания таблиц. В то же время таблицы можно создавать и с помощью инструкций SQL. Например, ниже приведена инструкция, при выполнении которой создается таблица `CUSTOMER`.

```
CREATE TABLE CUSTOMER (
   CustomerID   INTEGER      NOT NULL,
   FirstName    CHAR (15),
   LastName     CHAR (20)    NOT NULL,
   Street       CHAR (25),
   City         CHAR (20),
   State        CHAR (2),
   Zipcode      CHAR (10),
   Phone        CHAR (13)
) ;
```

Для каждого столбца указываются его имя (например, `CustomerID`), тип данных (например, `INTEGER`) и, возможно, одно или несколько ограничений, таких как `NOT NULL` (недопустимость пустых значений).

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



CustomerID	FirstName	LastName	Street	City	State	Zipcode	Phone
1	Bruce	Chickenson	3330 Mentone Street	San Diego	CA	92025	619-555-1234
2	Mini	Murray	3330 Pheasant Run	Jefferson	ME	04380	207-555-2345
3	Nikki	McBurrain	3330 Waldoboro Road	E. Kingston	NH	03827	603-555-3456
4	Adrianne	Smith	3330 Foxhall Road	Morton	IL	61550	309-555-4567
5	Steph	Harris	3330 W. Victoria Circle	Irvine	CA	92612	714-555-5678

Рис. 3.1. Таблица `CUSTOMER`, которую можно создать с помощью инструкции `CREATE TABLE`

ЗАПОМНИ!

Если используемая вами реализация SQL не соответствует последнему стандарту ISO/IEC SQL, то синтаксис, которым вам придется пользоваться, может не совпадать и с приведенным в книге. За уточнениями обратитесь к документации, прилагаемой к вашей СУБД.

Создание представлений

Иногда из таблицы `CUSTOMER` (клиент) вам потребуется извлечь определенную информацию. Более того, вы не хотите утомлять себя просмотром всего содержимого — вам нужны только конкретные столбцы и строки. В таком случае понадобится представление.

Представления — это виртуальные таблицы. В большинстве реализаций они не являются независимыми физическими объектами. Определения представлений существуют только в метаданных, а сами данные извлекаются из таблиц, на основе которых созданы представления. Эти данные больше нигде не хранятся. Одни представления состоят из столбцов и строк одной таблицы, другие же, которые называют *многотабличными представлениями*, создаются на основе сразу нескольких таблиц.

Создание однотабличного представления

Иногда нужные данные содержатся всего в одной таблице. В таком случае можно создать однотабличное представление. Например, вам нужно просмотреть имена, фамилии и телефонные номера всех клиентов, живущих в штате Нью-Гэмпшир (который обозначается аббревиатурой NH). Тогда на основе таблицы CUSTOMER можно создать представление, содержащее только те данные, которые вам нужны. Для этого достаточно выполнить следующую инструкцию.

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

Каким образом создается представление на основе таблицы CUSTOMER, показано на рис. 3.2.

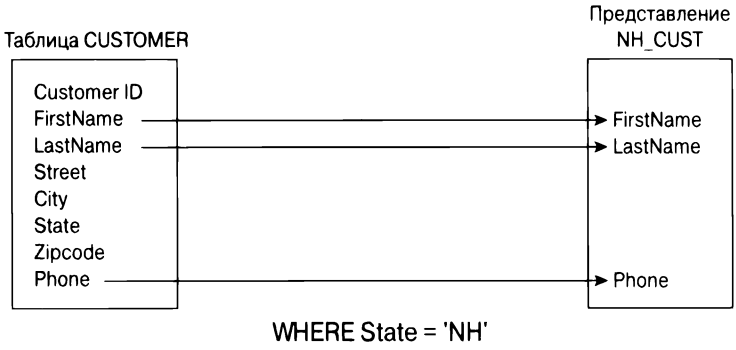


Рис. 3.2. Создание представления NH_CUST из таблицы CUSTOMER

Этот код абсолютно корректен, но несколько избыточен. Ту же задачу можно сформулировать более кратко при условии, что имеющаяся у вас реализация SQL работает, исходя из следующего допущения: если в перечисленных атрибутах не указаны ссылки на таблицу, то все атрибуты относятся к таблице из предложения FROM. Если ваша система принимает подобные условия по умолчанию, то приведенную выше инструкцию можно сократить до следующего вида.

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

Несмотря на то что этот вариант записи проще, подобное представление может неправильно работать после применения инструкций ALTER TABLE. В данном случае ничего плохого не случится, ведь выполняемая инструкция не содержит оператор JOIN. А для представлений с операторами JOIN лучше все же записывать полные имена. (С объединениями вы подробнее ознакомитесь в главе 11.)

Создание многотабличного представления

Чтобы получать ответы на различные вопросы, часто приходится извлекать данные из нескольких таблиц. К примеру, если вы работаете в магазине спорттоваров и для рассылки рекламы по почте вам нужен список клиентов, купивших у вас в прошлом году лыжное снаряжение, то, скорее всего, вам 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 отобразит нужные данные. К созданному представлению можно обращаться снова и снова. И каждый раз представление будет отражать последние изменения в таблицах, на основе которых оно было создано.

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

Таблица 3.1. Таблицы базы данных магазина спорттоваров

Таблица	Столбец	Тип данных	Ограничение
CUSTOMER (клиент)	CustomerID (идентификатор клиента)	INTEGER	NOT NULL
	FirstName (имя)	CHAR (15)	
	LastName (фамилия)	CHAR (20)	NOT NULL
	Street (улица)	CHAR (25)	
	City (город)	CHAR (20)	
	State (штат)	CHAR (2)	
	Zipcode (почтовый индекс)	INTEGER	
	Phone (телефон)	CHAR (13)	
PRODUCT (товар)	ProductID (идентификатор товара)	INTEGER	NOT NULL

Окончание табл. 3.1

Таблица	Столбец	Тип данных	Ограничение
	Name (название)	CHAR (25)	
	Description (описание)	CHAR (30)	
	Category (категория)	CHAR (15)	
	VendorID (идентификатор поставщика)	INTEGER	
	VendorName (наименование поставщика)	CHAR (30)	
INVOICE (счет-фактура)	InvoiceNumber (номер счета-фактуры)	INTEGER	NOT NULL
	CustomerID (идентификатор клиента)	INTEGER	
	InvoiceDate (дата выписки счета-фактуры)	DATE	
	TotalSale (всего продано на сумму)	NUMERIC (9, 2)	
	TotalRemitted (всего оплачено)	NUMERIC (9, 2)	
	FormOfPayment (форма платежа)	CHAR (10)	
INVOICE_LINE (строка счета-фактуры)	LineNumber (номер строки)	INTEGER	NOT NULL
	InvoiceNumber (номер счета-фактуры)	INTEGER	NOT NULL
	ProductID (идентификатор товара)	INTEGER	NOT NULL
	Quantity (количество)	INTEGER	
	SalePrice (цена продажи)	NUMERIC (9, 2)	

Обратите внимание на то, что для некоторых столбцов всех четырех таблиц задано ограничение `NOT NULL` (недопустимость пустых значений). Оно означает, что либо эти столбцы являются первичными ключами соответствующих таблиц, либо вы решили, что существуют 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 мы поговорим в главе 5.)

Таблицы связываются между собой с помощью общих столбцов. Ниже описаны связи между таблицами (рис. 3.3).

- Таблицы `CUSTOMER` и `INVOICE` связывает 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 к одному клиенту.
- Таблицы `INVOICE` и `INVOICE_LINE` также связывает отношение "один ко многим". В счете-фактуре может быть несколько строк, но каждая строка находится в одном и только в одном счете-фактуре.
- И таблицу `PRODUCT` с таблицей `INVOICE_LINE` связывает отношение "один ко многим". Каждый товар может быть указан во многих строках одного или нескольких счетов-фактур. Однако каждая строка относится к одному и только к одному товару.

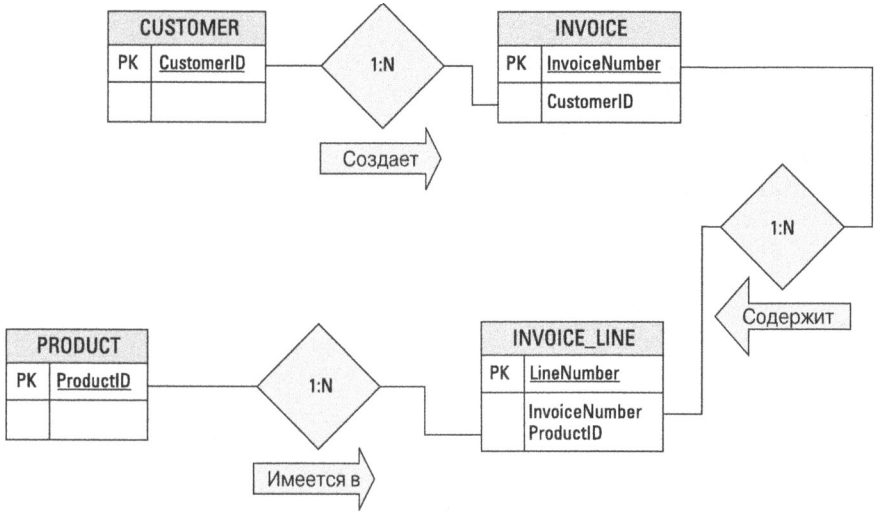


Рис. 3.3. Структура базы данных магазина спорттоваров

Таблица CUSTOMER связана с таблицей INVOICE посредством общего столбца CustomerID. А связь таблиц INVOICE и INVOICE_LINE обеспечивается с помощью общего столбца InvoiceNumber. Таблицы же PRODUCT и INVOICE_LINE связываются через общий столбец ProductID. В сущности, эти связи (отношения) и делают саму базу *реляционной*.

Чтобы получить информацию о тех клиентах, которые купили лыжное снаряжение, необходимы данные из следующих полей: FirstName, LastName, Street, City, State и Zipcode — из таблицы CUSTOMER, Category — из таблицы PRODUCT, InvoiceNumber — из таблицы INVOICE, LineNumber — из таблицы INVOICE_LINE. Нужное представление можно создавать поэтапно, используя показанные ниже инструкции.

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

Приведенные выше инструкции `CREATE VIEW` объединяют данные из нескольких таблиц с помощью оператора `JOIN`. Этот процесс схематически показан на рис. 3.4.

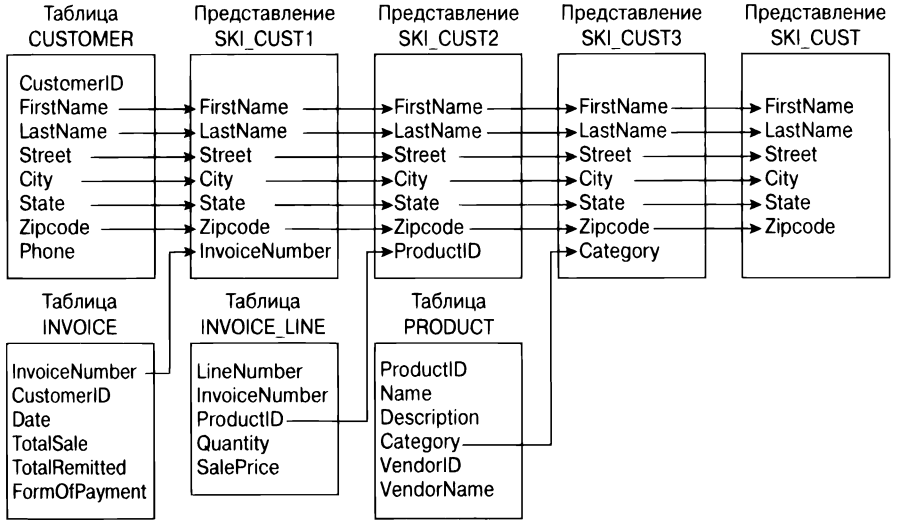


Рис. 3.4. Создание многотабличного представления с помощью оператора `JOIN`

Рассмотрим эти четыре инструкции `CREATE VIEW`.

- Первая инструкция объединяет столбцы из таблицы `CUSTOMER` со столбцом из таблицы `INVOICE` и создает представление `SKI_CUST1`.
- Вторая инструкция объединяет представление `SKI_CUST1` со столбцом из таблицы `INVOICE_LINE`, создавая, таким образом, представление `SKI_CUST2`.
- Третья инструкция объединяет представление `SKI_CUST2` со столбцом из таблицы `PRODUCT` и создает представление `SKI_CUST3`.
- Четвертая инструкция отфильтровывает все строки, в которых поле категории товара не содержит значение `Ski` (лыжи). В результате образуется представление `SKI_CUST`, в котором отображаются имена, фамилии и адреса тех клиентов, которые хотя бы один раз купили товары из 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), находящемуся в инструкции SELECT четвертой инструкции CREATE VIEW. (Операторы JOIN будут подробно рассмотрены в главе 11.)

Многотабличное представление вполне можно создать с помощью одной инструкции SQL. Но если вам показались сложными некоторые или все из приведенных выше инструкций, то представьте, насколько сложной будет одна инструкция, выполняющая все их функции. Я предпочитаю простоту, поэтому по возможности выбираю самый простой способ решения задачи, даже если он и не самый эффективный.

Объединение таблиц в схемы

Таблица состоит из строк и столбцов и соответствует некоторому объекту (например, такому, как список клиентов, товаров и счетов-фактур). Для эффективной работы обычно требуется информация о нескольких объектах, связанных между собой. Таблицы, соответствующие этим объектам, вы располагаете вместе, согласно определенной логической схеме. (*Логическая схема* — это организационная структура совокупности таблиц, связанных между собой отношениями.)


ЗАПОМНИ!

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

В системе, где может сосуществовать несколько не связанных проектов, все таблицы, связанные между собой, можно объединить в схему. А из таблиц, не вошедших в эту схему, можно образовать другие схемы.


СОВЕТ

Чтобы таблицы из одного проекта не оказались случайно в другом проекте, схемам нужно присваивать имена. Каждому проекту присуща собственная схема, имя которой позволяет отличать ее от других схем. Некоторые имена таблиц (например, CUSTOMER, PRODUCT и др.) могут встречаться сразу в нескольких проектах. Если существует хоть малейшая вероятность возникновения путаницы с именами, необходимо при ссылке на таблицу указывать не только ее имя, но и имя схемы (примерно так: *ИМЯ_СХЕМЫ.ИМЯ_ТАБЛИЦЫ*). Если имя схемы не указано явно, то SQL будет считать, что таблица относится к схеме, подразумеваемой по умолчанию.

Заказ по каталогу

Для очень больших баз данных использование даже нескольких схем может оказаться недостаточным. В больших распределенных системах дублирование встречается даже в именах схем. Чтобы этого не было, в SQL предусмотрен еще один уровень иерархии вложенности: каталог. *Каталог* — это именованный набор схем.

Можно *квалифицировать* имя таблицы с использованием имени ее каталога и схемы. Тем самым гарантируется, что никто не перепутает две одноименные таблицы, находящиеся в одноименных схемах. (Почему-то людям трудно придумывать новые имена. И программистам следует учитывать это в своей работе.) Имя таблицы с указанием каталога имеет следующий формат:

```
ИМЯ_КАТАЛОГА.ИМЯ_СХЕМЫ.ИМЯ_ТАБЛИЦЫ
```


СОВЕТ

Верхним уровнем иерархии вложенности базы данных являются кластеры. Впрочем, редко в какой системе нужно создавать полную иерархию вложенности. В большинстве случаев вполне можно ограничиться каталогами. В каталогах содержатся схемы, в схемах — таблицы и представления, а в таблицах и представлениях — столбцы и строки.

Каталог также включает *информационную схему*, в которой содержатся системные таблицы. В системных таблицах хранятся метаданные, отвечающие за связи с другими схемами. В главе 1 база данных была определена как самоописательный набор интегрированных записей. Метаданные в системных таблицах как раз и делают базу данных самоописательной.

Каталоги можно различать по именам, поэтому база данных может содержать множество каталогов. В каждом каталоге, в свою очередь, может быть множество схем, а в каждой схеме — множество таблиц. И конечно же, в каждой таблице может быть множество строк и столбцов. Взаимоотношения в иерархии базы данных представлены на рис. 3.5.

Инструкции DDL

Язык определения данных (DDL) работает со структурой базы данных, в то время как язык манипулирования (он будет описан далее) — с данными, которые находятся в этой структуре. DDL включает три инструкции:

- для создания основных структур базы данных используются разные формы инструкции `CREATE`;
- для изменения созданных структур применяется инструкция `ALTER`;

» для удаления структур, созданных инструкцией CREATE, применяется инструкция DROP.

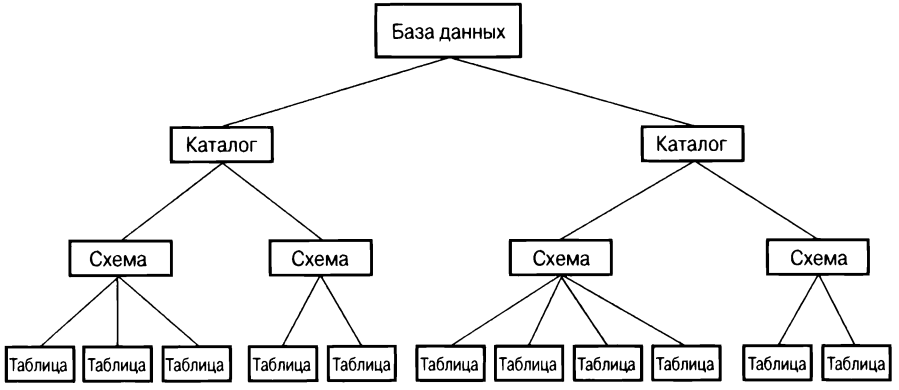


Рис. 3.5. Иерархическая структура типичной реляционной базы данных

Эти DDL-инструкции кратко описаны в следующих разделах, а в главах 4 и 5 будут приведены примеры их использования.

Инструкция CREATE

Инструкция CREATE позволяет создавать объекты SQL нескольких видов, в том числе схемы, домены, таблицы и представления. С помощью инструкции CREATE SCHEMA можно не только создать схему, но и идентифицировать ее владельца, а также определить набор символов, используемый по умолчанию. Например, вот как может выглядеть такая инструкция.

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

С помощью инструкции CREATE DOMAIN устанавливаются ограничения на значения, которые могут содержаться в столбце. Применяемые к домену ограничения определяют, какие объекты могут, а какие не могут в нем находиться. Создавать домены можно после того, как установлена схема. Следующий пример демонстрирует, как можно использовать эту инструкцию.

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

Таблицы создаются с помощью инструкции CREATE TABLE, а представления — с помощью инструкции CREATE VIEW (примеры их использования были приведены выше). При создании новой таблицы с помощью инструкции CREATE TABLE можно одновременно (т.е. в той же инструкции) установить и ограничения на ее столбцы.


СОВЕТ

Впрочем, иногда требуется устанавливать ограничения, которые относятся не только к таблице, но и ко всей схеме. В таких случаях используется инструкция `CREATE ASSERTION` (создать утверждение).

Кроме того, в вашем распоряжении есть инструкции `CREATE CHARACTER SET`, `CREATE COLLATION` и `CREATE TRANSLATION`, которые предоставляют широкие возможности по созданию новых наборов символов, схем сортировки или таблиц трансляции. (*Схемы сортировки* определяют порядок, в котором будут проводиться операции сравнения или сортировки. *Таблицы трансляции* управляют преобразованием символьных строк из одного набора символов в другой.) Используя инструкцию `CREATE`, можно создавать и другие виды объектов (см. табл. 2.1).

Инструкция `ALTER`

Таблица не должна навсегда оставаться такой, какой ее создали изначально. С первого момента ее использования часто обнаруживается, что в ней чего-то не хватает (или, наоборот, есть что-то лишнее). Чтобы изменить таблицу путем добавления, изменения или удаления столбца, воспользуйтесь инструкцией `ALTER TABLE`. Ее можно применять не только к таблицам, но также к столбцам и доменам.

Инструкция `DROP`

Можно легко удалить таблицу из схемы базы данных. Для этого достаточно использовать инструкцию `DROP TABLE <имя_таблицы>`. В результате удаляются все данные этой таблицы, а также метаданные, которые определяют ее в словаре данных, после чего таблицы как будто и не было. И вообще, инструкция `DROP` избавит вас от любого объекта, созданного инструкцией `CREATE`.


ЗАПОМНИ!

Инструкция `DROP` не будет работать при угрозе нарушения ссылочной целостности данных (об этом речь пойдет далее).

Язык манипулирования данными

Как говорилось выше, DDL предназначен для создания, модификации и удаления структур базы данных. Непосредственно с данными DDL-инструкции не работают. Для этого предназначена другая часть SQL: язык манипулирования данными (DML). Одни инструкции DML читаются как обычные предложения на английском языке, и их легко понять; другие, наоборот, могут быть

чрезвычайно сложными — как раз из-за того, что SQL предоставляет возможность работать с данными на уровне очень мелких структурных единиц.

Если DML-инструкция содержит множество выражений, предложений, предикатов или подзапросов, то понять, для чего она предназначена, может оказаться по-настоящему трудным делом. Впрочем, все не так уж страшно. Дело в том, что такие сложные инструкции SQL можно мысленно разбивать на простые части и анализировать одну за другой.

Поддерживаются такие DML-инструкции, как `INSERT`, `UPDATE`, `DELETE`, `MERGE` и `SELECT`. Они могут состоять из разных частей, в том числе из множества предложений. А каждое предложение может включать выражения, логические операторы, предикаты, итоговые функции и подзапросы. Все они позволяют лучше распознавать записи базы данных и извлекать более точную информацию. В главе 6 рассказывается о том, как работают DML-инструкции, а более подробно об их использовании речь пойдет в главах 7–13.

Выражения

Для объединения двух или более значений используются выражения. Имеется несколько разновидностей выражений:

- числовые;
- строковые;
- даты/времени;
- интервальные;
- логические;
- определяемые пользователем;
- записи;
- коллекции.

Логические и определяемые пользователем типы данных, а также записи и коллекции появились в SQL только с выходом стандарта SQL:1999. В некоторых реализациях они вообще еще не поддерживаются. Прежде чем использовать один из этих типов, необходимо убедиться, что он включен в вашу СУБД.

Выражения с числовыми значениями

Для объединения *числовых значений* используйте операторы сложения (+), вычитания (-), умножения (*) и деления (/). Ниже показано несколько примеров выражений с числовыми значениями.

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

Значения в этих примерах являются *числовыми литералами*. В качестве значений можно использовать также имена столбцов, параметры функций, базовые переменные или подзапросы — при условии, что при вычислении выражений их составляющие элементы приводятся к числовым значениям. Вот несколько примеров.

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

Двоеточие в последнем примере говорит о том, что следующая за ним сущность (`months`) является либо параметром, либо базовой переменной.

Выражения со строковыми значениями

В *выражениях со строковыми значениями* можно использовать 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'`
`CITY || ' ' || STATE || ' ' || ZIP`	Одна строка с названиями города и штата, почтовым индексом и с разделителями в виде одиночного пробела


ЗАПОМНИ!

В некоторых реализациях SQL в качестве оператора конкатенации вместо `||` используется знак "плюс" (+). Чтобы уточнить, какой именно оператор конкатенации используется в вашей реализации SQL, обратитесь к соответствующей документации.

Существуют реализации, в которых вместо конкатенации используются строковые операторы, но сам ISO-стандарт SQL эти операторы не поддерживает. К двоичным строкам операция конкатенации применяется так же, как и к текстовым.

Выражения со значениями даты/времени и интервальными значениями

Выражения со значениями даты/времени могут содержать данные типа `DATE`, `TIME`, `TIMESTAMP` и `INTERVAL`. Результатом вычисления таких выражений всегда является некоторое значение даты/времени. К значению этого типа можно применить операцию сложения (или вычитания) с каким-либо интервалом, а также указать часовой пояс.

Ниже приведен пример выражения со значениями даты/времени (название DueDate означает "крайний срок", а INTERVAL '7' DAY — "интервал в 7 дней").

```
DueDate + INTERVAL '7' DAY
```

Такое выражение можно использовать в библиотечной базе данных, чтобы вовремя отправлять напоминание должникам. А вот еще один пример, в котором правда, указывается не дата, а время:

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

СОВЕТ

Фраза AT LOCAL означает, что указано местное время.

Выражения со значениями интервалов работают с промежутками времени, разделяющими два значения даты/времени. Существуют два вида интервалов: *год–месяц* и *день–время*. Их нельзя использовать в одном и том же выражении.

Приведем пример использования интервалов. Предположим, что кто-то возвращает в библиотеку книгу по истечении крайнего срока. Тогда, используя выражение со значениями интервалов (как в следующем примере), можно вычислить, сколько дней прошло после наступления крайнего срока, и назначить соответствующий штраф.

```
(DateReturned - DueDate) DAY
```

Поскольку в качестве интервала может быть использовано значение типа "год–месяц" либо "день–время", следует указать этот тип. В последнем примере с помощью ключевого слова DAY (день) выбран второй тип.

Выражения с булевыми значениями

Выражение с булевыми значениями проверяет, является ли истинным значение заданного предиката. Рассмотрим следующий пример:

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

Если это условие предназначено для выбора строк из таблицы, содержащей список учеников-старшеклассников, то будут получены записи, соответствующие ученикам только 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 такое выражение:

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

Выражения со значениями пользовательских типов

О типах, определяемых пользователями, мы говорили в главе 2. Благодаря такой возможности пользователи могут создавать собственные типы данных, а не довольствоваться теми, которые имеются в арсенале SQL. Если есть выражение, имеющее элементы данных какого-либо пользовательского типа, то значением этого выражения должен быть элемент того же типа.

Выражения со значениями типа ROW

Выражение со значениями типа записи определяет значение целой записи. Последнее может состоять из одного выражения либо из нескольких таких выражений, разделенных запятыми, например:

```
('Джон Смит', 'профессор', 1918)
```

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

Выражения с коллекциями

Значением выражения с коллекцией является массив.

Выражения со ссылочными значениями

Значением выражения со ссылочными значениями является ссылка на другой компонент базы данных, например на столбец таблицы.

Предикаты

Предикаты — это SQL-эквиваленты логических высказываний. В качестве примера высказывания можно привести следующее выражение:

```
"Ученик является старшеклассником"
```

В таблице, содержащей информацию об учениках, домен столбца CLASS (класс) может представлять собой набор таких значений: SENIOR (старшеклассник), JUNIOR (начальные классы), SOPHOMORE (второй курс), FRESHMAN (первый курс) и NULL (неизвестен). Предикат CLASS = SENIOR можно использовать для отсева тех строк, для которых его значение ложно, оставляя, соответственно, только строки с истинным значением этого предиката. Возможна ситуация, когда для какой-то строки значение предиката окажется неизвестным (NULL). В таком случае строку можно или отбросить, или оставить (в зависимости от конкретной ситуации).

Выражение `Class = SENIOR` — это пример предиката сравнения. В SQL предусмотрено шесть операторов сравнения. В простом предикате сравнения используется только один из этих операторов. Предикаты сравнения и примеры их использования приведены в табл. 3.3.

Таблица 3.3. Операторы и предикаты сравнения

Оператор	Проверка	Выражение
=	Равно	`Class = SENIOR`
<>	Не равно	`Class <> SENIOR`
<	Меньше	`Class < SENIOR`
>	Больше	`Class > SENIOR`
<=	Меньше или равно	`Class <= SENIOR`
>=	Больше или равно	`Class >= SENIOR`

В примере таблицы, содержащей информацию об учениках, только первые два выражения (см. табл. 3.3) имеют смысл (`Class = SENIOR` и `Class <> SENIOR`). Это объясняется тем, что в сортировке по умолчанию (т.е. в алфавитном порядке) значение `SOPHOMORE` считается бо́льшим, чем `SENIOR`, так как слово, начинающееся с `SO`, будет стоять после слова, начинающегося с `SE`. Однако такая интерпретация, по всей вероятности, вас совсем не устроит.

Логический оператор

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

- "ученик является старшеклассником";
- "ученику еще нет 14 лет".

Чтобы отделить нужные записи от остальных, можно с помощью логического оператора `AND` (И) создать составной предикат, например:

```
Class = SENIOR AND Age < 14
```

Если используется оператор `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 истинным является лишь один из входящих в него предикатов, то используйте логический оператор OR (ИЛИ). Третьим логическим оператором является NOT (НЕ). Строго говоря, этот оператор не соединяет два предиката. Он применяется к одному предикату и меняет его логическое значение на противоположное. Возьмем для примера следующее выражение:

```
NOT (Class = SENIOR)
```

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

Итоговые функции

Иногда информация, которую вы хотите получить из таблицы, связана с содержимым не отдельных строк, а некоторого их набора. Для таких ситуаций в SQL предусмотрены пять *итоговых (агрегирующих) функций*: COUNT, MAX, MIN, SUM и AVG. Каждая из них выполняет действие над данными, хранимыми в нескольких строках, а не в одной.

Функция COUNT

Функция COUNT возвращает число строк указанной таблицы. Чтобы в базе данных средней школы, используемой в качестве примера, подсчитать количество юных учеников выпускных классов, воспользуйтесь следующей инструкцией.

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

Функция MAX

Функция MAX используется для определения максимального значения в заданном столбце. Скажем, требуется найти самого старшего ученика школы. Строку с нужными данными позволит извлечь следующая инструкция.

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

В результате возвращаются данные обо всех самых взрослых учениках, т.е. если возраст самого старшего ученика равен 18 годам, то при выполнении этой инструкции мы получим данные обо всех 18-летних учениках. В этом запросе используется подзапрос SELECT MAX(Age) FROM STUDENT, который встроен в основной запрос. О подзапросах (также называемых *вложенными запросами*) мы поговорим в главе 12.

Функция 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 выполнения этого запроса мы получим данные о самых младших учениках школы.

Функция SUM

Функция SUM суммирует значения из указанного столбца. Тип заданного столбца должен быть одним 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, установленный для этого типа данных. В таблице INVOICE (счет-фактура) из базы данных, о которой уже говорилось ранее, хранятся данные обо всех продажах. Чтобы найти общую сумму в долларах для всех продаж, зафиксированных в этой базе данных, используйте функцию SUM следующим образом:

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

Функция AVG

Функция AVG возвращает среднее арифметическое всех значений указанного столбца. Как и SUM, функция AVG применяется только к столбцам с числовым типом данных. Чтобы найти среднее арифметическое продаж по всем финансовым операциям, зафиксированным в базе данных, используйте функцию AVG следующим образом:

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

Имейте в виду, что пустые значения (null) в расчет не берутся, так что, если в каких-либо строках столбца TotalSale (всего продано) обнаружатся пустые значения, при вычислении среднего объема продаж эти строки будут проигнорированы.

Функция LISTAGG

Функция LISTAGG, появившаяся в стандарте SQL:2016, преобразует значения из группы табличных строк в список значений, разделенных указанным пользователем разделителем. Эта функция часто применяется для

преобразования значений, находящихся в строках таблицы, в строку значений, разделенных запятыми (comma separated values — CSV), либо в похожий формат, легче воспринимаемый человеком.


ВНИМАНИЕ!

Поскольку функция LISTAGG не экранирует разделители, затруднительно определить, каким является текущий символ: служебным либо частью значения. Поэтому используйте эту функцию только в том случае, если уверены в том, что выбранный символ разделителя не включен в агрегируемые данные.

Функция LISTAGG является функцией упорядоченного агрегирования. Для упорядочения используется предложение WITHIN GROUP. Основной синтаксис таков:

```
LISTAGG (<выражение>, <разделитель>) WITHIN GROUP (ORDER BY ...)
```

Параметр *<выражение>* не может включать оконные функции, агрегирующие функции и подзапросы. Параметр *<разделитель>* должен быть символьным литералом.

Функция LISTAGG удаляет значения NULL перед агрегированием других значений. Если после удаления не остается ненулевых значений, то результатом применения функции LISTAGG будет NULL.

Если применяется ключевое слово DISTINCT, то наравне со значениями NULL удаляются дубликаты значений. При этом используется следующий синтаксис:

```
LISTAGG (DISTINCT <выражение>, <разделитель>) ...
```

Функция LISTAGG, которая появилась в стандарте SQL:2016, является необязательным средством стандартного SQL и до сих пор отсутствует в некоторых реализациях. Если же вы используете реализацию SQL, в которой это средство имеется, то просмотрите документацию на предмет дополнительных предложений, которые могут обеспечить расширенную функциональность.

Подзапросы

Подзапросами называются запросы, находящиеся в другом запросе. В любом месте инструкции SQL, где может стоять выражение, можно использовать и подзапрос. Подзапросы являются мощным инструментом связывания информации из одной таблицы с информацией из другой. Запрос к одной таблице можно встроить (вложить) в другой запрос, применяемый к другой таблице. С помощью вложенных запросов можно получить доступ к информации сразу из нескольких таблиц. Если правильно пользоваться подзапросами, то из базы данных можно извлечь любую нужную информацию. Не стоит беспокоиться о том, сколько уровней подзапросов поддерживает ваша база данных. Когда вы

начнете выстраивать вложенные запросы, уверяю вас, что ваши возможности осмыслить собственные действия иссякнут задолго до того, как база данных исчерпает свои резервы по количеству уровней подзапросов, которое она поддерживает.

Язык управления данными

В языке управления данными (Data Control Language — DCL) существуют четыре инструкции: `COMMIT` (завершить), `ROLLBACK` (откатить), `GRANT` (предоставить) и `REVOKE` (отозвать). Все эти инструкции связаны с защитой базы данных 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 данных. Аппаратный или программный сбой, происшедший во время изменения, может застать базу данных в неопределенном (промежуточном) состоянии — при переходе от состояния на момент начала изменений к состоянию, которое было бы зафиксировано в случае успешного завершения этих изменений.

Для защиты базы данных в SQL предусмотрено ограничение операций, которые могут ее изменить, так что они выполняются только в пределах транзакций. Во время транзакции SQL записывает каждую операцию над данными в журнал изменений (журнал транзакций). Если транзакцию, перед тем как она будет завершена инструкцией `COMMIT`, что-то прервет, вы сможете восстановить первоначальное состояние системы с помощью инструкции `ROLLBACK`. Эта инструкция 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 попытаться выполнить транзакцию повторно.


СОВЕТ

База данных может пострадать из-за сбоев в аппаратном или программном обеспечении. Чтобы свести вероятность такой неприятности к минимуму, современные СУБД стараются "закрыть окно уязвимости" выполнением всех операций с базой данных в пределах транзакции и их завершением в конце этой транзакции. В современных СУБД также ведутся журналы транзакций, в которых регистрируются все изменения. Это позволяет гарантировать защиту данных даже в случае отказов аппаратного и программного обеспечения или человеческих ошибок. После завершения транзакции данные

защищены от всех системных сбоев, кроме самых катастрофических. В случае ее неудачного проведения возможен откат транзакции к ее начальной фазе, а после устранения причин неполадок — повторное выполнение этой транзакции.

В многопользовательской системе повреждения базы данных или некорректные результаты возможны даже тогда, когда нет никаких аппаратных или программных сбоев. Серьезные неприятности могут быть вызваны совместными действиями нескольких пользователей по отношению к одной и той же таблице. SQL решает эту проблему, ограничивая возможность внесения изменений только пределами транзакций.

Поместив все операции, воздействующие на базу данных, в транзакции, вы сможете отделить действия одного пользователя от действий другого. Это очень важно, если вы хотите гарантировать получение из базы данных корректных записей.

Если вам непонятно, почему совместные действия двух пользователей могут привести к некорректным результатам, попробуем разобраться в этом на примере. Предположим, что Галина читает запись некоторой таблицы из базы данных, а через мгновение Денис изменяет в этой записи значение числового поля. Затем в то же поле Галина записывает число, полученное на основе значения, прочитанного ею вначале. А так как она не знает об изменении, внесенном Денисом, ее операция оказывается некорректной.

Еще одна неприятность происходит, когда Галина вносит в запись какие-то значения, а Денис затем читает эту запись. О какой корректности данных можно говорить в случае, если Галина проведет откат своей транзакции, а Денис, не зная об откате, выполнит свои действия на основе прочитанного им значения, которое уже не совпадает с тем, что записано в базе данных после отката? Подобная ситуация может рассмешить вас в кинокомедии, но отнюдь не в реальной жизни.

Пользователи и привилегии доступа

Особую угрозу целостности данных могут нести сами пользователи. Одним людям вообще не стоит разрешать доступ к данным. Другим можно предоставить ограниченный доступ к одной части данных и никакого доступа — к остальным. И только очень узкий круг людей может иметь неограниченный доступ ко всем данным. Поэтому вам нужна система, предназначенная для классификации пользователей по категориям и присвоения этим пользователям соответствующих привилегий (полномочий или прав) доступа.

Создатель схемы указывает, кого следует считать ее владельцем. Как владелец схемы вы можете предоставлять пользователям привилегии доступа. Любые привилегии, не предоставленные вами явно, являются недействительными. Вы также можете отозвать уже предоставленные вами привилегии. 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 позволяет защищать следующие объекты базы данных:

- таблицы;
- столбцы;
- представления;
- домены;
- наборы символов;
- схемы сортировки;
- трансляции.

О наборах символов, схемах сортировки и трансляциях мы поговорим в главе 5.

В SQL предусмотрена поддержка различных способов защиты данных — при *просмотре информации*, *добавлении*, *модификации*, *удалении*, *использовании ссылок* и *предоставлении привилегий доступа к базе данных*. Имеются также способы защиты, связанные с выполнением внешних процедур.

СОВЕТ

Для разрешения доступа к данным применяется инструкция `GRANT`, а для его аннулирования — инструкция `REVOKE`. С помощью DCL-инструкций `GRANT` и `REVOKE` можно управлять использованием инструкции `SELECT`, т.е. определять тех, кому разрешено просматривать такие объекты базы данных, как таблица, столбец или представление. Аналогично, управляя инструкцией `INSERT`, вы сможете указывать тех, кому разрешается добавлять в таблицу новые строки. Тот факт, что инструкции `UPDATE` и `DELETE` могут выполняться пользователями, наделенными определенными полномочиями, позволяет назначать пользователей, ответственных за изменение и удаление табличных строк.

Если некоторая таблица базы данных содержит в качестве внешнего ключа столбец, который является первичным ключом в другой таблице из этой базы данных, то можно установить ограничение для первой таблицы, чтобы она ссылалась на вторую (о внешних ключах речь пойдет в главе 5). Если одна

таблица ссылается на другую, то пользователь первой таблицы имеет возможность получать информацию о содержимом второй. Не исключено, что владельцу второй таблицы захочется пресечь такой доступ. В этом может помочь инструкция GRANT REFERENCES. О проблеме, связанной с "предательской" ссылкой, и о ее решении с помощью инструкции GRANT REFERENCES речь пойдет в следующем разделе. Используя инструкцию GRANT USAGE, можно назначать пользователей, которым дается право просматривать содержимое домена, набора символов, схемы сортировки или таблицы трансляции. (Об этом рассказывается в главе 14.)

Инструкции SQL, с помощью которых предоставляют или отзывают привилегии, приведены в табл. 3.4.

Таблица 3.4. Разновидности защиты

Инструкция	Назначение
GRANT SELECT	Позволяет просматривать таблицу
REVOKE SELECT	Запрещает просматривать таблицу
GRANT INSERT	Позволяет вставлять строки в таблицу
REVOKE INSERT	Запрещает вставлять строки в таблицу
GRANT UPDATE	Позволяет изменять значения в строках таблицы
REVOKE UPDATE	Запрещает изменять значения в строках таблицы
GRANT DELETE	Позволяет удалять строки из таблицы
REVOKE DELETE	Запрещает удалять строки из таблицы
GRANT REFERENCES	Позволяет ссылаться на таблицу
REVOKE REFERENCES	Запрещает ссылаться на таблицу
GRANT USAGE ON DOMAIN, GRANT USAGE ON CHARACTER SET, GRANT USAGE ON COLLATION, GRANT USAGE ON TRANSLATION	Позволяет использовать домен, набор символов, схему сортировки или трансляцию
REVOKE USAGE ON DOMAIN, REVOKE USAGE ON CHARACTER SET, REVOKE USAGE ON COLLATION, REVOKE USAGE ON TRANSLATION	Запрещает использовать домен, набор символов, схему сортировки или трансляцию

Разным пользователям, в зависимости от их потребностей, можно предоставить доступ разного уровня. Рассмотрим пример.

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

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

В следующем примере показана инструкция, 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 инструкции, которая позволяет менеджеру по продажам видоизменять розничный прайс-лист. При этом он может изменять содержимое имеющихся строк, но добавлять или удалять строки он не имеет права.

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

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

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

А теперь благодаря инструкции 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

Вероятно, вы думаете, что если можете управлять разрешениями на выполнение функций просмотра, создания, изменения и удаления данных в таблице, то вы надежно защищены. В большинстве случаев это правда. Однако опытный хакер, действуя не напрямую, все равно сумеет взломать вашу базу данных.

Правильно спроектированная реляционная база данных имеет ссылочную целостность, т.е. данные в одной таблице согласуются с данными во всех

остальных таблицах. Чтобы обеспечить ссылочную целостность, проектировщики баз данных применяют к таблицам ограничения, относящиеся к вводимым в таблицу данным. Но в этом и заключается недостаток такой защиты. Если ваша база данных имеет ограничения ссылочной целостности, то какой-нибудь пользователь может создать новую таблицу, в которой 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.

Для примера предположим, что вы являетесь известным аналитиком с Уолл-стрит. Многие верят в точность вашего биржевого анализа, и если вы рекомендуете подписчикам своего блога какие-либо ценные бумаги, то многие их покупают, и стоимость этих бумаг растет. Ваш анализ хранится в базе данных, в которой находится таблица `FOUR_STAR`. В этой таблице содержатся ценные рекомендации, предназначенные для следующей публикации в блоге. Естественно, что доступ к таблице `FOUR_STAR` ограничен, чтобы ни слова не просочилось в "народ", пока блог не прочитают ваши платные подписчики.

Но вы будете оставаться уязвимым, если кому-то удастся создать таблицу, в качестве внешнего ключа которой используется поле таблицы `FOUR_STAR`, содержащее названия ценных бумаг. Вот, например, как выглядит инструкция, 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, какие именно ценные бумаги совпали с имеющимися в вашей конфиденциальной таблице. Благодаря быстродействию компьютеров хакеру не потребуется много времени, чтобы извлечь весь ваш список ценных бумаг.

Вы сможете защитить себя от проделок, подобных показанным в предыдущем примере, если будете остерегаться вводить инструкции такого рода.

```
GRANT REFERENCES (Stock)
      ON FOUR_STAR
      TO SECRET_HACKER;
```

ВНИМАНИЕ!

Я здесь немного преувеличил. Ведь вы никогда не предоставите доступ к важной таблице не заслуживающим доверия лицам, правда? Но это только в том случае, если вы четко понимали, что делали. Однако современные хакеры не просто прекрасно подготовлены технически. Они также хорошие специалисты в области *социальной*

инженерии — искусстве дезориентации людей, в результате чего они выполняют действия, которые поначалу совершенно не собирались выполнять. Вас должно сильно насторожить, если вы услышите от постороннего что-то, имеющее отношение к конфиденциальной информации.


СОВЕТ

Не предоставляйте полномочия тем, кто может ими злоупотребить. Конечно, у людей на лбу не написано, можно ли им доверять. Но если вы не доверите человеку сесть за руль своего автомобиля, то, скорее всего, не стоит предоставлять ему и полномочия REFERENCES на доступ к ценной таблице.

Этот пример демонстрирует первую уважительную причину, по которой следует осторожно обращаться с полномочиями REFERENCES. А вот еще две причины.

- Если кто-то установил для таблицы HOT_STOCKS ограничение с помощью ключевого слова RESTRICT, а вы пытаетесь удалить из своей таблицы строку, то СУБД сообщит, что вам этого делать нельзя, так как будет нарушена ссылочная целостность.
- Вы хотите удалить свою таблицу с помощью инструкции DROP и вдруг обнаруживаете, что вначале кто-то другой должен убрать свое ограничение (или свою таблицу).


ЗАПОМНИ!

Итак, предоставление другим лицам возможности устанавливать для вашей таблицы ограничения, связанные с соблюдением целостности, не только создает потенциальную угрозу безопасности, но и прокладывает путь другим лицам к вашим данным.

Делегирование ответственности за безопасность

Если вы хотите сохранять свою систему в безопасности, то должны строго ограничить и предоставляемые привилегии доступа, и круг людей, которым предоставляете эти привилегии. Однако те, кто не может выполнять свою работу 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. Рассмотрим следующий пример.

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

Этот пример подобен предыдущему (с использованием инструкции GRANT UPDATE) в том смысле, что позволяет менеджеру по продажам обновлять розничный прайс-лист. Но, кроме того, предложение WITH GRANT OPTION дает ему право предоставлять полномочия на обновление любому, кому он захочет. Если вы используете такую форму инструкции GRANT, то должны быть уверены не только в том, что менеджер по продажам разумно использует предоставленные ему полномочия, но и в том, что он будет с осторожностью предоставлять подобные полномочия другим пользователям.

Чрезмерная доверчивость ведет к высокой степени уязвимости. Будьте очень осторожны, используя инструкции, подобные следующей.

```
GRANT ALL PRIVILEGES
      ON FOUR_STAR
      TO Benedict_Arnold WITH GRANT OPTION;
```

Будьте *предельны осторожны* при передаче полномочий. В этом примере пользователь Benedict_Arnold получает все полномочия на использование таблицы FOUR_STAR с возможностью передачи этих полномочий другим лицам. Передача всех привилегий с использованием предложения WITH GRANT OPTION оставляет вас максимально незащищенным. Во время Войны за независимость США одним из доверенных лиц Джорджа Вашингтона был генерал Бенедикт Арнольд. Он перешел на сторону британцев, став, таким образом, самым порицаемым предателем в истории США. Ведь вы же не хотите, чтобы нечто подобное случилось и с вами?

2 Использование SQL для создания баз данных

В ЭТОЙ ЧАСТИ...

- **Создание простой базы данных**
- **Установление связей между таблицами**

Глава 4

Создание простой базы данных

В ЭТОЙ ГЛАВЕ...

- Использование инструментов быстрой разработки для создания, изменения и удаления таблиц базы данных
- Работа с таблицами средствами SQL
- Перенос базы данных в другую СУБД

За всю свою историю компьютерные технологии менялись так стремительно, что в череде их "поколений" порой 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 работы с базами данных, такие как dBASE, Paradox и R:BASE. (Интересно, к какому поколению их можно отнести? Возможно, к третьему с половиной?) Следующим этапом стало появление сред разработки, в которых приложения создавались с минимумом процедурного программирования или совсем без такового. Это, к примеру, такие среды, как Access, PowerBuilder и C++ Builder, называемые языками четвертого поколения. Затем появились инструменты быстрой разработки приложений (Rapid Application Development — RAD) и интегрированные среды разработки (Integrated Development Environments — IDE), такие как Eclipse и Visual Studio .NET, которые поддерживают множество языков (C, C++, C#, Python, Java, Visual Basic и PHP). Они используются для сборки отдельных компонентов в готовые рабочие приложения.

ЗАПОМНИ!

Поскольку SQL не является полноценным языком, он не относится ни к одной из упомянутых выше категорий. Несмотря на то что в SQL применяются инструкции, аналогичные инструкциям языков третьего поколения, в сущности он, подобно языкам четвертого поколения, является непроцедурным. Впрочем, не имеет значения, к какому классу отнести SQL. Его можно использовать в сочетании с инструментами разработки как третьего, так и четвертого поколений. Код SQL можно писать вручную, а можно — с помощью графических инструментов, и тогда соответствующий код будет генерироваться средой разработки. В любом случае СУБД получит только SQL-инструкции.

Из этой главы вы узнаете, как с помощью графических инструментов создать, изменить и удалить простую таблицу и как то же самое проделать с помощью SQL.

Создание простой базы данных с помощью инструмента быстрой разработки

Люди используют базы данных потому, что им нужно сохранять важную информацию. Иногда такая информация является простой, а иногда — нет. Однако в любом случае хорошая СУБД всегда должна предоставлять ту информацию, которая вам нужна. В одних СУБД можно использовать только SQL, в других же, называемых инструментами быстрой разработки (Rapid Application Development — RAD), имеется объектно-ориентированная графическая среда. Также существуют СУБД, поддерживающие оба этих подхода. В следующих разделах для примера мы создадим простую однотабличную базу данных с помощью графических инструментов. Несмотря на то что в данном примере будет использована программа Microsoft Access, в других Windows-ориентированных средах разработки процедура создания базы данных аналогична.

Что хранить в базе данных

Прежде всего нужно решить, какую информацию имеет смысл сохранять в базе данных. Рассмотрим следующий пример. Представьте, что вы только что выиграли в лотерею 10 млн долларов. (Почему бы не помечтать?) И тут откуда ни возьмись начинают появляться приятели и знакомые, о которых вы не слышали годами и о которых уже почти забыли. Одни из них делают вам "беспроигрышные" деловые предложения, которые требуют ваших инвестиций. У других есть достойные инициативы, которые могли бы выиграть от

вашей поддержки. Как хороший распорядитель своего нового капитала вы понимаете, что не все деловые предложения равноценны. Поэтому, чтобы не упустить ни одной из возможностей и сделать справедливый и беспристрастный выбор, вы принимаете решение — добавить все предложения в базу данных.

Вы решили, что по каждому деловому предложению в базу данных имеет смысл занести следующие данные:

- имя;
- фамилия;
- адрес;
- город;
- штат или округ;
- почтовый индекс;
- телефон;
- краткие сведения о том, кто внес предложение;
- само предложение;
- категория предложения: бизнес или благотворительность.

Кроме того, не желая слишком вдаваться в подробности, вы решили заносить все эти данные в единственную таблицу базы данных.

Создание таблицы базы данных

Запустив приложение Access 2016, вы увидите экран приветствия (рис. 4.1). На этом этапе можно создать таблицу базы данных, причем разными способами. Итак, мы приступаем к работе с Microsoft Access в режиме таблицы.

Режим таблицы

По умолчанию Access 2016 открывается в режиме таблицы. Чтобы создать базу данных Access в режиме таблицы, дважды щелкните на шаблоне `Пустая база данных рабочего стола`. В результате приложение будет готово ввести данные в `Таблицу1`, первую таблицу вашей базы данных (рис. 4.2). Вы сможете изменить ее имя позже. Access присвоит вашей базе данных имя `База данных1` (или `База данных31`, если вы создали 30 баз данных и не потрудились дать им осмысленные имена). Все же лучше с самого начала присваивать базам данных информативные имена — так легче избежать путаницы впоследствии.

СОВЕТ

Это метод создания таблицы "с нуля". Существуют и другие методы, например создание таблицы в режиме конструктора.

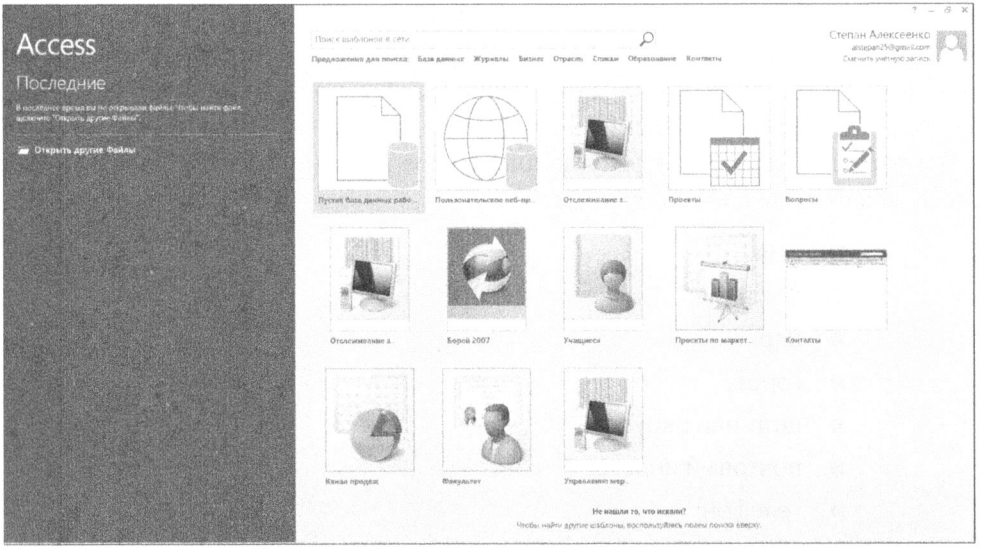


Рис. 4.1. Это окно появляется при создании новой базы данных в Microsoft Access

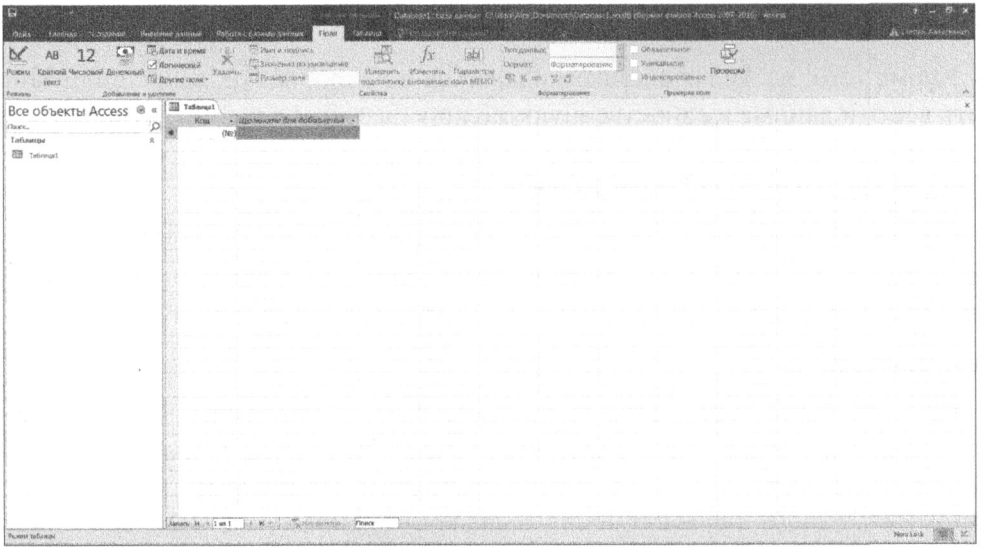


Рис. 4.2. Режим таблицы в среде разработки Access

Режим конструктора

Построить таблицу базы данных в режиме таблицы довольно просто: достаточно начать ввод данных. Но этот подход чреват ошибками, поскольку запросто можно упустить детали. Лучше создавать таблицы в режиме конструктора.

Для этого необходимо выполнить следующие действия.

1. **Открыв Access в режиме таблицы (по умолчанию), щелкните на вкладке Главная ленты, а затем — на кнопке Режим в левом верхнем углу окна. После этого в раскрывающемся меню выберите пункт Конструктор.**

   Откроется диалоговое окно, предлагающее ввести название таблицы.

2. **Введите `POWER` и щелкните на кнопке OK.**

   Приложение перейдет в режим конструктора (рис. 4.3).

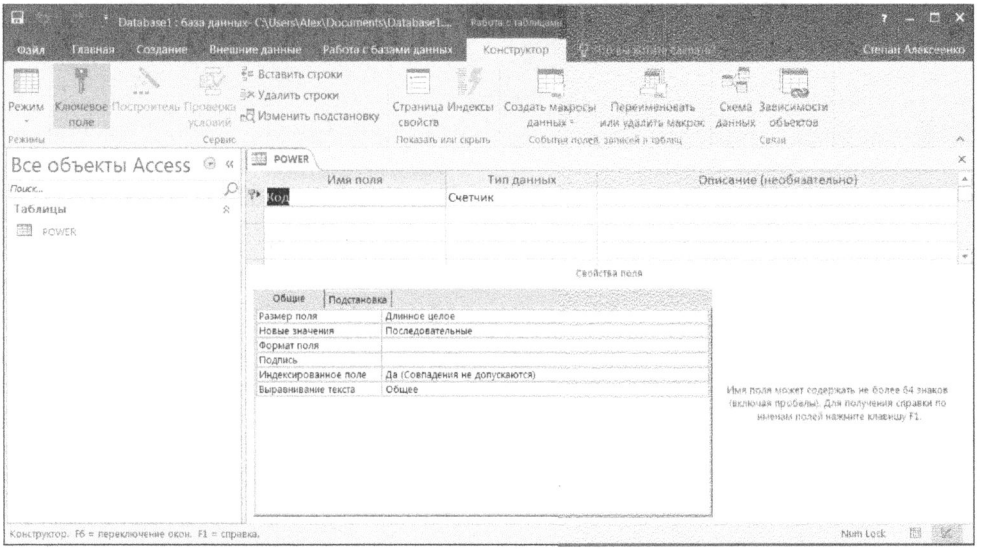


Рис. 4.3. Начальное окно режима конструктора

Обратите внимание на то, что окно делится на функциональные области. Две из них особенно полезны при создании таблицы базы данных.

- *Меню окна конструктора*. В верхней части окна находятся меню Главная, Создание, Внешние данные, Работа с базами данных и Конструктор. Если лента отображена, то ниже меню представлены значки инструментов, доступных в конструкторе.
- *Панель свойств полей*. Эта область предназначена для определения полей базы данных (курсор мигает в первой строке столбца Имя поля). Access предлагает определить здесь первичный ключ, присвоить ему имя (`Код`) и присвоить ему тип счетчика.

СОВЕТ

Счетчик — это тип данных Access, не являющийся стандартным SQL-типом; он автоматически увеличивает на единицу значение поля при каждом добавлении новой записи в таблицу. Этот тип данных гарантирует, что поле, используемое как первичный ключ, не будет продублировано и останется, таким образом, уникальным.

3. **В области свойств полей измените имя поля первичного ключа с `Код` на `ProposalNumber` (номер предложения).**

СОВЕТ

Предлагаемое для первичного ключа имя поля, `Код`, не очень информативно. Если у вас войдет в привычку изменять его на нечто более осмысленное (или предоставлять дополнительную информацию в столбце `Описание`), то будет проще понимать, для чего предназначены поля в базе данных. Теперь имя поля будет достаточно осмысленным.

Текущее состояние проекта таблицы базы данных представлено на рис. 4.4.

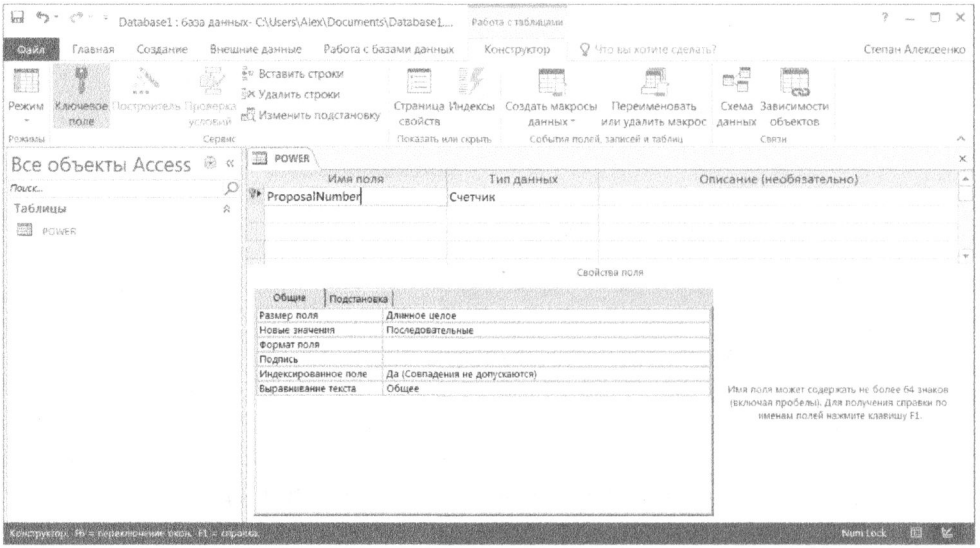


Рис. 4.4. Присвойте полю первичного ключа осмысленное имя

4. **На панели Свойства поля проверьте предположения о поле `ProposalNumber`, сделанные автоматически в среде Access.**

На рис. 4.4 демонстрируются следующие предположения:

- размер поля выбран как Длинное целое;
- для свойства Новые значения выбран вариант Последовательные;
- индексированное поле не допускает совпадений;
- выравнивание текста является общим.

Как это нередко бывает, Access делает правильные предположения о том, что вы собираетесь сделать. Но если какое-нибудь предположение окажется неправильным, то его можно будет легко переопределить, введя новое значение.

5. **Определите остальные поля, которые должна содержать эта таблица.**

На рис. 4.5 показано окно конструктора после ввода поля `FirstName`.

Рис. 4.5. Окно создания таблицы после определения поля `FirstName`

СОВЕТ

Тип данных поля `FirstName` — Короткий текст, а не Счетчик, поэтому и свойства полей будут разными. Программа назначила полю `FirstName` размер 255 символов, принимаемый по умолчанию для коротких текстовых данных. Я лично не знаю людей с такими длинными фамилиями. Access — достаточно интеллектуальная программа, чтобы выделять столько места, сколько реально нужно для конкретного значения. Она не оставляет "вслепую" 255 байтов для хранения любого значения. Однако другие среды разработки могут не обладать подобным свойством. Я предпочитаю присваивать длинам полей разумные значения. И это оберегает меня от возможных неприятностей в случае перехода от одной среды разработки к другой.

Access по умолчанию предполагает, что поле `FirstName` не является обязательным. Вы можете ввести запись в таблицу `POWER` и оставить поле `FirstName` пустым, как будто знаете человека только по имени, например Шер или Боно.

6. **Установите параметр Размер поля `FirstName` равным 15.**

   Причины этого описаны во врезке.

ПРОДУМЫВАЙТЕ СТРУКТУРУ ТАБЛИЦ ЗАРАНЕЕ

В некоторых средах разработки (отличных от Microsoft Access) сокращение размера поля `FirstName` до 15 символов сэкономит на каждой записи в базе данных по 240 байтов, если вы работаете с символами ASCII (UTF-8), по 480 байтов, если используются символы UTF-16, или по 960 байтов, если используются

символы UTF-32. Это немало. Кроме того, обратите внимание на стандартные предположения для других свойств полей и попробуйте предугадать, как вы могли бы использовать их при увеличении базы данных. Одни поля требуют внимания всегда (например, поле `FirstName`), а другие — только в отдельных случаях. Возможно, вы обратили внимание на еще одно популярное свойство поля: Индексированное поле. Если вы не предполагаете возвращать записи по данному полю, то не тратьте впустую ресурсы процессора на его индексирование. Но в таблице с большим количеством строк можно существенно ускорить поиск, проиндексировав поле, которое собираетесь использовать для идентификации искомых записей. Другими словами, обращайте внимание на любые детали при проектировании таблиц.

7. **Чтобы гарантировать возможность быстрого получения записей из таблицы `POWER` по полю `LastName` (что весьма вероятно), измените значение его свойства Индексированное поле на `Да` (Допускаются совпадения), как показано на рис. 4.6.**

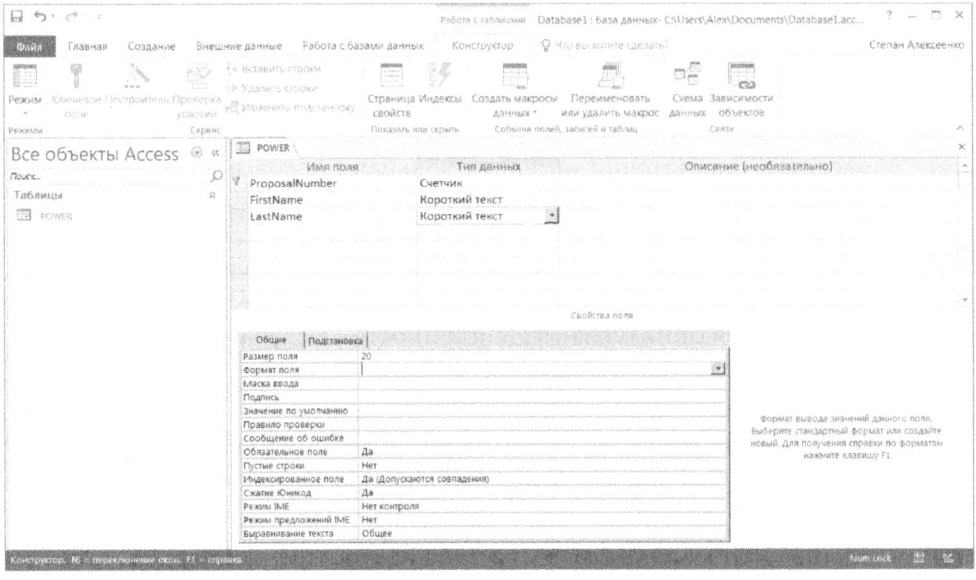


Рис. 4.6. Окно создания таблицы после определения поля `LastName`

На рисунке видно несколько изменений, внесенных мною на панели свойств поля.

- Максимальный размер поля сокращен с 255 до 20 символов.
- Значение свойства Обязательное поле изменено на Да, Пустые строки — на Нет и Индексированное поле — на Да (Допускаются совпадения). Поскольку для каждого предложения я хочу включить фамилию человека, ответствен-

ного за него, поле `LastName` будет индексировано и не допустит нулевой длины.

- Я допускаю совпадения, поскольку предложения могут внести люди с одинаковой фамилией. В случае таблицы `POWER` это практически неизбежно, ведь я ожидаю предложений от всех трех моих братьев, а также от моих сыновей и незамужней дочери, не говоря уже о дальних родственниках.

ЗАПОМНИ!

- Значение `Да (Совпадения не допускаются)`, от которого я отказался, больше подходит для поля первичного ключа таблицы. Первичный ключ таблиц никогда не должен содержать совпадений.

8. **Введите остальные поля, изменяя их стандартный размер на более подходящий.**

   Результат представлен на рис. 4.7.

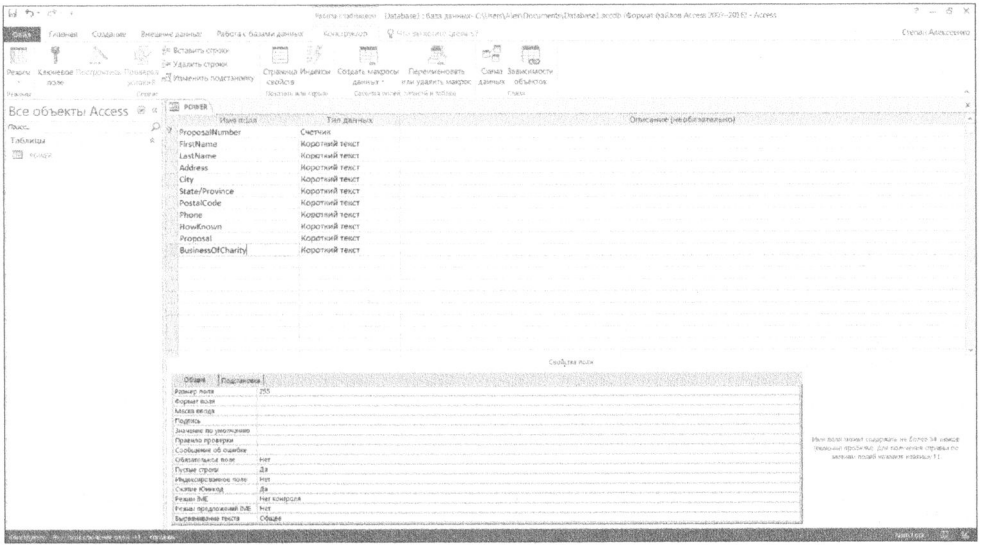


Рис. 4.7. Окно создания таблицы после определения всех полей

СОВЕТ

Как можно заметить на рис. 4.7, поле `BusinessOrCharity` не индексируется. Нет никакого смысла индексировать поле, у которого есть только два возможных значения. Индексация не сузит выбор настолько, чтобы это стоило делать.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

В Access вместо термина "столбец" или "атрибут" используется термин "поле". Первоначально системы обработки файлов не были реляционными и в них использовались термины "файл", "поле" и "запись", которые характерны для систем плоских файлов.

9. **Сохраните таблицу, щелкнув на пиктограмме с изображением дискеты в левом верхнем углу.**

СОВЕТ

Разрабатывая базу данных, будьте осторожны. Например, чаще сохраняйте проект по мере разработки. Для этого достаточно время от времени щелкать на пиктограмме дискеты. Это может избавить вас от утомительных повторений в случае скачков напряжения в сети или других неприятностей. Кроме того, если вы назовете базу данных и одну из ее таблиц одним и тем же именем, то это может запутать пользователей и администраторов. Как правило, логичнее (и удобнее) придумывать для всех объектов разные имена.

Сохранив таблицу, вы, возможно, решите, что первоначальный проект нуждается в улучшении. Этот вопрос мы обсудим в следующем разделе.

Изменение структуры таблицы

Как правило, созданные таблицы не получатся с первого раза такими, как надо. Если вы работаете на кого-то, будьте уверены: ваш клиент ждет не дождется, пока вы наконец-то создадите базу данных, чтобы сообщить вам о необходимости дополнить ее еще одним элементом данных, а возможно, даже несколькими. Это значит, что "наша песня хороша, начинай сначала".

Если вы создаете базу данных для себя, то учтите, что недостатки в ее структуре, которые поначалу совсем не были видны, всплывут на поверхность уже после создания структуры, и это неизбежно (по закону Мерфи). Возможно, к вам начнут поступать предложения из другой страны, и тогда потребуется добавить столбец `Country` (страна). Возможно, вы решите, что не лишним будет еще и задать адрес электронной почты. В любом случае придется переделать то, что вы создали вначале. Такая возможность — переделывать уже созданное — существует не только в Access, но и во всех инструментах быстрой разработки.

СОВЕТ

Если возникнет необходимость модифицировать таблицы базы данных, уделите внимание всем полям. Например, можно добавить второе поле `Address` для людей со сложными адресами и поле `Country` — для предложений из других стран.

СОВЕТ

Несмотря на то что изменять таблицы базы данных вовсе не сложно, старайтесь по возможности этого избегать. Все приложения, которые зависят от старой структуры базы данных, по всей вероятности, перестанут работать, и их придется переделывать. Если у вас много приложений, то эта задача может оказаться очень сложной. Постарайтесь предусмотреть все заранее и избегать последующих модернизаций. Небольшой дополнительный резерв, заложенный в базу данных изначально, обычно предпочтительнее переделки всех

приложений, написанных несколько лет назад. И если вы уже забыли нюансы их работы, то может оказаться, что такие приложения, по сути, "не подлежат восстановлению".

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

1. **В окне создания таблицы щелкните правой кнопкой мыши на маленьком цветном квадрате слева от поля `City`, чтобы выделить эту строку, и выберите из контекстного меню команду Вставить строки.**

   Выше позиции курсора появится пустая строка, которая сдвинет вниз все существовавшие ранее строки (рис. 4.8).

2. **Введите в таблицу необходимые поля.**

   Я добавил поле `Address2` выше поля `City` и поле `Country` выше поля `Phone`.

3. **Сохраните таблицу, прежде чем ее закрыть.**

   Результат должен выглядеть так, как на рис. 4.9.

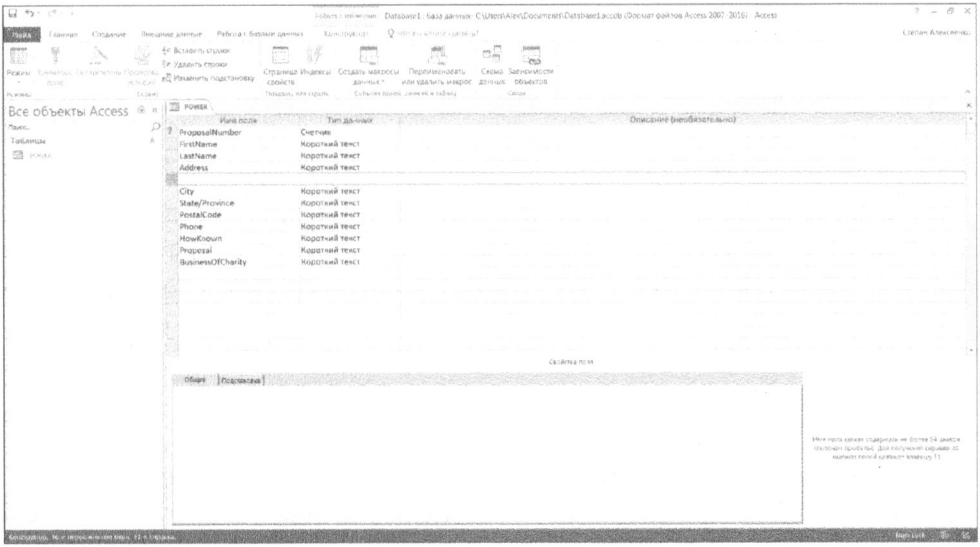


Рис. 4.8. Появилось свободное место для второй строки адреса

Создание индекса

В любой базе данных необходим способ быстрого доступа к интересующим вас данным. (Это особенно верно, когда количество инвестиционных и благотворительных предложений может легко перевалить за тысячу.) Предположим, вы хотите просмотреть все предложения, сделанные вашими братьями. Можно

выделить эти предложения с помощью запроса, опираясь на содержимое поля LastName. Вот как выглядит этот SQL-запрос.

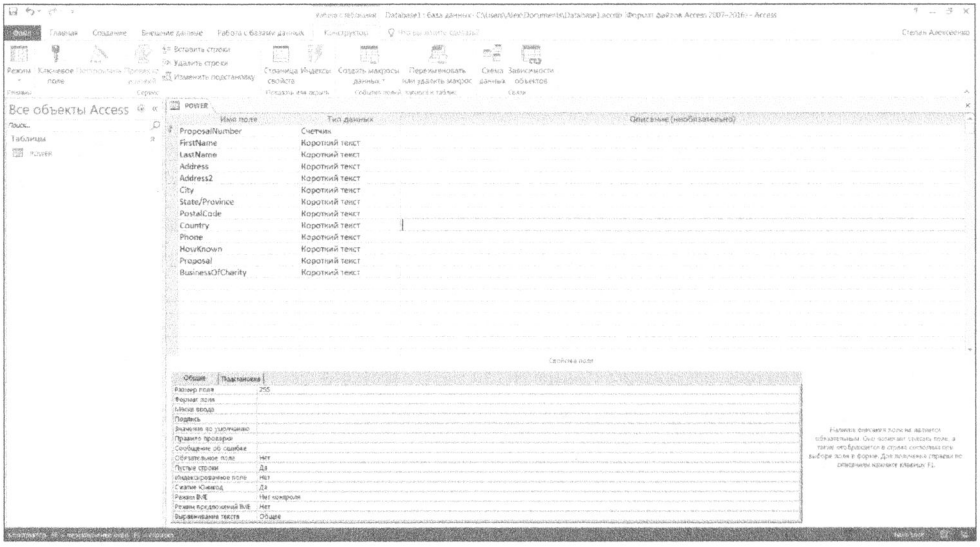


Рис. 4.9. Измененное определение таблицы должно выглядеть так

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

Эта стратегия может не сработать для предложений, сделанных вашими двоюродными братьями и шуринами, поэтому необходимо "заглянуть" еще в одно поле (а именно — в поле HowKnown), как показано в следующем примере.

```
SELECT * FROM POWER
 WHERE HowKnown = 'brother-in-law'
    OR
       HowKnown = 'half brother' ;
```

SQL просматривает таблицу строка за строкой, отыскивая записи, удовлетворяющие условию предложения WHERE. Если таблица POWER довольно велика (десятки тысяч записей), то результата придется ждать долго. Но можно ускорить поиск, применив к таблице POWER индексы. (*Индекс* — это таблица указателей, каждая строка которой ссылается на соответствующую строку в таблице данных.)

Можно определить индекс для всех полей, с помощью которых вы собираетесь получать доступ к своим данным. Если вы добавляете, изменяете или удаляете строки в таблице данных, то вам не нужно пересортировывать саму таблицу, достаточно лишь модифицировать индексы. Это намного быстрее, чем сортировать таблицу. После установки индекса с необходимым порядком

сортировки можно использовать его для почти мгновенного доступа к строкам в таблице данных.


СОВЕТ

Поскольку поле `ProposalNumber` (номер предложения) одновременно и уникальное, и короткое, с его помощью можно быстрее всего получить доступ к конкретным записям. Именно оно является лучшим кандидатом на роль первичного ключа. Поскольку первичные ключи, как правило, обеспечивают самый быстрый доступ к данным, они всегда должны быть проиндексированы. В приложении Access это делается автоматически. Но, чтобы использовать поле `ProposalNumber`, необходимо знать его значение в нужной вам записи. Вам могут понадобиться и дополнительные индексы, создаваемые на основе других полей, таких как `LastName` (фамилия), `PostalCode` (почтовый индекс) или `HowKnown` (кто таков). Если в таблице, проиндексированной по полю `LastName`, в результате поиска будет найдена первая строка, в которой значением этого поля является `Marx`, то будут найдены и все остальные строки с таким же значением этого поля. В индексе ключи для всех строк `Marx` следуют непосредственно один за другим. Поэтому все строки, относящиеся к именам `Chico`, `Groucho`, `Harpo`, `Zeppo` и `Karl`, можно получить почти так же быстро, как и для только одного имени `Chico`.

В результате создания индексов система испытывает дополнительную нагрузку, от чего ее работа немного замедляется. Это замедление необходимо компенсировать за счет увеличения скорости доступа к записям в результате использования индекса.


СОВЕТ

Ниже приведен ряд советов, которые помогут правильно выбрать поля для индексирования.

- Если индексировать поля, часто используемые для доступа к записям из большой таблицы, то получать результаты запросов можно будет практически мгновенно.
- Если создавать индексы для полей, которые практически никогда не будут использоваться для доступа к записям, то потери времени и ресурсов памяти окажутся бессмысленными.
- Нет смысла создавать индексы для полей, которые не позволяют отличить одну запись от множества других. Например, поле `BusinessOrCharity` (бизнес или благотворительность) разбивает все записи в таблице только на две категории, поэтому хороший индекс на основе этого поля создать нельзя.


ЗАПОМНИ!

Эффективность индекса в разных реализациях бывает разной. Если перенести базу данных с одной платформы на другую, то индексы, обеспечившие максимальную производительность в первой системе, могут плохо работать во второй. Случается даже так, что после переноса база данных хуже работает с индексами, чем если бы их не было вовсе. Попробуйте использовать разные схемы индексации и опытным путем выяснить, какая из них больше повышает производительность. Следует оптимизировать индексы так, чтобы изменение платформы не оказывало негативного влияния ни на скорость извлечения данных, ни на их обновление.

Чтобы создать индексы для таблицы `POWER`, достаточно выбрать значение Да для параметра Индексированное поле на панели Свойства поля окна создания таблицы.


СОВЕТ

Access автоматически создает индекс для поля `PostalCode`, поскольку оно часто используется для поиска данных, *а также* автоматически индексирует первичный ключ.

В отличие от поля `ProposalNumber`, поле `PostalCode` не является первичным ключом, и его значения не обязательно уникальны. Вы уже создали индекс для поля `LastName`, и сделайте то же для поля `HowKnown`, так как оба этих поля, вероятно, будут часто использоваться для доступа к данным.

Создав все нужные индексы, не забудьте сохранить новую табличную структуру, прежде чем ее закрыть.


СОВЕТ

Конечно, если вы используете не Microsoft Access, то конкретные действия, которые описывались в этом разделе, вряд ли сможете применить, однако общий процесс будет аналогичным.

Удаление таблицы

Прежде чем таблица (в данном случае — `POWER`) приобретет нужную вам структуру, вы, возможно, успеете создать несколько промежуточных вариантов. Наличие в системе этих вариантов может впоследствии запутать пользователей, поэтому, пока вы еще помните, что к чему, лучше всего удалить лишние таблицы. Для этого щелкните правой кнопкой мыши на предназначенной для удаления таблице в списке `Все таблицы`, расположенном в левой части окна. В появившемся контекстом меню выберите команду Удалить (рис. 4.10), и таблица будет удалена из базы данных.

При удалении лишний раз убедитесь в правильности своих действий. После выбора команды Удалить таблица и все данные в ней будут уничтожены.

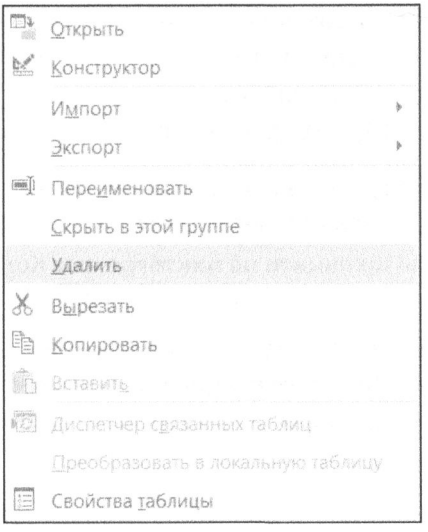


Рис. 4.10. Для удаления таблицы достаточно выбрать из меню команду Удалить

При удалении таблицы программа Access также удаляет все подчиненные ей таблицы и все связанные с ними индексы.

Создание таблицы POWER средствами SQL

Все действия по созданию базы данных, которые можно выполнять с помощью графических инструментов СУБД, можно выполнять и с помощью SQL. В этом случае вместо того, чтобы щелкать мышью на элементах интерфейса, нужно вводить команды с клавиатуры. Те, кто предпочитают манипулировать графическими объектами, найдут инструменты СУБД более понятными и простыми для изучения. Другие же (например, те, кому больше нравится объединять слова в логические предложения) предпочтут работу с SQL-инструкциями.

Поскольку одни вещи легче представить, используя объектно-ориентированную парадигму, а другие — используя SQL-инструкции, полезно знать оба метода.

В следующих разделах для создания той же самой таблицы, а также для выполнения операций по ее изменению и удалению, будет применяться SQL.

Создание SQL-запросов в Microsoft Access

Программа Access была создана как инструмент быстрой разработки, не требующий программирования. В то же время Access позволяет писать и выполнять SQL-инструкции, хотя для этого вам придется "зайти с черного хода". Чтобы открыть редактор SQL-кода, выполните следующие действия.

1. **Откройте свою базу данных и щелкните на вкладке Создание, чтобы отобразить ленту в верхней части окна.**
2. **В разделе Запросы щелкните на пиктограмме Конструктор запросов.**

   Откроется диалоговое окно Добавление таблицы.
3. **Выберите таблицу `POWER`. Щелкните на кнопке Добавить, а затем — на кнопке Закрыть, чтобы закрыть диалоговое окно.**

   В результате окно приложения будет выглядеть как на рис. 4.11.

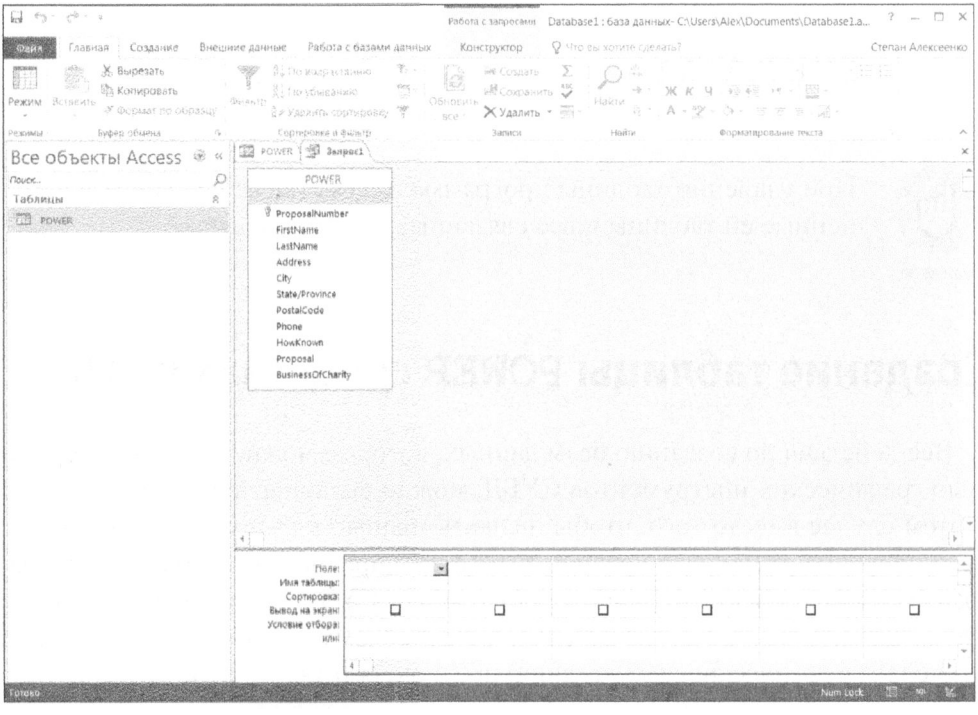


Рис. 4.11. Окно запроса с выбранной таблицей `POWER`

В верхней части рабочей области будет отображена таблица `POWER` с атрибутами, а чуть ниже — раздел Запрос по образцу. Теперь ожидается, что вы введете

запрос, перетаскивая поля таблицы и задавая различные параметры. (Вы, конечно, могли бы сделать это, но тогда вы ничего не узнали бы об использовании SQL в среде Access.)

4. **Щелкните на вкладке Главная, а затем в левом углу ленты — на значке Режим.**

   Появится меню режимов, доступных при работе с запросами (рис. 4.12).

   Среди них нетрудно найти Режим SQL.

Рис. 4.12. Режимы, доступные при работе с запросами

5. **Чтобы отобразить вкладку представления объектов SQL, выберите Режим SQL.**

   Как показано на вкладке представления объектов SQL (рис. 4.13), Access предполагает (небезосновательно), что вы хотите извлечь информацию из таблицы `POWER`, поэтому первая часть запроса написана без вашего участия. Приложению не известно в точности, какие именно данные вы хотите получить, поэтому отображается только та часть запроса, которая будет использована наверняка.

   Вот что здесь пока написано.

```
SELECT
  FROM POWER ;
```

6. **Заполните пустое место в первой строке звездочкой (*) и добавьте предложение `WHERE` после строки `FROM`.**

   Если вы уже ввели какие-нибудь данные в таблицу `POWER`, то можете задать параметры поиска примерно так.

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

   Убедитесь, что ваша SQL-инструкция завершается точкой с запятой (`;`). Не забудьте перенести ее от слова `POWER` в конец следующей строки.

7. **Щелкните на значке Сохранить (с изображением дискеты).**

   Access запросит у вас имя только что созданного запроса.

8. **Введите имя и щелкните на кнопке OK.**

Ваша инструкция сохранена и может быть выполнена впоследствии как запрос.

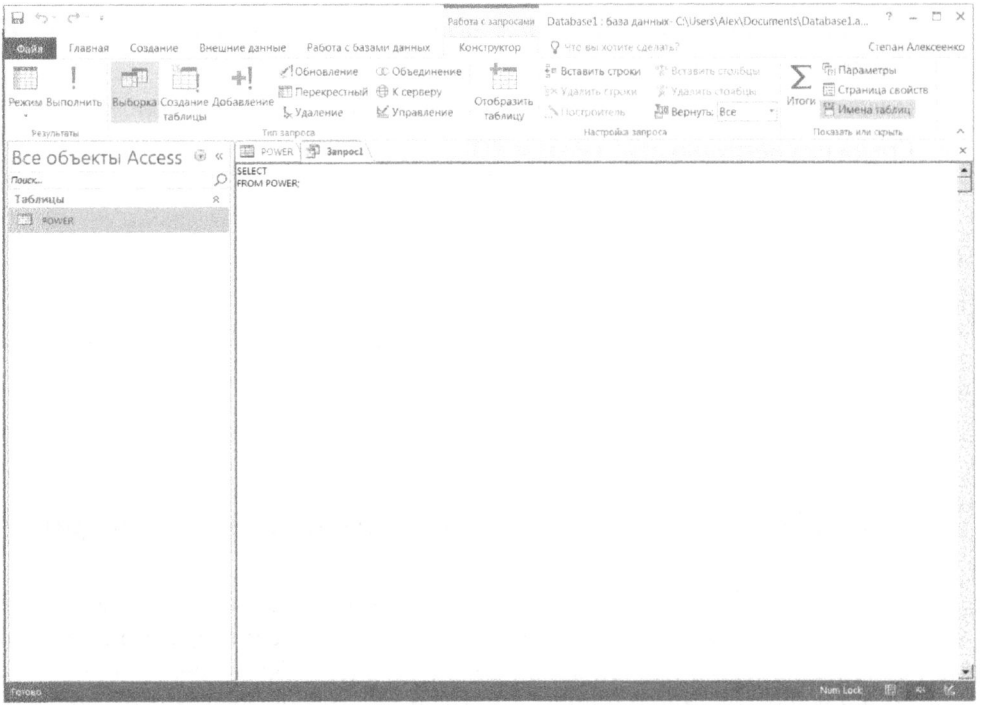


Рис. 4.13. Вкладка представления объектов в режиме SQL

Создание таблицы

Чтобы создать таблицу базы данных с помощью SQL в такой полнофункциональной СУБД, как Microsoft SQL Server, Oracle или IBM DB2, необходимо ввести ту же информацию, что и при создании таблицы с помощью инструмента быстрой разработки. Отличие состоит в том, что инструмент быстрой разработки предоставляет в ваше распоряжение диалоговое окно создания таблицы (или какую-либо форму) и не позволяет вводить неправильные имена полей, типы или размеры.

От SQL не стоит ожидать такой любезности. Вам следует с самого начала знать, что нужно делать. Вы должны ввести инструкцию `CREATE TABLE` целиком, не надеясь на то, что SQL сообщит о наличии в ней ошибок.

В соответствии со стандартом ISO/IEC инструкция SQL, которая создает таблицу деловых предложений (идентичную созданной ранее), использует следующий синтаксис.

```
CREATE TABLE POWERSQL (
   ProposalNumber      INTEGER    PRIMARY KEY,
   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),
   BusinessOrCharity   CHAR (1)
) ;
```

Как видите, информация в сущности та же, что и при создании таблицы с помощью графического интерфейса Access. Впрочем, что хорошо в языке SQL, так это его универсальность. С помощью одного и того же стандартного синтаксиса можно работать с любой СУБД.

В Access 2016 создание таких объектов базы данных, как таблицы, немного усложнено. Вы не сможете просто ввести во вкладке представления объектов инструкцию CREATE (подобную представленной выше). Дело в том, что вкладка представления объектов SQL доступна только как инструмент запросов, и вам придется сообщить программе (посредством дополнительных операций) о предстоящем вводе запроса описания данных, а не обычного запроса на получение информации из базы данных. И еще: поскольку создание таблицы — это действие, способное поставить под угрозу безопасность базы данных, по умолчанию оно запрещено. Прежде чем Access примет запрос описания данных, вы должны убедить программу, что это безопасная база данных.

1. **Перейдите на вкладку Создание ленты, чтобы отобразить пиктограммы создания различных объектов.**
2. **В разделе Запросы щелкните на пиктограмме Конструктор запросов.**

   Откроется диалоговое окно Добавление таблицы, содержащее таблицу POWER и несколько системных таблиц.
3. **Выберите таблицу POWER и щелкните на кнопке Добавить.**

   Как вы уже видели в предыдущем примере, таблица POWER и ее атрибуты отображаются в верхней части рабочей области.
4. **Щелкните на кнопке Закрыть диалогового окна Добавление таблицы.**
5. **Щелкните на вкладке Главная, а затем в левом углу ленты — на пиктограмме Режим, после чего выберите в раскрывающемся меню Режим SQL.**

   Как и в предыдущем примере, программа Access помогла вам ввести в SQL-редакторе код SELECT FROM POWER. Но на сей раз помощь вам не нужна.

6. **Удалите код `SELECT FROM POWER`, а на его месте введите следующий запрос описания данных.**

```
CREATE TABLE POWERSQL (
   ProposalNumber      INTEGER    PRIMARY KEY,
   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),
   BusinessOrCharity   CHAR (1)
) ;
```

На данный момент окно должно выглядеть так, как на рис. 4.14.

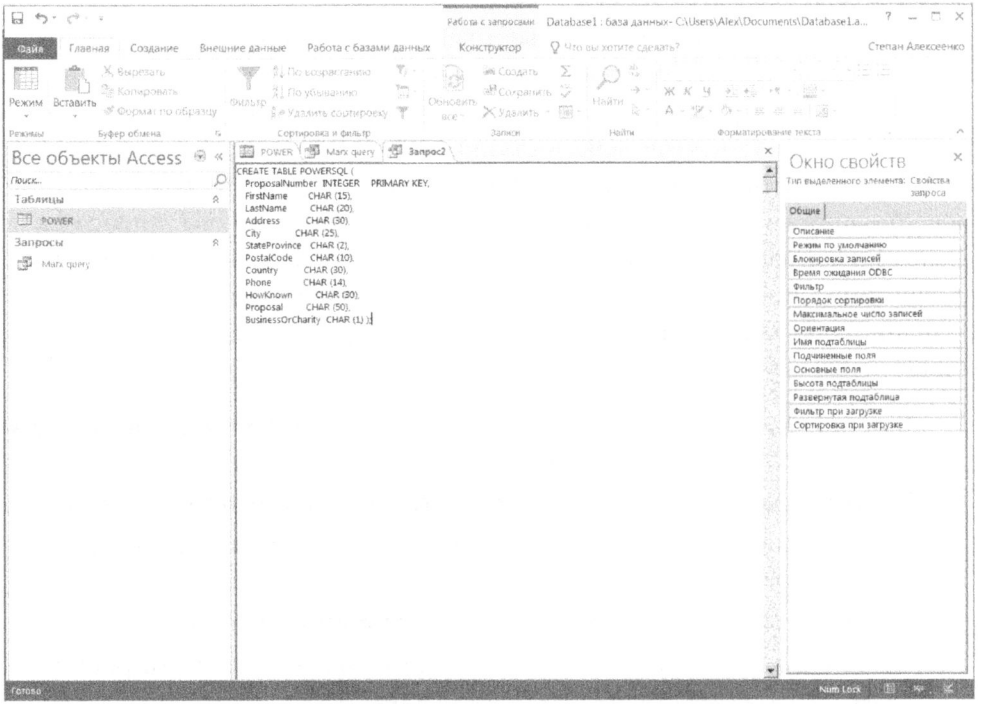


Рис. 4.14. Запрос на создание таблицы

7. **После выбора вкладки Конструктор на ленте щелкните на пиктограмме Выполнить (с красным восклицательным знаком).**

Тем самым будет выполнен запрос, который создаст таблицу POWERSQL, как показано на рис. 4.15.

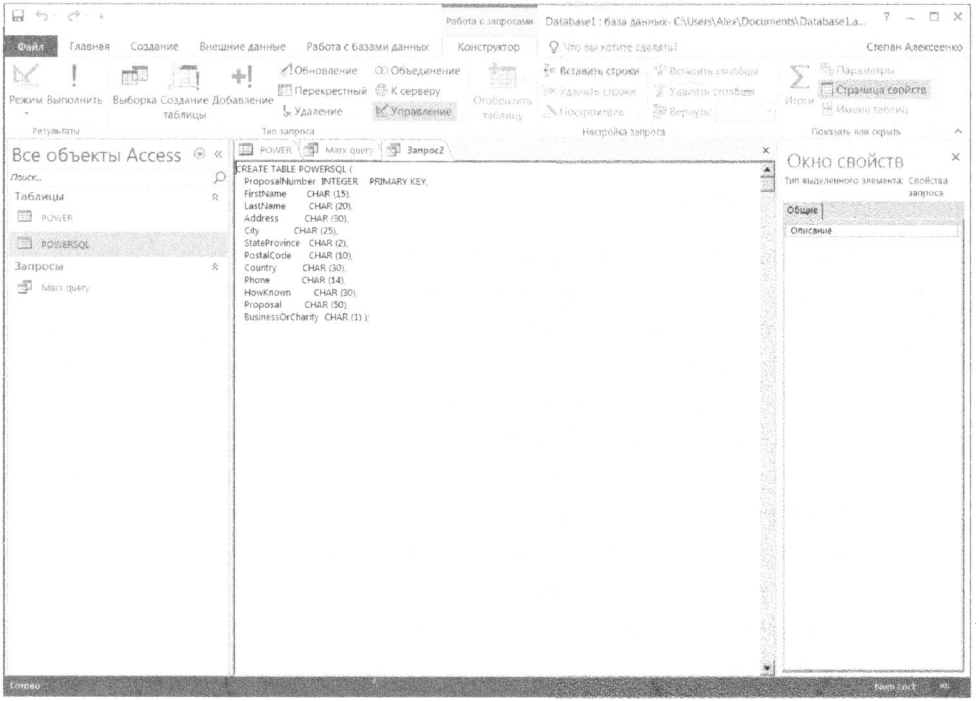


Рис. 4.15. Таблица POWERSQL создана

Теперь вы должны увидеть таблицу POWERSQL в списке Все объекты Access в левой части окна приложения. Если это так, все в порядке. Но если вы не увидите ее, то читайте далее.

ВНИМАНИЕ!

Access 2016 идет на все, чтобы защитить вас от злонамеренных хакеров и собственных небрежных ошибок. Поскольку выполнение запроса описания данных потенциально опасно для базы данных, Access по умолчанию препятствует этому действию. Если так произошло и в вашем случае, то таблица POWERSQL отсутствует в упомянутом списке, поскольку запрос не был выполнен. Вместо этого в строке сообщений (под лентой) может появиться следующее сообщение:

```
Предупреждение системы безопасности: Часть содержимого базы
данных отключена
```

Если вы получили такое сообщение, переходите к следующим действиям.

8. **Щелкните на вкладке Файл и выберите из меню пункт Параметры.**

   Откроется диалоговое окно Параметры Access.

9. **В диалоговом окне Параметры Access выберите раздел Центр управления безопасностью.**
10. **Щелкните на кнопке Параметры центра управления безопасностью.**
11. **Выберите из меню слева пункт Панель сообщений, а затем установите переключатель Показывать панель сообщений, если он еще не выбран.**
12. **Вернитесь к месту выполнения запроса описания данных, создающего таблицу `POWERSQL`.**
13. **Выполните запрос.**


ЗАПОМНИ!

Усилия, затраченные на изучение SQL, еще долго будут приносить вам дивиденды, потому что сходить со сцены этот язык пока не собирается. В то же время усилия, затраченные на освоение конкретного инструмента разработки, скорее всего, окажутся менее эффективными. Ведь каким бы прекрасным ни было даже самое современное приложение, будьте уверены — в течение трех-пяти лет его заменит более совершенная технология. Хорошо еще, если за это время вы успеете окупить вложенные затраты. Конечно, можно продолжать пользоваться уже освоенной программой, но все же будет разумнее глубоко изучить SQL. Этот язык (как старый и испытанный друг) никогда не подведет и намного дольше будет приносить вам пользу.

Создание индекса

Индексы — очень важная часть любой реляционной базы данных. Они служат указателями в таблицах, содержащих нужные данные. С помощью индекса можно перейти к определенной записи, не выполняя последовательный перебор всех строк таблицы. В больших таблицах без индексов просто не обойтись, поскольку они позволяют получить результат в течение секунд (а не ожидать долгие годы). Конечно, я несколько преувеличил, однако некоторые операции по выборке данных могут действительно выполняться очень долго. И если вы не в состоянии столько ждать, то операцию придется прервать.


ВНИМАНИЕ!

Но даже если в двух СУБД для инструкции `CREATE INDEX` используются одни и те же слова, способы ее реализации могут быть разными. Чтобы узнать, как создавать индексы, необходимо внимательно изучить документацию по используемой вами СУБД.

Удивительно, но в спецификации SQL нет средств создания индексов. В то же время различные СУБД оснащены собственными реализациями этого компонента. А так как подобные реализации не стандартизированы, то вполне

могут отличаться друг от друга. Большинство поставщиков реализуют средства создания индексов, расширяя SQL инструкцией CREATE INDEX.

Изменение структуры таблицы

Для изменения структуры существующей таблицы можно использовать SQL-инструкцию ALTER TABLE. Интерактивные средства SQL не такие удобные, как аналогичные средства инструмента быстрой разработки, который отображает на экране табличную структуру, предоставляя пользователю возможность ее изменить. Если вы используете SQL, то должны заранее знать и структуру таблицы, и то, как ее следует изменить. Для этого нужно ввести соответствующую инструкцию (в том месте экрана, где находится приглашение на ввод). Впрочем, если вы хотите встроить инструкции изменения таблицы в какое-нибудь приложение, то использование SQL — самый легкий способ.

Чтобы добавить в таблицу POWERSQL второе поле для адреса, используйте следующую DDL-инструкцию.

```
ALTER TABLE POWERSQL
  ADD COLUMN Address2 CHAR (30);
```

Чтобы расшифровать этот код, не нужно быть профессионалом. В действительности это может сделать даже новичок со слабыми познаниями в английском. Приведенная выше инструкция изменяет таблицу с названием POWERSQL, добавляя в нее новый столбец, который называется Address2, имеет тип данных CHAR и размер 30 символов. Приведенный пример показывает, насколько легко изменять структуру таблиц в базе данных с помощью DDL-инструкций.

Стандарт SQL разрешает использовать эту инструкцию не только для добавления в таблицу нового столбца, но и для удаления уже существующего столбца, как показано в следующем примере.

```
ALTER TABLE POWERSQL
 DROP COLUMN Address2;
```

Удаление таблицы

Таблицу, которая вам больше не нужна, можно легко удалить — достаточно воспользоваться инструкцией DROP TABLE, например:

```
DROP TABLE POWERSQL;
```

Что может быть проще? При удалении таблицы с помощью этой инструкции будут удалены все ее данные и метаданные. От таблицы не останется и следа. Как правило, это прекрасно срабатывает. Единственный случай, когда это не сработает, — если в базе данных существует другая таблица, которая ссылается на ту, которую вы пытаетесь удалить. Это называется *ограничением*

ссылочной целостности. В таком случае SQL выдаст сообщение об ошибке и не удалит таблицу.

Удаление индекса

ВНИМАНИЕ!

Удаляя таблицу с помощью инструкции `DROP TABLE`, вы одновременно удаляете и все связанные с ней индексы. Иногда возникают ситуации, когда нужно оставить таблицу, но удалить один из ее индексов. Стандарт SQL не определяет инструкцию удаления индекса `DROP INDEX`, но в большинстве реализаций СУБД она все-таки предусмотрена. Эта команда пригодится тогда, когда ваша система замедлит свою работу до черепашьей скорости и обнаружится, что таблицы в ней индексированы не самым лучшим образом. Исправление индексов должно привести к резкому увеличению производительности системы. Это, правда, может опечалить пользователей, привыкших в ожидании результатов своих запросов устраивать перекур.

Переносимость

В любой реализации SQL могут существовать расширения, не предусмотренные стандартом SQL. Одни из них могут войти в следующий выпуск стандарта SQL. Другие же останутся уникальными для конкретной реализации и, вероятно, никогда не войдут в стандарт.

Часто эти (другие) расширения могут упростить создание приложений, и у вас может возникнуть искушение воспользоваться ими. Это тоже вариант, но учтите: не исключено, что в таком случае придется чем-то пожертвовать. Если когда-нибудь потребуется перенести ваше приложение в другую СУБД, вам, возможно, придется переписывать те части приложения, в которых используются расширения, не поддерживаемые в новой среде.

СОВЕТ

Чем лучше вы знаете свою реализацию и существующие тенденции разработки приложений, тем лучшее решение примете. Подумайте о вероятности переноса приложения на другую платформу в будущем, а также поинтересуйтесь, является ли используемое вами расширение уникальным для конкретной реализации или все-таки достаточно распространенным. Использование расширения может в будущем потребовать гораздо больше времени на переделку приложения, чем сэкономит в данный момент. С другой стороны, вы попросту можете не найти причин, по которым вам стоит отказаться от использования расширения. Так что решать вам.

Глава 5

Создание многотабличной базы данных

В ЭТОЙ ГЛАВЕ...

- Что включать в базу данных
- Определение отношений между таблицами
- Связывание таблиц с помощью ключей
- Вопросы целостности данных
- Нормализация базы данных

В этой главе будет рассмотрен пример создания многотабличной базы данных. Приступая к проектированию любой базы данных, необходимо решить, что в нее включать, а что — нет. Затем нужно определиться, как именно включаемые в базу элементы будут связаны между собой, и создать таблицы с учетом этой информации. Мы обсудим, как использовать ключи для получения быстрого доступа к отдельным табличным записям и индексам.

База данных должна не только хранить данные, но и защищать их от повреждений. Поэтому мы поговорим о том, как можно защитить целостность данных, и рассмотрим один из основных методов защиты — *нормализацию*. Вы узнаете о различных нормальных формах и проблемах, которые позволяет устранить нормализация.

Проектирование базы данных

Проектирование базы данных начинайте с выполнения следующих действий.

1. **Решите, какие объекты должны храниться в вашей базе данных.**
2. **Установите, какие из объектов должны быть таблицами, а какие — столбцами в этих таблицах.**
3. **Определите таблицы в соответствии с проектом базы данных.**

   Возможно, вы захотите назначить ключом некоторый столбец таблицы или их комбинацию. Ключи позволят вам быстро найти в таблице нужную строку.

В последующих разделах подробно описываются эти действия, а также некоторые проблемы, возникающие в процессе проектирования базы данных.

Шаг 1: определение объектов

Прежде всего, необходимо решить, какие аспекты системы являются достаточно важными для включения в модель. Каждый такой аспект рассмотрите как объект и составьте список этих объектов — всех, какие только придут вам в голову. И не пытайтесь пока оценивать, каким образом эти объекты связаны друг с другом, — просто внесите их в список.

СОВЕТ

Было бы полезно собрать команду из нескольких человек, в той или иной мере знакомых с системой, которую вы моделируете. После проведения мозгового штурма и в результате реакции на высказываемые при этом идеи вам, скорее всего, удастся разработать более полный и точный набор объектов, чем если бы вы все делали в одиночку.

Получив достаточно полный набор объектов, можете приступать к следующему этапу: определите, как именно эти объекты связаны между собой. Одни объекты будут играть ключевую роль в получении нужных вам результатов, а другие — вспомогательную. Кроме того, вы можете прийти к выводу, что некоторые объекты вообще не являются частью модели.

Шаг 2: идентификация таблиц и столбцов

Ключевые объекты становятся таблицами базы данных. У каждого объекта существует набор *атрибутов*, которые могут стать столбцами соответствующей таблицы. К примеру, во многих базах данных, используемых в производственной среде, хранятся имена клиентов, адреса и другая подобная

информация. Каждый атрибут клиента, такой, например, как имя, улица, город, штат, почтовый индекс, номер телефона и адрес в Интернете, становится столбцом в таблице.

Нет четких правил, позволяющих быстро определить, какие объекты должны стать таблицами и как распределить атрибуты в системе между таблицами. Возможно, у вас есть причины, по которым некоторый атрибут следует отнести к одной таблице, но одновременно есть основания для отнесения того же самого атрибута к другой. Чтобы принять правильное решение, ответьте на два вопроса.

- Какую информацию вы хотите получать из базы данных?
- Каким образом должна использоваться эта информация?

ВНИМАНИЕ!

При проектировании структуры базы данных важно учитывать интересы ее будущих пользователей, а также лиц, которым предстоит принимать решения на основе работы с ней. Если созданная вами "разумная" структура не будет согласована с тем, как люди работают с информацией, то в лучшем случае ваша система может разочаровать своих пользователей, а в худшем выдать неверную информацию. Этого нельзя допустить! Поэтому очень важно не сделать ошибку в принятии решения о структуре таблиц.

Рассмотрим пример, демонстрирующий процесс создания многотабличной базы данных. Скажем, вы только что основали компанию VetLab — клиническую микробиологическую лабораторию, где проводятся анализы проб, присланных из ветеринарных клиник. Конечно, вам хотелось бы иметь следующие данные:

- клиенты;
- выполненные анализы;
- сотрудники;
- заказы;
- результаты.

С каждым из этих объектов связаны определенные атрибуты. Для клиента это название, адрес и другая контактная информация, для анализа — название и стандартная стоимость, для каждого сотрудника — его адрес, телефон, должность и ставка заработной платы. О заказе необходимо знать, кто его сделал, когда и какой именно анализ в нем указан. А что касается результатов анализов, то (помимо самих результатов) нужно знать, предварительные они или окончательные, а также номер заказа.

Шаг 3: точное определение таблиц

Теперь для каждого объекта необходимо точно определить таблицу, а для каждого атрибута — столбец. В табл. 5.1 показано, как можно определить структуру таблиц базы данных `VetLab`, которую я описал в предыдущем разделе.

Таблица 5.1. Таблицы базы данных `VetLab`

Таблица	Столбец
`CLIENT` (фирма-клиент)	`Client Name` (название фирмы-клиента)
	`Address 1` (адрес 1)
	`Address 2` (адрес 2)
	`City` (город)
	`State` (штат)
	`Postal Code` (почтовый индекс)
	`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 индекс)
	`Home Phone` (домашний телефон)
	`Office Extension` (телефон в офисе)
	`Hire Date` (дата приема на работу)

Окончание табл. 5.1

Таблица	Столбец
	Job Classification (классификация работ)
	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 (дата получения результата)
	Preliminary/Final (предварительный/окончательный)

Таблицы базы данных, определенные в табл. 5.1, можно создать с помощью либо инструмента быстрой разработки, либо языка определения данных (DDL), выполнив приведенный ниже код.

```
CREATE TABLE CLIENT (
   ClientName          CHAR (30)    NOT NULL,
   Address1            CHAR (30),
   Address2            CHAR (30),
   City                CHAR (25),
   State               CHAR (2),
   PostalCode          CHAR (10),
   Phone               CHAR (13),
   Fax                 CHAR (13),
   ContactPerson       CHAR (30)
) ;
CREATE TABLE TESTS (
   TestName            CHAR (30)    NOT NULL,
   StandardCharge      CHAR (30)
) ;
```

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

Эти таблицы связаны между собой с помощью следующих общих атрибутов (столбцов).

- **Таблица** `CLIENT` **связана с таблицей** `ORDERS` **столбцом** `ClientName`.
- **Таблица** `TESTS` **связана с таблицей** `ORDERS` **столбцом** `TestName` (`TestOrdered`).
- **Таблица** `EMPLOYEE` **связана с таблицей** `ORDERS` **столбцом** `Employee Name` (`Salesperson`).
- **Таблица** `RESULTS` **связана с таблицей** `ORDERS` **столбцом** `Order Number`.

Если вы хотите сделать таблицу неотъемлемой частью реляционной базы данных, свяжите ее хотя бы еще с одной таблицей базы данных посредством общего столбца. Отношения между таблицами нашей базы данных показаны на рис. 5.1.

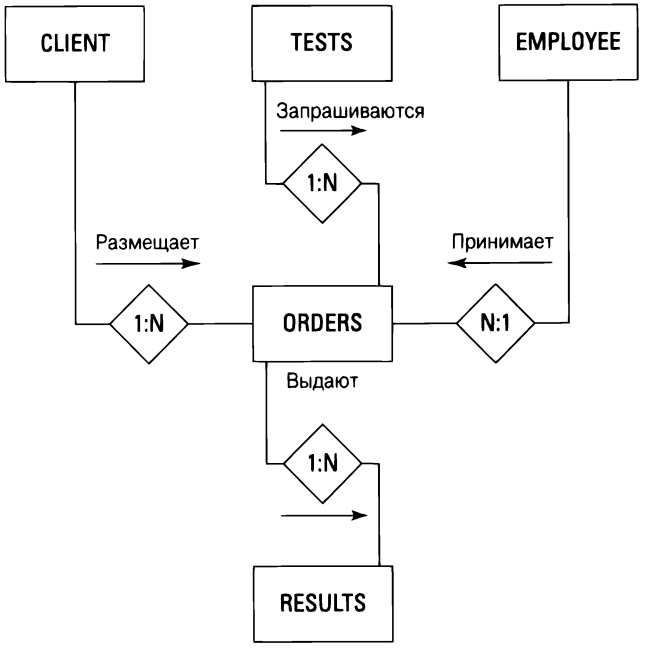


Рис. 5.1. Таблицы базы данных `VetLab` *и связи между ними*

На приведенной схеме продемонстрированы четыре разных отношения (связи) типа "один ко многим". В ромбиках показано максимальное количество элементов на каждой стороне связи. Цифра 1 означает, что на данном конце связи может участвовать только один элемент, а буква N — много элементов.

- Один клиент может сделать множество заказов, но каждый заказ оформляется одним и только одним клиентом.
- Каждый анализ может быть во многих заказах, но каждый заказ оформляется на проведение одного и только одного анализа.
- Каждый заказ принимается одним и только одним сотрудником лаборатории, но каждый сотрудник может принимать (и, как вы догадались, принимает) множество заказов.
- При выполнении каждого заказа может быть получено несколько предварительных результатов и один окончательный, но каждый результат относится только к одному заказу.

Атрибуты, которые связывают одну таблицу с другой, могут называться в них по-разному, но должны быть одного типа. На данном этапе я не подключил никаких ограничений для соблюдения ссылочной целостности, чтобы не перегружать вас слишком большим объемом идей. Мы вернемся к ссылочной целостности немного позже.

Домены, символьные наборы, схемы сортировки и трансляции

Таблицы — главные компоненты базы данных, но есть и другие элементы, играющие свою роль. В главе 1 было введено понятие *домена* табличного столбца как множества всех значений, которые допустимы в данном столбце. Создание с помощью ограничений четко определенных доменов табличных столбцов — важная часть проектирования базы данных.

Реляционными базами данных пользуются не только те, кто разговаривает на английском языке. С ними можно работать, используя и другие языки с другими наборами символов. Даже если база данных создана на основе только английского языка, некоторые приложения все же могут потребовать подключения специальных символьных наборов. Стандарт SQL позволяет задать набор символов, который вы хотите использовать. Более того, в случае необходимости даже для каждого табличного столбца можно указать отдельный символьный набор. В языках, отличных от SQL, подобная гибкость обычно не предусматривается.

Схема сортировки, или *порядок сортировки*, — это набор правил, которые определяют, каким образом сравниваются строки, состоящие из элементов определенного символьного набора. Каждый символьный набор имеет свой порядок сортировки, используемый по умолчанию. Так, по умолчанию для символьного набора ASCII символ B следует после A, а символ C — после B. Поэтому при сравнении считается, что A меньше B, а C больше B. С другой стороны, стандарт SQL позволяет применять к набору символов и другие схемы сортировки. Повторюсь: в других языках подобная гибкость обычно не допускается (это еще одна причина ценить SQL).

Иногда данные в базе кодируются с помощью одного набора символов, но работать с ними необходимо с помощью другого. Предположим, у вас есть данные, закодированные в немецком символьном наборе, но те немецкие буквы, которые не входят в набор ASCII, на вашем принтере не печатаются. И тогда SQL позволяет выполнить трансляцию, т.е. преобразование символьных строк из одного набора в другой. В частности, с помощью трансляции один символ можно преобразовать в два (например, немецкий символ ü — в ASCII-вариант ue) или заменить нижний регистр символов верхним. Можно даже трансформировать один алфавит в другой, например алфавит языка иврит в символы ASCII.

Ускорение работы базы данных с помощью ключей

При проектировании баз данных следует придерживаться такого правила: каждая строка таблицы должна отличаться от любой другой, т.е. каждая строка должна быть уникальной. Иногда с определенной целью — например, для

проведения статистического анализа — вы решите извлечь из своей базы некоторые данные и создать на их основе таблицы, в которых строки не обязательно будут уникальными. В подобной одиночной ситуации правило запрета дублирования может не выполняться, но в общем случае (когда таблицы могут использоваться неоднократно) его нужно строго соблюдать.

Ключ — это атрибут или сочетание атрибутов, которые однозначно определяют каждую строку в таблице. Чтобы получить доступ к строке, необходимо иметь способ, позволяющий отличить эту строку от всех остальных. Поскольку ключи уникальны, они как раз и обеспечивают такой механизм доступа.


ЗАПОМНИ!

Ключ никогда не должен содержать пустое значение. При использовании пустых ключей две строки, содержащие такие ключевые поля, были бы неотличимы друг от друга.

Возвращаясь к примеру с ветеринарной лабораторией, найдем для ключей подходящие поля. В таблице `CLIENT` хороший ключ получится из столбца `ClientName`. Этот ключ позволит отличить любого клиента от всех остальных. Таким образом, ввод значения в этот столбец будет обязательным для всех строк таблицы `CLIENT`. Столбцы `TestName` и `EmployeeName` — хорошие претенденты на звание ключей для таблиц `TESTS` и `EMPLOYEE` соответственно. То же самое относится к столбцам `OrderNumber` и `ResultNumber` таблиц `ORDERS` и `RESULTS` соответственно. При заполнении таблиц нужно обеспечить ввод уникального значения ключа для каждой строки.

Ключи могут быть двух видов: *первичные* и *внешние*. Ключи, упомянутые в предыдущем абзаце, служат примерами первичных ключей. Они гарантируют уникальность строк. О внешних ключах вы узнаете чуть позже.

Первичные ключи

Первичный ключ представляет собой столбец (или сочетание столбцов) в таблице, значения которого (или которых) уникальным образом идентифицируют строки в данной таблице. Чтобы реализовать в базе данных `VetLab` идею ключей, при создании таблицы можно сразу указать ее первичный ключ. В следующем примере для этого будет достаточно одного столбца (при условии, что у фирм-клиентов `VetLab` разные названия).

```
CREATE TABLE CLIENT (
   ClientName        CHAR (30)     PRIMARY KEY,
   Address1          CHAR (30),
   Address2          CHAR (30),
   City              CHAR (25),
   State             CHAR (2),
   PostalCode        CHAR (10),
   Phone             CHAR (13),
```

```
   Fax              CHAR (13),
   ContactPerson    CHAR (30)
) ;
```

Здесь ограничение NOT NULL (недопущение пустых значений), которое было в приведенном выше определении таблицы CLIENT, заменено другим ограничением: PRIMARY KEY (первичный ключ). Второе ограничение подразумевает первое, потому что первичный ключ не может иметь пустое значение.

Несмотря на то что большинство СУБД позволяет создавать таблицы без первичного ключа, важно помнить, что все таблицы базы данных должны его иметь. Поэтому нужно ввести ограничение PRIMARY KEY в таблицы TESTS, EMPLOYEE, ORDERS и RESULTS, как показано в следующем примере.

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

Иногда в таблице ни один отдельный столбец не может гарантировать уникальность строки. В таких случаях можно использовать *составной ключ* — сочетание столбцов, совместное использование которых обеспечит уникальность строки. Представьте, что некоторые клиенты VetLab — это сетевые предприятия, имеющие свои филиалы в нескольких городах. В таком случае одного поля ClientName будет недостаточно, чтобы различить два разных филиала одного и того же клиента. Чтобы решить эту проблему, можно определить следующий составной ключ.

```
CREATE TABLE CLIENT (
   ClientName      CHAR (30)     NOT NULL,
   Address1        CHAR (30),
   Address2        CHAR (30),
   City            CHAR (25)     NOT NULL,
   State           CHAR (2),
   PostalCode      CHAR (10),
   Phone           CHAR (13),
   Fax             CHAR (13),
   ContactPerson   CHAR (30),
   CONSTRAINT BranchPK    PRIMARY KEY
      (ClientName, City)
) ;
```

В качестве альтернативы использованию составного ключа для уникального определения записи можно разрешить СУБД назначить ключ автоматически, как это делает Access, предполагая, что первое поле новой таблицы называется Код и имеет тип Счетчик. У такого ключа нет никакого конкретного назначения. Его единственная задача — быть уникальным идентификатором.

Внешние ключи

Внешний ключ — это столбец или группа столбцов в таблице, ссылающихся на первичный ключ в другой таблице. Внешний ключ сам по себе может и не быть уникальным, но должен однозначно указывать на столбец (столбцы) в той таблице, на которую ссылается.

Если, например, столбец `ClientName` — это первичный ключ таблицы `CLIENT`, то каждая строка этой таблицы должна иметь в этом столбце уникальное значение. В свою очередь, в таблице `ORDERS` столбец `ClientName` является внешним ключом. Этот внешний ключ соответствует первичному ключу таблицы `CLIENT`, но в таблице `ORDERS` он может и не быть уникальным. В действительности вы, конечно же, надеетесь, что внешний ключ не является уникальным. Ведь если бы каждый из ваших клиентов, сделав всего один заказ, больше никогда к вам не обращался, ваш бизнес довольно быстро угас бы. Поэтому вы очень хотели бы, чтобы каждой строке таблицы `CLIENT` соответствовало много строк таблицы `ORDERS`, ведь это означало бы, что почти все ваши клиенты постоянно пользуются вашими услугами.

Следующее определение таблицы `ORDERS` показывает, каким образом в инструкции `CREATE` можно задавать внешние ключи.

```
CREATE TABLE ORDERS (
   OrderNumber INTEGER PRIMARY KEY,
   ClientName        CHAR (30),
   TestOrdered       CHAR (30),
   Salesperson       CHAR (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)
) ;
```

Здесь внешние ключи таблицы `ORDERS` связывают ее с первичными ключами таблиц `CLIENT`, `TESTS` и `EMPLOYEE`.

Работа с индексами

Тема индексов не раскрыта в спецификации SQL, но это не значит, что они являются редкой или даже необязательной частью СУБД. Индексы поддерживаются каждой реализацией SQL, но общего соглашения по их поддержке не существует. В главе 4 объяснялось, как создать индекс с помощью Microsoft

Access. Чтобы разобраться, как индексы реализованы в конкретной СУБД, необходимо обратиться к ее документации.

Что такое индекс

Данные в таблице обычно отображаются в том порядке, в каком они были изначально введены. Но этот порядок совершенно не влияет на тот, в котором данные будут обрабатываться впоследствии. Предположим, вы хотите обрабатывать таблицу `CLIENT` в алфавитном порядке следования значений столбца `ClientName`. Но тогда компьютер должен сначала выполнить сортировку таблицы по столбцу `ClientName`, а на это потребуется определенное время, и чем таблица больше, тем больше времени займет сортировка. А что делать, если в вашей таблице сто тысяч записей? Или миллион? А ведь в некоторых приложениях таблицы такого размера совсем не редкость. Для размещения записей таблицы в нужном порядке (даже при использовании самых быстрых алгоритмов) может потребоваться выполнение порядка 20 млн операций сравнения и миллионы перестановок. Даже при использовании очень быстрого компьютера ждать все-таки придется долго.

Индексы могут оказаться прекрасным средством экономии времени. *Индекс* представляет собой вспомогательную таблицу, которая сопровождает таблицу данных. Каждой строке таблицы данных соответствует определенная строка индексной таблицы. Однако порядок расположения строк в таблице индекса другой. Небольшой пример таблицы данных приведен в табл. 5.2.

Таблица 5.2. Таблица `CLIENT`

ClientName	Address1	Address2	City
Клиника домашних животных	Киевская, 44		Люберцы
Ветеринарный институт	Цветной бульвар, 3		Москва
Ветеринария Булкина	Братиславская, 166	Московская, 32	Киев
Доктор Чайкин	Садово-Спасская, 57		Москва
Санэпидемстанция	Музейная, 4	Николаевское шоссе, 172	Одесса
Дельфинарий	Парк им. Шевченко		Одесса
Клиника животных Бург	Безымянный проезд, 6		Орехово-Зуево
Ветеринарная аптека	Садовая, 4		Шатура

Строки в этой таблице следуют в том порядке, в котором их вводили. Значения в столбце `ClientName` располагаются не по алфавиту, и вообще никакой логики в их последовательности не наблюдается.

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

Таблица 5.3. Индекс для таблицы `CLIENT` (по названию клиента)

ClientName	Указатель на таблицу данных
Ветеринария Булкина	3
Ветеринарный институт	2
Дельфинарий	6
Доктор Чайкин	4
Клиника домашних животных	1
Клиника животных Бург	7
Санэпидемстанция	5
Ветеринарная аптека	8

Индекс содержит поле, которое является его основой (в данном случае это поле `ClientName`), а также указатель на таблицу данных. Значение указателя, расположенное в каждой строке индекса, является номером соответствующей строки в таблице данных.

Зачем нужен индекс

Предположим, вам нужно обрабатывать таблицу в порядке, заданном полем `ClientName`. При этом у вас сформирован индекс, в котором значения этого поля упорядочены по алфавиту. Тогда работу с таблицей данных можно выполнять практически так же быстро, как если бы ее записи сами были выстроены в том же порядке. Ведь используя указатель в индексе, вы можете немедленно переходить от строк индекса к соответствующим строкам таблицы данных.

При использовании индекса время обработки таблицы пропорционально значению N, где N — количество ее строк. А при выполнении той же операции, но без индекса, время обработки таблицы пропорционально $N\log N$, где $\log N$ — логарифм N по основанию 2. В маленьких таблицах разница между этими значениями получается незначительной, но в больших — огромной. Некоторые

операции с большими таблицами без помощи индексов становятся просто невыполнимыми.

Например, у вас есть таблица с миллионом записей (т.е. $N = 1\ 000\ 000$), и на обработку каждой записи уходит одна миллисекунда (одна тысячная доля секунды). Если у вас создан индекс, то на обработку всей таблицы уйдет только 1000 секунд, т.е. меньше 17 минут. Однако, чтобы получить тот же результат обработки без индекса, придется сделать $1\ 000\ 000 \times 20$ проходов по таблице. Таким образом, этот процесс займет 20 000 секунд, т.е. больше пяти с половиной часов. Думаю, вы согласитесь, что разница между семнадцатью минутами и пятью с половиной часами довольно-таки существенная. И это лишь один пример того влияния, которое оказывает индексация на обработку записей.

Поддержка индекса

Если индекс создан, то его необходимо поддерживать. К счастью, вместо вас поддержкой индексов занимается СУБД, которая автоматически обновляет их каждый раз, когда вы обновляете соответствующие таблицы данных. Этот процесс требует дополнительного времени, но затраты окупаются. После того как индекс создан и обеспечен поддержкой СУБД, он всегда готов ускорять обработку ваших данных, и при этом не важно, сколько раз вы будете к нему обращаться.


СОВЕТ

Лучше всего создавать индекс одновременно с соответствующей таблицей данных. Если индекс создается в самом начале и сразу же начинает поддерживаться системой, то тем самым вы избавляете себя от серьезной головной боли в будущем, ведь полная операция требует отдельного и длительного сеанса работы. Постарайтесь предугадать все варианты доступа к данным, а затем *для каждого из них* создайте свой индекс.

В некоторых СУБД существует возможность отключения поддержки индексов. Так придется поступать в некоторых приложениях реального времени, когда вы не можете тратить время — очень уж драгоценный ресурс — на автоматическое обновление индексов. Поэтому иногда обновление индексов выполняют как отдельную операцию, причем не в часы пик.


ВНИМАНИЕ!

Не создавайте индексы для извлечения данных, которые вы вряд ли когда-нибудь будете использовать. С поддержкой индекса связана определенная потеря производительности, потому что эта поддержка выражается в дополнительной операции, выполняемой компьютером каждый раз, когда он модифицирует поле индекса или добавляет (либо удаляет) строку в таблице данных. Чтобы добиться

оптимальной производительности, создавайте только те индексы, которые действительно собираетесь использовать при поиске данных по ключам, и только для таблиц с большим количеством строк, в противном случае индексы лишь ухудшат производительность.


СОВЕТ

Возможно, при составлении некоторого документа (например, ежемесячного или ежеквартального отчета) вам придется выстраивать данные в каком-то необычном порядке. В таком случае перед генерацией отчета создайте соответствующий индекс, а после завершения отчета удалите его, чтобы не обременять СУБД его поддержкой в течение долгого времени между отчетами.

Обеспечение целостности данных

База данных представляет ценность лишь тогда, когда вы уверены в правильности ее содержимого. Например, некорректная информация в медицинских, авиационных и космических базах данных может привести к человеческим жертвам. Неправильные данные в других приложениях, возможно, и не станут причиной таких трагических событий, но нанести ущерб все же могут. Поэтому разработчик базы данных должен гарантировать, что в нее не попадут некорректные данные. Это возможно не всегда, но по крайней мере вполне реально проверять вводимые данные на допустимость. *Поддержка целостности данных* означает гарантию соответствия любых данных, введенных в базу, установленным для них условиям. Например, если поле базы данных имеет тип даты, то СУБД должна отклонить попытку записи в него любых данных, не являющихся допустимой датой.

Возникновению ряда неприятностей нельзя помешать на уровне использования базы данных. Об их заблаговременном предотвращении (перед тем, как они нанесут ущерб базе) должен позаботиться разработчик приложения. Все, кто отвечает за работу с базой данных, должны знать, что угрожает целостности ее данных, и принять соответствующие меры, чтобы свести эти угрозы к нулю.

В базах данных обеспечивается целостность нескольких видов, причем довольно-таки разных. Разными могут быть и неприятности, которые представляют угрозу целостности. В следующих разделах мы рассмотрим три вида целостности: *логическую*, *доменную* и *ссылочную*.

Логическая целостность

Каждая таблица базы данных соответствует какому-либо объекту реального мира. Такой объект может быть физическим или абстрактным, но его существование в определенном смысле не зависит от базы данных. Если таблица полностью согласуется с объектом, который она моделирует, то говорят, что она обладает логической (объектной) целостностью. Чтобы обеспечить такую целостность, необходимо создать в таблице первичный ключ, который однозначно определяет каждую строку таблицы. Если его нет, то не может быть и уверенности в том, что при запросе вы получите нужную вам строку.

Для поддержки логической целостности необходимо, чтобы столбец (или группа столбцов), составляющий первичный ключ, не содержал пустых значений (ограничение `NOT NULL`). Кроме того, для первичного ключа необходимо указать ограничение `UNIQUE`. В одних реализациях SQL такие ограничения вводятся непосредственно в определении таблицы. В других же это делается уже после того, как будет указано, каким образом данные следует добавлять в таблицу, изменять и удалять из нее.

Чтобы сделать первичный ключ одновременно и не пустым, и уникальным, лучше всего использовать ограничение `PRIMARY KEY`, как показано в следующем примере.

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

В качестве альтернативы можно предложить одновременное использование ограничений `NOT NULL` и `UNIQUE`.

```
CREATE TABLE CLIENT (
   ClientName    CHAR (30)    NOT NULL,
   Address1      CHAR (30),
   Address2      CHAR (30),
   City          CHAR (25),
   State         CHAR (2),
   PostalCode    CHAR (10),
   Phone         CHAR (13),
```

```
   Fax           CHAR (13),
   ContactPerson CHAR (30),
   UNIQUE (ClientName)
) ;
```

Доменная целостность

Как правило, невозможно гарантировать корректность конкретного элемента информации в базе данных, но можно хотя бы определить, является ли он допустимым. Многие элементы данных имеют ограниченный набор значений, и если вводится значение, которое не входит в этот набор, то такой ввод должен считаться ошибочным. Например, США состоит из пятидесяти штатов, округа Колумбия, Пуэрто-Рико и еще нескольких территорий. Каждой из этих территорий присвоен двухсимвольный код, распознаваемый почтовой службой США. И если в базе данных существует столбец `State` (штат), то доменную целостность можно обеспечить, потребовав, чтобы любое значение в этом столбце являлось одним из распознаваемых двухсимвольных кодов. Если оператор вводит код, не входящий в список допустимых, он тем самым нарушает *доменную целостность*. Контролируя же соблюдение доменной целостности, вы вправе отклонить любую операцию, которая вызывает нарушение этой целостности.

Опасения за доменную целостность возникают при вводе в таблицу новых данных с помощью инструкции `INSERT` или `UPDATE`. Домен для столбца можно создать с помощью инструкции `CREATE DOMAIN`, причем это нужно сделать до использования данного столбца в инструкции `CREATE TABLE`. Рассмотрим пример создания таблицы для команд Главной лиги бейсбола.

```
CREATE DOMAIN LeagueDom CHAR (8)
   CHECK (VALUE IN ('Американская', 'Национальная'));
CREATE TABLE TEAM (
   TeamName       CHAR (20)     NOT NULL,
   League         LeagueDom     NOT NULL
) ;
```

Домен для столбца `League` состоит из двух разрешенных значений: `'Американская'` и `'Национальная'`. СУБД не позволит успешно выполнить операцию ввода или обновления в таблице `TEAM`, если в столбце `League` добавляемой записи окажется значение, отличное от `'Американская'` или `'Национальная'`.

Ссылочная целостность

Даже если в базе данных для каждой таблицы обеспечивается логическая и доменная целостность, этой базе все равно грозят неприятности, если связь одной таблицы не согласована с другой. В большинстве хорошо

спроектированных баз данных каждая таблица содержит как минимум один столбец, который ссылается на столбец другой таблицы в той же базе. Такие ссылки играют важную роль в поддержании общей целостности базы данных. Но одновременно те же ссылки создают вероятность возникновения аномалий обновления. *Аномалии обновления* — это проблемы, которые могут возникнуть после обновления значений в строке таблицы базы данных.

Проблемы отношений между родительскими и дочерними таблицами

Отношения между таблицами, как правило, не являются равноправными — обычно одна таблица зависит от другой. Предположим, например, что у вас есть база данных с таблицами `CLIENT` (клиент) и `ORDERS` (заказы). Скорее всего, вы введете в таблицу `CLIENT` данные о фирме-клиенте еще до того, как ею будут сделаны какие-либо заказы. Однако в таблицу `ORDERS` нельзя ввести ни одного заказа, если в таблице `CLIENT` не будет записи для клиента, сделавшего этот заказ. В данном примере таблица `ORDERS` зависит от таблицы `CLIENT`. Такой вид отношений между таблицами часто называют *родительско-дочерними связями*. В этом контексте таблица `CLIENT` является родительской, а `ORDERS` — дочерней. Дочерний объект базы данных зависит от родительского.


СОВЕТ

Обычно первичный ключ родительской таблицы — это столбец (или группа столбцов), который одновременно существует и в дочерней таблице, где является внешним ключом. Имейте в виду, что внешний ключ может не быть уникальным.

Аномалии обновления (между родительской и дочерней таблицами) возникают в нескольких случаях. Например, фирма-клиент перестала делать у вас заказы, и вы хотите удалить связанную с ней информацию из базы данных. Но если она уже делала у вас некоторые заказы (что зафиксировано в таблице `ORDERS`), то удаление ее данных из таблицы `CLIENT` может вызвать проблемы. Дело в том, что в этом случае в дочерней таблице `ORDERS` остались бы записи, для которых не существовало бы соответствующих записей в родительской таблице `CLIENT`. Сходные проблемы могут возникнуть и тогда, когда в дочернюю таблицу вставляется запись, а соответствующее добавление в родительскую таблицу еще не сделано.


ЗАПОМНИ!

Все изменения первичного ключа, вносимые в любую строку родительской таблицы, должны отражаться в соответствующих внешних ключах всех дочерних таблиц. Если этого не происходит, возникают аномалии обновления.

Осторожно: каскадное удаление

Большинства проблем, связанных со ссылочной целостностью, можно избежать, если тщательно управлять процессом обновления. В некоторых случаях необходимо реализовать *каскадное* удаление, т.е. распространить этот процесс с родительской таблицы на дочерние. Чтобы удаление строки родительской таблицы сделать каскадным, нужно и во всех дочерних таблицах удалить строки, значение внешнего ключа которых равно значению первичного ключа удаляемой строки родительской таблицы. Рассмотрим следующий пример.

```
CREATE TABLE CLIENT (
   ClientName      CHAR (30)     PRIMARY KEY,
   Address1        CHAR (30),
   Address2        CHAR (30),
   City            CHAR (25)     NOT NULL,
   State           CHAR (2),
   PostalCode      CHAR (10),
   Phone           CHAR (13),
   Fax             CHAR (13),
   ContactPerson   CHAR (30)
) ;

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

CREATE TABLE EMPLOYEE (
   EmployeeName        CHAR (30)       PRIMARY KEY,
   ADDRESS1            CHAR (30),
   Address2            CHAR (30),
   City                CHAR (25),
   State               CHAR (2),
   PostalCode          CHAR (10),
   HomePhone           CHAR (13),
   OfficeExtension     CHAR (4),
   HireDate            DATE,
   JobClassification   CHAR (10),
   HourSalComm         CHAR (1)
) ;

CREATE TABLE ORDERS (
   OrderNumber       INTEGER       PRIMARY KEY,
   ClientName        CHAR (30),
   TestOrdered       CHAR (30),
   Salesperson       CHAR (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 (CONSTRAINT) NameFK делает поле ClientName таблицы ORDERS внешним ключом, который указывает на столбец ClientName таблицы CLIENT. Если в таблице CLIENT вы удалите какую-нибудь строку, то в таблице ORDERS будут автоматически удалены все строки, в столбце ClientName которых находится 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 можно сказать и о внешних ключах таблицы ORDERS, которые ссылаются на первичные ключи таблиц TESTS и EMPLOYEE.

Альтернативные способы контроля аномалий обновления

В некоторых случаях вместо каскадного удаления вам хотелось бы заменить внешний ключ дочерней таблицы пустым значением NULL. Проанализируем следующий вариант предыдущего примера.

```
CREATE TABLE ORDERS (
   OrderNumber         INTEGER      PRIMARY KEY,
   ClientName          CHAR (30),
   TestOrdered         CHAR (30),
   SalesPerson         CHAR (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 (который получит ее учетную запись), но сейчас удаление строки с ее данными из таблицы EMPLOYEE приводит к замене значений столбца

Salesperson во всех строках таблицы ORDERS, соответствующих ее заказам, значением NULL.

СОВЕТ

Существуют и другие способы предохранения базы данных от несогласованных данных.

- **Можно запретить добавление строк в дочернюю таблицу, пока в родительской таблице не появится соответствующая им запись.** Тем самым вы предотвратите появление "висячих строк" в дочерней таблице. Это позволит легко поддерживать согласованность таблиц.
- **Можно запретить изменение первичного ключа таблицы.** В таком случае можно не беспокоиться об обновлении внешних ключей в других таблицах, которые зависят от первичного ключа.

Когда кажется, будто все безопасно

Единственное, что точно ожидает любую базу данных, — так это изменения. Вы создаете базу данных, наполняете ее таблицами, полями, ограничениями и данными. Через некоторое время руководство (которому виднее) требует внести изменения в структуру базы данных. Как добавить новый столбец в уже существующую таблицу? Как удалить из таблицы столбец, который больше не нужен? Здесь вам на помощь придет SQL.

Добавление столбца в существующую таблицу

Предположим, в вашей компании была введена новая традиция — устраивать вечеринки в дни рождения сотрудников. Координатора таких вечеринок (для их плановой организации) необходимо заранее предупреждать. Для этого в таблицу EMPLOYEE нужно добавить столбец Birthday (день рождения). Какие проблемы? Это можно легко сделать, используя инструкцию ALTER TABLE, как показано в следующем примере.

```
ALTER TABLE EMPLOYEE
  ADD COLUMN Birthday DATE ;
```

Теперь остается только добавить в созданный столбец (для каждой строки) информацию, и компания готова к вечеринкам.

Удаление столбца из существующей таблицы

Теперь предположим, что в компании наступили тяжелые времена и тратить деньги на вечеринки стало непозволительной роскошью. В этом случае информация о днях рождения сотрудников становится бесполезной. И опять вам на помощь придет все та же инструкция ALTER TABLE.

```
ALTER TABLE EMPLOYEE
 DROP COLUMN Birthday ;
```

Что ж, будем надеяться, что вам никогда не придется столкнуться с тяжелыми временами.

Потенциальные проблемы

Угрозы целостности данных приходят с самых разных сторон. Одни из этих неприятностей возникают только в многотабличных базах, в то время как другие могут произойти даже в базах, имеющих одну таблицу. Необходимо уметь распознавать эти угрозы и сводить вероятность их появления к минимуму.

Ввод некорректных данных

В документах или файлах с исходной информацией, которыми вы пользуетесь для заполнения своей базы, могут содержаться неверные данные. Они могут представлять собой искаженный вариант правильных данных или оказаться совсем "из другой оперы". Проверка принадлежности к диапазону допустимых значений покажет, имеют ли данные доменную целостность. Такие проверки помогают предотвратить часть проблем, но далеко не все. Если некорректные данные не выходят из допустимого диапазона, то в результате такой проверки проблему выявить не удастся.

Ошибки оператора

Исходные данные могут быть корректными, но оператор при вводе в компьютер мог переписать их неправильно. Ошибка такого рода способна привести к тем же проблемам, что и ввод некорректных данных, и некоторые из средств их предотвращения оказываются теми же самыми. Проверка принадлежности к диапазону допустимых значений в этом случае полезна, но не является панацеей. Есть еще одно решение. Оно состоит в том, чтобы все вводимые данные независимо проверялись еще одним оператором. Такой подход обходится дорого, потому что предполагает удвоение числа сотрудников и затраченного на работу времени. Однако, когда целостность данных важна, дополнительные усилия и затраты себя окупают.

Механическое повреждение

При механических повреждениях устройств, например жесткого диска, данные в таблицах могут быть испорчены. И главная защита здесь — регулярное резервное копирование.

Злой умысел

Не следует исключать возможность умышленной порчи данных. Ваша первая линия обороны — это запрещение доступа к базе данных потенциально опасным пользователям. Также нужно предоставлять доступ пользователям только к тем данным, которые им действительно нужны. Вторая линия обороны — хранение резервных копий данных в безопасном месте. Периодически проверяйте, насколько защищена ваша база данных.

Избыточность данных

В иерархических моделях баз данных *избыточность данных* — это существенная проблема. Впрочем, она не обходит стороной и реляционные базы. При наличии информационной избыточности не только зря расходуется дисковая память и снижается производительность, но и создаются условия для серьезного повреждения 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, которая 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. Поэтому желательно сводить избыточность данных к минимуму.


СОВЕТ

Некоторая избыточность не повредит, если первичный ключ одной таблицы служит внешним ключом другой. В то же время старайтесь избегать любой другой избыточности.


ВНИМАНИЕ!

После удаления избыточности из базы данных ее производительность может оказаться неудовлетворительной. Часто операторы баз данных преднамеренно создают некоторую избыточность для ускорения обработки информации. Например, в базе данных `VetLab` таблица `ORDERS` для идентификации источника заказа содержит только имена клиентов. При подготовке заказа для получения адреса клиента вы должны связать таблицы `ORDERS` и `CLIENT`. Если такое объединение приводит к замедлению программы печати счетов-фактур, то можно допустить избыточность и продублировать адреса клиентов и в таблице заказов. В этом случае избыточность ускорит печать, но приведет к замедлению (и усложнению) операций обновления адресов клиентов.


СОВЕТ

Распространенной практикой является изначальное проектирование баз данных с малой избыточностью и высокой степенью нормализации. Впоследствии, если обнаруживается, что важные приложения

работают медленно, избирательно добавляется избыточность и снижается уровень нормализации. Ключевое слово здесь — "избирательно". Добавляемая избыточность должна служить конкретной цели и не должна добавлять больше проблем, чем может устранить. Более подробно этот вопрос мы рассмотрим в разделе, посвященном нормализации.

Превышение технических возможностей базы данных

База данных может работать безукоризненно годами, а затем вдруг начать выдавать ошибки, которые постепенно становятся все более и более серьезными. Это является сигналом о том, что достигнут предел мощности системы. Существуют определенные предельные количества строк в таблицах, а также предельные количества столбцов, ограничений и других характеристик. Старайтесь контролировать соответствие текущего размера и содержимого базы данных техническим характеристикам вашей СУБД. Если вы обнаружили, что приближаетесь к предельному значению какой-то характеристики, займитесь улучшением возможностей системы. Вы также можете заархивировать старые данные, к которым больше не обращаетесь, а затем удалить их из базы.

Ограничения

Ранее мы упоминали ограничения как механизм, благодаря которому в столбец таблицы могут быть введены только данные из домена этого столбца. *Ограничение* — это правило, за выполнением которого следит СУБД. После создания базы данных можно включить в определения таблиц ограничения (такие, например, как `NOT NULL`). И тогда СУБД проследит за тем, чтобы нельзя было выполнить транзакцию, если она нарушает какое-либо ограничение.


ЗАПОМНИ!

Существуют три типа ограничений.

- **Ограничение на значения столбца** накладывает определенное условие на столбец таблицы.
- **Ограничение на таблицу** относится к таблице в целом.
- **Утверждение** — это ограничение, которое может влиять на несколько таблиц.

Ограничение на значения столбца

Ограничение на значения столбца продемонстрировано в следующей инструкции DDL.

```
CREATE TABLE CLIENT (
   ClientName      CHAR (30)       NOT NULL,
```

```
   Address1       CHAR (30),
   Address2       CHAR (30),
   City           CHAR (25),
   State          CHAR (2),
   PostalCode     CHAR (10),
   Phone          CHAR (13),
   Fax            CHAR (13),
   ContactPerson  CHAR (30)
) ;
```

В этой инструкции ограничение NOT NULL, примененное к столбцу ClientName, определяет недопустимость пустого значения для этого столбца. Другое ограничение, которое можно применять к столбцам, — это UNIQUE. Оно указывает на то, что все значения, находящиеся в столбце, должны быть уникальными. Ограничение CHECK особенно полезно тем, что можно задавать в качестве аргумента любое корректное выражение. Рассмотрим пример.

```
CREATE TABLE TESTS (
   TestName         CHAR (30)       NOT NULL,
   StandardCharge   NUMERIC (6, 2)
      CHECK (StandardCharge >= 0.0
        AND StandardCharge <= 200.0)
) ;
```

В базе данных VetLab стандартная стоимость проведения 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 один способ установки того же ограничения:

```
CHECK (StandardCharge BETWEEN 0.0 AND 200.0)
```

Ограничения на таблицу

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 объединению двух ограничений на значения столбца: NOT NULL и UNIQUE. Это ограничение, как показано в следующем примере, можно задавать в инструкции CREATE.

```
CREATE TABLE CLIENT (
   ClientName     CHAR (30)     PRIMARY KEY,
   Address1       CHAR (30),
   Address2       CHAR (30),
   City           CHAR (25),
   State          CHAR (2),
```

```
   PostalCode      CHAR (10),
   Phone           CHAR (13),
   Fax             CHAR (13),
   ContactPerson   CHAR (30)
) ;
```

Именованные ограничения (например, `NameFK` в примере из раздела "Осторожно: каскадное удаление") могут иметь дополнительные применения. Предположим, вы хотите выполнить массовую загрузку в свою таблицу `PROSPECT` данных о нескольких тысячах потенциальных клиентов. У вас есть файл, который содержит данные о возможных клиентах, проживающих в США (их большинство) и в Канаде (их совсем немного), причем данные второй категории разбросаны по всему файлу. Вам нужно ограничить свою таблицу `PROSPECT` так, чтобы она включала записи, относящиеся только к жителям США, но вы не хотите, чтобы процесс массовой загрузки данных прерывался каждый раз, когда попадется одна из канадских записей. (Канадские почтовые индексы включают буквы и цифры, а американские — только цифры.) В таком случае можно не вводить в действие ограничение на столбец `PostalCode` до тех пор, пока массовая загрузка данных не будет завершена, и восстановить применение этого ограничения позже.

Изначально таблица `PROSPECT` была создана с использованием следующей инструкции `CREATE TABLE`.

```
CREATE TABLE PROSPECT (
   ClientName      CHAR (30)      PRIMARY KEY,
   Address1        CHAR (30),
   Address2        CHAR (30),
   City            CHAR (25),
   State           CHAR (2),
   PostalCode      CHAR (10),
   Phone           CHAR (13),
   Fax             CHAR (13),
   ContactPerson   CHAR (30),
   CONSTRAINT Zip CHECK (PostalCode BETWEEN 0 AND 99999)
) ;
```

До начала массовой загрузки данных можно отключить применение ограничения `Zip`.

```
ALTER TABLE PROSPECT
   CONSTRAINT Zip NOT ENFORCED;
```

Но после завершения загрузки можно восстановить действие этого ограничения.

```
ALTER TABLE PROSPECT
   CONSTRAINT Zip ENFORCED;
```

Теперь можно исключить из таблицы все строки, которые не удовлетворяют упомянутому ограничению.

```
DELETE FROM PROSPECT
      WHERE PostalCode NOT BETWEEN 0 AND 99999 ;
```

Утверждения

Утверждение задает ограничение не для одной, а для нескольких таблиц. В следующем примере для создания утверждения применяется условие поиска, составленное для столбцов из двух таблиц.

```
CREATE TABLE ORDERS (
   OrderNumber  INTEGER      NOT NULL,
   ClientName   CHAR (30),
   TestOrdered  CHAR (30),
   Salesperson  CHAR (30),
   OrderDate    DATE
) ;

CREATE TABLE RESULTS (
   ResultNumber INTEGER      NOT NULL,
   OrderNumber  INTEGER,
   Result       CHAR (50),
   DateOrdered  DATE,
   PrelimFinal  CHAR (1)
) ;

CREATE ASSERTION
   CHECK (NOT EXISTS (SELECT * FROM ORDERS, RESULTS
   WHERE ORDERS.OrderNumber = RESULTS.OrderNumber
     AND ORDERS.OrderDate > RESULTS.DateReported)
) ;
```

Благодаря этому утверждению дата заказа анализа всегда будет предшествовать дате его выполнения.

Нормализация базы данных

Существуют разные способы организации данных, и какие-то вам покажутся в чем-то лучше остальных: одни — логичнее, другие — проще. Есть и такие, которые позволяют предотвратить некоторые нестыковки при использовании базы данных. Увы, изменение базы данных открывает простор для целого ряда неприятностей, поэтому нужно разобраться со способами их устранения.

Аномалии изменения и нормальные формы

Если организовать структуру базы данных неправильно, то она может стать жертвой множества различных неприятностей (которые называются *аномалиями изменения*). Чтобы их предотвратить, нужно нормализовать структуру базы данных. Нормализация обычно влечет за собой разделение в базе данных одной таблицы на две.

Аномалии изменения называются так потому, что генерируются в таблице базы данных в результате каких-либо изменений: при добавлении данных, их модификации или удалении.

Чтобы показать, каким образом могут проявляться аномалии изменения, рассмотрим таблицу, приведенную на рис. 5.2.

SALES

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

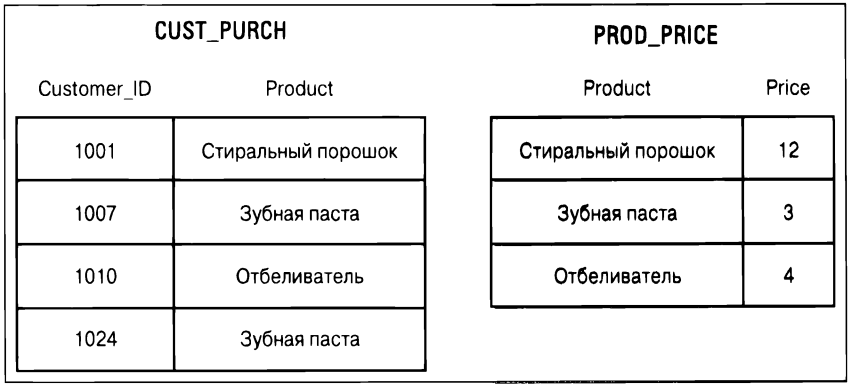
Рис. 5.2. Эта таблица `SALES` является источником аномалий изменения

Допустим, ваша компания продает моющие средства для дома и предметы личной гигиены, причем за один и тот же товар все покупатели платят одинаково. Все данные содержатся в таблице `SALES` — о продажах стирального порошка, зубной пасты и т.п. Теперь предположим, что покупатель 1001 уехал и больше ничего у вас не покупает. И так как он не собирается больше ничего приобретать, вам уже не интересно, что же он покупал раньше. Это наводит вас на мысль удалить его строку в таблице. Но если вы это сделаете, то не только потеряете данные о том, что покупатель 1001 покупал стиральный порошок, но и о том, что стиральный порошок стоит 12 долларов. Такая ситуация называется *аномалией удаления*. Удалив одни данные (о том, что покупатель 1001 покупал стиральный порошок), вы непреднамеренно 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 дезодорант стоимостью 2 доллара. Но эти

данные вы не сможете поместить в таблицу SALES до тех пор, пока дезодорант не потребуется какому-либо покупателю.

Проблема представленной выше таблицы SALES заключается в том, что эта таблица слишком универсальна. В ней есть данные не только о том, что именно приобрели у вас покупатели, но и о том, сколько стоят купленные товары. Эту таблицу необходимо разбить на две, и каждая из них будет посвящена только одной теме (рис. 5.3).



CUST_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 разделена на две

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

- таблица CUST_PURCH (покупки) содержит данные только о совершенных покупках;
- таблица PROD_PRICE (цена товара) содержит данные только о ценах 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 отдельной таблице PROD_PRICE. К тому же данные о дезодоранте теперь можно занести в таблицу PROD_PRICE независимо от того, купил кто-то этот товар или нет. Информация о покупках хранится уже не в этой таблице, а в таблице CUST_PURCH.

Нормализацией называется процесс разделения одной таблицы на множество других, каждая из которых посвящена отдельной теме. Операция нормализации, которая решает одну задачу, может не повлиять на другие. И чтобы получить таблицы, каждая из которых посвящена единственной теме, возможно, придется выполнить несколько последовательных операций нормализации. У каждой таблицы базы данных должна быть одна и только одна главная тема. Правда, иногда довольно сложно определить наличие у таблицы двух или более тем.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Таблицы можно классифицировать по видам тех аномалий изменения, которым они подвержены. В своей статье, опубликованной в 1970 году (первой, где была описана реляционная модель), доктор Кодд диагностировал три источника аномалий изменения и для "лечения" от этих аномалий выписал три "лекарства". Это первая, вторая и третья *нормальные формы* (1НФ, 2НФ, 3НФ). В последующие годы доктор Кодд (и не только он) открыл как другие виды аномалий, так и средства против них — новые нормальные формы. Нормальная форма Бойса — Кодда (НФБК, или BCNF), четвертая нормальная форма (4НФ) и пятая нормальная форма (5НФ) — каждая из них обеспечивала еще более высокую защиту от аномалий изменения, чем предшественницы. И только в 1981 году появилась статья с описанием доменно-ключевой нормальной формы (ДКНФ, или DKNF). Ее автор — Рональд Фейджин. Эта последняя нормальная форма *гарантирует* полное отсутствие в таблице аномалий изменения.

Нормальные формы являются *вложенными* в том смысле, что таблица, приведенная ко второй нормальной форме, автоматически приведена и к первой. Аналогично таблица, которая приведена к 3НФ, приведена к 2НФ, 1НФ и т.д. Для большинства приложений приведения базы данных к 3НФ вполне достаточно, чтобы обеспечить этой базе высокую степень целостности. Впрочем, чтобы была абсолютная уверенность в целостности базы данных, необходимо привести ее к ДКНФ.

СОВЕТ

После проведения максимально возможной нормализации базы данных для увеличения ее производительности иногда требуется выполнить выборочную денормализацию. В этом случае нужно полностью отдавать себе отчет, какие аномалии станут возможными.

Первая нормальная форма

Чтобы соответствовать первой нормальной форме (1НФ), таблица должна обладать следующими качествами.

- Быть двухмерной, т.е. состоять из строк и столбцов.
- В каждой строке должны находиться данные, соответствующие одному объекту или его части.
- В каждом столбце должны находиться данные, относящиеся к одному из атрибутов описываемого объекта.

- В каждой табличной ячейке (на пересечении строки и столбца) должно находиться только одно значение.
- В каждом столбце должны быть только однотипные данные. Если, например, в какой-либо строке столбца находится фамилия сотрудника, то и во всех остальных строках этого столбца также должны быть фамилии сотрудников.
- У каждого столбца должно быть уникальное имя.
- Никакие две строки не могут быть идентичными (т.е. каждая строка должна быть уникальной).
- Порядок расположения столбцов и строк не имеет значения.

Таблица (отношение), находящаяся в первой нормальной форме, хотя и может иметь "иммунитет" к некоторым видам аномалий изменения, все равно остается подверженной остальным. Первой нормальной форме соответствует таблица `SALES` (см. рис. 5.2), но, как уже говорилось, она подвержена аномалиям удаления и вставки. Таким образом, первая нормальная форма может быть полезной в одних приложениях и ненадежной — в других.

Вторая нормальная форма

Чтобы понять вторую нормальную форму, необходимо изучить понятие функциональной зависимости. *Функциональная зависимость* — это связь между атрибутами. Один атрибут функционально зависит от другого, если значение последнего определяет значение первого. Проще говоря, значение первого атрибута можно определить, зная значение второго.

Предположим, например, что у таблицы существуют следующие атрибуты (столбцы): `StandardCharge` (стандартная стоимость), `NumberOfTests` (число анализов) и `TotalCharge` (общая стоимость), которые связаны следующей формулой:

```
TotalCharge = StandardCharge * NumberOfTests
```

В данном случае столбец `TotalCharge` функционально зависим от двух других: `StandardCharge` и `NumberOfTests`. Если известны значения `StandardCharge` и `NumberOfTests`, то можно однозначно определить значение `TotalCharge`.

Каждая таблица в первой нормальной форме должна иметь уникальный первичный ключ, который может состоять из одного или множества столбцов. Ключ, состоящий из множества столбцов, называется *составным*. Чтобы таблица считалась приведенной ко второй нормальной форме (2НФ), все ее неключевые атрибуты (столбцы) должны зависеть от ключа, взятого в целом. Любое 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. Предположим, у вас есть таблица, в которой некоторые неключевые атрибуты не зависят от всех компонентов ключа. Тогда вам лучше разбить эту таблицу на несколько новых, чтобы в каждой из них все неключевые атрибуты зависели от всех компонентов первичного ключа.

Не правда ли, достаточно запутанно? Для наглядности рассмотрим пример. Пусть существует таблица `SALES_TRACK` (данные о продажах), подобная таблице `SALES` (см. рис. 5.2). Но, вместо того чтобы записывать для каждого покупателя только одну покупку, вы добавляете для него строку каждый раз, когда он впервые покупает какой-либо товар. Еще одно отличие состоит в том, что первые покупатели товара (те, у которых значения столбца `Customer_ID` находятся в диапазоне 1001–1007) получают скидку. Некоторые строки этой таблицы показаны на рис. 5.4.

SALES_TRACK

Customer_ID	Product	Price
1001	Стиральный порошок	11.00
1007	Зубная паста	2.70
1010	Отбеливатель	4.00
1024	Зубная паста	3.00
1010	Стиральный порошок	12.00
1001	Зубная паста	2.70

Рис. 5.4. В таблице `SALES_TRACK` составной ключ состоит из столбцов `Customer_ID` и `Product`

На рис. 5.4 столбец `Customer_ID` не определяет строку однозначно. В двух строках его значения равны 1001, еще в двух — 1010. В то же время строку однозначно определяет комбинация столбцов `Customer_ID` и `Product`. Вместе эти столбцы и являются составным ключом.

Если бы не существовало условие, что одни покупатели имеют скидку, а другие — нет, то таблица не находилась бы во второй нормальной форме, поскольку столбец `Price` (цена), являющийся неключевым атрибутом, зависел бы только от столбца `Product` (товар). Однако в данном случае часть покупателей

имеет скидку, и значение Price зависит и от Customer_ID, и от Product. Таким образом, таблица все же находится во второй нормальной форме.

Третья нормальная форма

Как бы там ни было, остаются аномалии изменения, против которых таблицы во второй нормальной форме беззащитны. Эти аномалии связаны с транзитивными зависимостями.

Транзитивная зависимость возникает тогда, когда один атрибут зависит от второго, а второй, в свою очередь, от третьего. Удаления в таблице, имеющей такие зависимости, могут вызвать нежелательную потерю информации. Отношение в третьей нормальной форме — это отношение во второй нормальной форме, не имеющее транзитивных зависимостей.

Давайте вернемся к таблице SALES (см. рис. 5.2), которая, как вам известно, находится в первой нормальной форме. Пока существует ограничение, запрещающее ввод нескольких строк для одного значения Customer_ID (идентификатор покупателя), первичный ключ может состоять только из одного атрибута, и следовательно, таблица находится во второй нормальной форме. Тем не менее эта таблица все равно подвержена аномалиям. К примеру, что произойдет, если покупателю 1010 не понравится отбеливатель и он вернет свою покупку, получив назад деньги? В этом случае вам нужно удалить из таблицы третью строку, в которой записаны данные о том, что покупатель 1010 приобрел отбеливатель. Именно в этом месте возникает проблема. Если строка будет удалена, то также будут удалены данные о том, что цена отбеливателя составляет 4 доллара. Такая ситуация служит примером транзитивной зависимости. Атрибут Price (цена) зависит от атрибута Product (товар), который, в свою очередь, зависит от первичного ключа Customer_ID.

Проблема транзитивной зависимости устраняется посредством разделения таблицы SALES на две. В результате две таблицы, CUST_PURCH (покупки) и PROD_PRICE (цена товара), показанные на рис. 5.3, входят в состав базы данных, соответствующей третьей нормальной форме.

Доменно-ключевая нормальная форма (ДКНФ)

После того как база данных была приведена к третьей нормальной форме, шансы на возникновение аномалий изменения были практически исключены. Практически, но не абсолютно. Для исправления этих оставшихся проблем как раз и предназначены другие (помимо первых трех) нормальные формы. Это нормальная форма Бойса — Кодда (НФБК), четвертая нормальная форма

(4НФ) и пятая нормальная форма (5НФ). Каждая из этих форм ликвидирует угрозу конкретной аномалии изменения, но не дает гарантии защиты от всех возможных аномалий. Такую гарантию дает только доменно-ключевая нормальная форма (ДКНФ).

Отношение находится в *доменно-ключевой нормальной форме (ДКНФ)*, если каждое ограничение в этом отношении является логическим следствием определений ключей и доменов. Ограничением в этом определении называется любое правило, которое определено достаточно точно для того, чтобы его можно было проверить. *Ключ* — это уникальный идентификатор строки таблицы, а *домен* — набор допустимых значений некоторого атрибута.

Снова взгляните на таблицу `SALES` (см. рис. 5.2), находящуюся в 1НФ. Это нам нужно для того, чтобы понять, каким образом привести базу данных к ДКНФ.

Таблица	`SALES(Customer_ID, Product, Price)`
Ключ	`Customer_ID`
Ограничения	1. `Customer_ID` **определяет** `Product` 2. `Product` **определяет** `Price` 3. `Customer_ID`: **целое число > 1000**

Как заставить работать ограничение 3 (атрибут `Customer_ID` должен выражаться целым числом, б*о*льшим 1000)? Можно так определить домен `Customer_ID`, чтобы он включал это ограничение. В таком случае ограничение становится логическим следствием домена столбца `Customer_ID`. Значение `Product` зависит от `Customer_ID`, а `Customer_ID` — это ключ, так что трудностей с ограничением 1 не будет, поскольку оно является логическим следствием определения ключа. Однако возникает проблема с ограничением 2: значение `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 будем использовать столбец `Customer_ID`, а в другой — `Product`. Такая схема была показана на рис. 5.3. Мы видим, что эта база данных соответствует не только 3НФ, но и ДКНФ.

Проектируйте базы данных так, чтобы они по возможности находились в ДКНФ. В этом случае определения ключей и доменов будут включать все необходимые ограничения, и аномалии изменений станут невозможными. Если же структура базы данных спроектирована

так, что ее нельзя привести к ДКНФ, то ограничения придется встроить в приложение, которое использует эту базу данных. Сама же база данных не будет гарантировать соблюдения ограничений.

Денормализация

Определенная степень денормализации порой тоже является благом. Возможно, вы увлеклись нормализацией, и вас занесло слишком далеко. Ведь базу данных можно разделить на такое количество таблиц, что она станет громоздкой и неэффективной. Ее работа может просто застопориться. Довольно часто оптимальная структура может быть немного денормализованной. На практике производственные базы данных (особенно крупные) никогда не нормализуют до уровня ДКНФ. В то же время, чтобы минимизировать риск повреждения данных вследствие аномалий изменений, следует нормализовать проектируемую базу данных до максимально приемлемого уровня.

После нормализации испытайте свою базу данных на ряде выборок данных. Если производительность вас не устраивает, подумайте, а нельзя ли с помощью небольшой денормализации повысить производительность, не жертвуя при этом целостностью. Осторожно добавляя избыточность в некоторых стратегически важных местах и проводя денормализацию, вы получите базу данных, которая будет одновременно и эффективной, и защищенной от аномалий.

Хранение и извлечение данных

В ЭТОЙ ЧАСТИ...

- Управление данными
- Контроль времени
- Обработка значений
- Создание запросов

Глава 6

Манипулирование содержимым базы данных

В ЭТОЙ ГЛАВЕ...

- Работа с данными
- Извлечение из таблицы нужных данных
- Отображение информации, полученной из одной или нескольких таблиц
- Обновление информации, находящейся в таблицах и представлениях
- Добавление новой строки в таблицу
- Изменение всех или части данных, находящихся в строке таблицы
- Удаление строки таблицы

В главах 3 и 4 вы узнали о том, что для обеспечения целостности информации, хранящейся в базе данных, очень важно сделать все возможное, чтобы ее структура была безупречна. Впрочем, для пользователя интерес представляет не структура базы данных, а ее содержимое, т.е. сами данные. И с ними пользователь должен иметь возможность выполнять определенный набор действий: добавлять их в таблицы, извлекать и отображать, изменять, а также удалять из таблиц.

В принципе, манипулировать данными достаточно просто. Вы можете добавить в таблицу одну или сразу несколько строк данных. Изменение, удаление и извлечение строк из таблиц баз данных также не представляет особого труда. Главная трудность в работе с базами данных — *отобрать* строки, которые следует изменить, удалить или извлечь. Нужные данные могут находиться в базе данных, содержащей гигантский массив совершенно *ненужной* вам информации. Если вы сможете с помощью инструкции SELECT точно указать, что именно вам нужно, то всю остальную работу по поиску данных компьютер возьмет на себя. Лично мне кажется, что манипулирование данными с помощью SQL проще пареной репы — как извлечение, так и обновление с удалением. Возможно, с "репой" я несколько переборщил, так что начнем с простой операции извлечения данных.

Извлечение данных

Задачи, которые выполняют пользователи с базами данных, чаще всего сводятся к извлечению из них необходимой информации. К примеру, вы захотите получить содержимое одной конкретной строки, находящейся в таблице среди тысяч других. Или вы решите просмотреть строки, удовлетворяющие некоторому условию или комбинации условий. Не исключено, что вам потребуется извлечь из таблицы все ее строки. Для выполнения всех этих задач служит всего одна инструкция: SELECT.

Самый простой способ использования инструкции SELECT состоит в извлечении всех данных, хранящихся во всех строках конкретной таблицы. Для этого используется следующий синтаксис:

```
SELECT * FROM CUSTOMER ;
```

Звездочка (*) — это символ обобщения, подразумевающий все. В приведенном примере этот символ заменяет список имен всех столбцов таблицы CUSTOMER. В результате выполнения этой инструкции на экран выводятся все данные, находящиеся во всех строках и столбцах этой таблицы.

Инструкции SELECT могут быть намного сложнее, чем приведенная в данном примере. Некоторые из них могут быть настолько сложными, что в них очень трудно разобраться. Это связано с возможностью присоединения к базовой инструкции множества уточняющих предложений. Подробно об уточняющих предложениях вы узнаете в главе 10. В этой же главе мы вкратце

поговорим о предложении WHERE — самом распространенном средстве ограничения количества строк, возвращаемых инструкцией SELECT.

Инструкция SELECT с предложением WHERE имеет следующий общий вид.

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

Параметр *список_столбцов* определяет, какие столбцы таблицы следует отобрать для вывода, поскольку только они отобразятся на экране. В предложении FROM определяется имя той таблицы, столбцы которой требуется отобразить. А предложение WHERE исключает те строки, которые не удовлетворяют заданному в нем условию. Условие может быть простым (например, WHERE CUSTOMER_STATE = 'NH', отбирающее записи, относящиеся только к штату Нью-Гэмпшир) или составным (например, WHERE CUSTOMER_STATE = 'NH' AND STATUS = 'Active', включающее в себя комбинацию двух условий).

Следующий пример показывает, как выглядит составное условие в инструкции SELECT.

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

Эта инструкция возвращает имена, фамилии и телефонные номера всех активных клиентов, живущих в штате Нью-Гэмпшир. Ключевое слово AND означает, что для того, чтобы строка была возвращена, она должна соответствовать обоим условиям: State = 'NH' и Status = 'Active'.

ПРИМЕНЕНИЕ SQL В ИНТЕРАКТИВНЫХ ИНСТРУМЕНТАХ

Инструкция SELECT — не единственное средство извлечения информации из базы данных. В СУБД, как правило, для манипуляций данными имеются встроенные интерактивные средства. С помощью этих инструментов можно добавлять записи в базу, изменять и удалять их из нее, а также посылать запросы к базе.

Многие СУБД предоставляют пользователю выбор между интерактивными средствами и режимом SQL. В большинстве случаев предлагаемые инструменты не покрывают всех возможностей, предлагаемых SQL. Если доступные интерактивные средства вас не устраивают, вам придется вручную писать инструкции SQL. По этой причине ознакомиться с SQL просто необходимо, даже если вы предпочитаете использовать графический интерфейс. Этот язык позволит выполнить операции, слишком сложные для интерактивных инструментов, поэтому исключительно важно понять, как он работает и что можно сделать с его помощью.

Создание представлений

Структура базы данных, спроектированной в соответствии с разумными принципами, включая и соответствующую нормализацию (см. главу 5), обеспечивает максимальную целостность данных. Однако такая структура данных часто не позволяет обеспечить наилучший способ их просмотра. Одни и те же данные могут использоваться разными приложениями, и у каждого из них может быть своя специализация. Одной из самых востребованных возможностей SQL является вывод данных в виде представлений, структура которых отличается от структуры таблиц базы, в которой 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 (DDL). В этом разделе мы будем рассматривать представления в контексте извлечения данных и манипуляции ими.

Инструкция `SELECT` всегда возвращает результат в виде виртуальной таблицы. Представление же само является особой разновидностью виртуальных таблиц. Оно отличается от других виртуальных таблиц тем, что в метаданных базы данных хранится его определение. Это придает представлению те признаки постоянства, которыми не обладают другие виртуальные таблицы.


СОВЕТ

Работать с представлением можно так же, как и с настоящей таблицей базы данных. Различие состоит в том, что данные представления не являются физически независимой частью базы данных. Представление извлекает данные из одной или множества таблиц, на которых оно базируется. В каждом приложении могут быть свои, непохожие друг на друга представления, но созданные на основе одних и тех же данных.

Рассмотрим базу данных `VetLab`, описанную в главе 5. Она состоит из пяти таблиц: `CLIENT` (фирма-клиент), `TESTS` (анализы), `EMPLOYEE` (сотрудник), `ORDERS` (заказы) и `RESULTS` (результаты). Предположим, что менеджеру по маркетингу компании VetLab необходимо просмотреть, из каких штатов поступают заказы. Часть этой информации находится в таблице `CLIENT`, а часть — в таблице `ORDERS`. В то же время сотруднику службы контроля качества нужно сравнить дату оформления заказа одного из анализов и дату получения его окончательного результата. Для этого ему требуются данные из таблиц `ORDERS` и `RESULTS`. Для каждого из этих сотрудников можно создать отдельные представления, содержащие только нужные им данные.

Создание представлений из таблиц

Для менеджера по маркетингу можно создать представление, показанное на рис. 6.1.

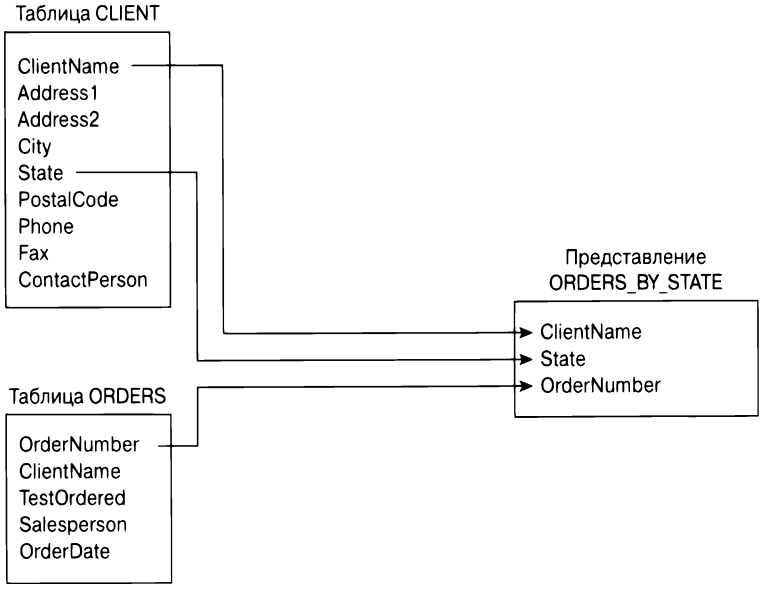


Рис. 6.1. Представление ORDERS_BY_STATE, предназначенное для менеджера по маркетингу

Это представление создается с помощью следующей инструкции.

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

Итак, в наше представление входят три столбца: `ClientName` (название фирмы-клиента), `State` (штат) и `OrderNumber` (номер заказа). Поле `ClientName` находится как в таблице `CLIENT`, так и в таблице `ORDERS` и используется для создания связи между ними. Представление получает информацию из столбца `State` таблицы `CLIENT` и берет для каждого заказа значение из столбца `OrderNumber` таблицы `ORDERS`. В приведенном примере имена столбцов объявляются явным образом.

Обратите внимание на то, что для поля `ClientName` указано имя содержащей его таблицы, а для полей `State` и `OrderNumber` — нет. Дело в том, что поле `State` имеется только в таблице `CLIENT`, а поле `OrderNumber` — только в таблице `ORDERS`, поэтому никакой двусмысленности не возникает. Но поле

ClientName входит в состав двух таблиц, CLIENT и ORDERS, поэтому необходим дополнительный (уточняющий) идентификатор.


ЗАПОМНИ!

Впрочем, если имена столбцов в представлении совпадают с именами столбцов исходных таблиц, то такое детальное объявление использовать необязательно. Пример, приведенный в следующем разделе, демонстрирует похожую инструкцию CREATE VIEW, в которой имена столбцов для представления не указываются явно, а только подразумеваются.

Создание представления с условием отбора

Как видно на рис. 6.2, для сотрудника службы контроля качества создано представление, отличающееся от того, которое использует менеджер по маркетингу.

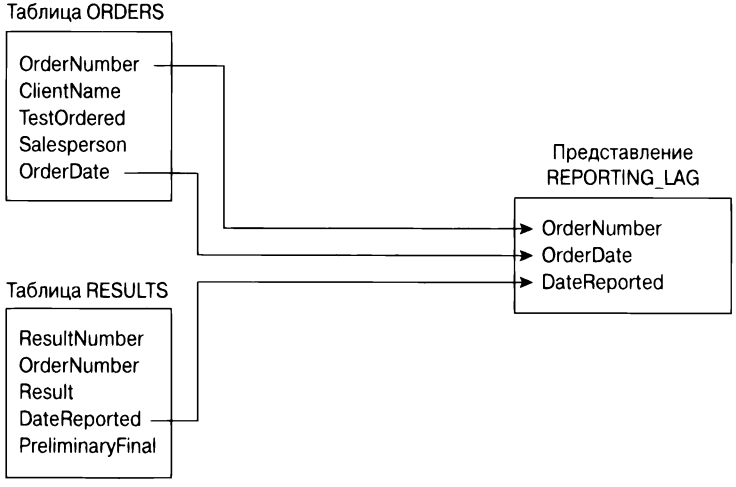


Рис. 6.2. Представление REPORTING_LAG, предназначенное для сотрудника службы контроля качества

Ниже приведен код, с помощью которого создано представление, показанное 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 приведенном представлении содержится информация из таблицы ORDERS о датах заказов и из таблицы RESULTS о датах получения результатов анализов.

В этом представлении отображаются только строки, у которых в столбце PreliminaryFinal, взятом из таблицы RESULTS, находится значение 'F' (что соответствует окончательному результату). Обратите внимание на то, что список явно задаваемых столбцов, включенный в определение представления ORDERS_BY_STATE, совершенно не обязателен. Представление REPORTING_LAG прекрасно работает и без такого списка.

Создание представления с модифицированным атрибутом

В примерах из двух предыдущих разделов предложения инструкции SELECT содержат только имена столбцов. Однако инструкция SELECT может включать и выражения. Предположим, владелец лаборатории VetLab в честь своего дня рождения хочет предоставить всем клиентам скидку 10%. Это можно реализовать путем создания представления BIRTHDAY (день рождения) на основе двух таблиц: ORDERS и TESTS.

```
CREATE VIEW BIRTHDAY
       (ClientName, Test, OrderDate, BirthdayCharge)
    AS SELECT ClientName, TestOrdered, OrderDate,
       StandardCharge *0.9
  FROM ORDERS, TESTS
 WHERE TestOrdered = TestName ;
```

Обратите внимание на то, 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 создается это представление, показано на рис. 6.3.

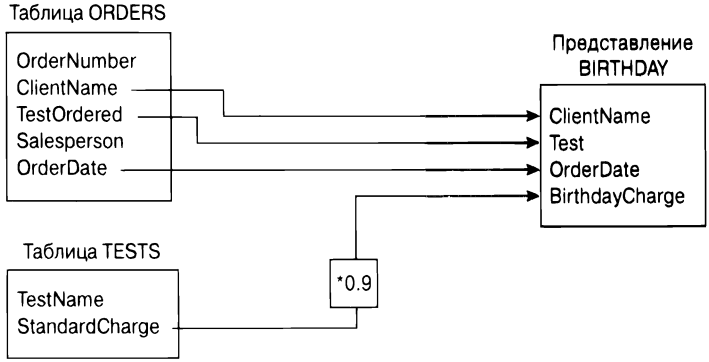


Рис. 6.3. Представление, созданное для демонстрации скидок в честь дня рождения

Представления можно создавать как на основе множества таблиц (мы видели это в предыдущих примерах), так и на основе всего лишь одной таблицы.

Если вам не нужны конкретные столбцы и строки какой-либо таблицы, создайте представление, в котором их нет, а затем работайте уже не с самой таблицей, а с представлением. Этот подход защищает пользователя от путаницы и не отвлекает внимание, что бывает при обилии не относящихся к делу данных на экране.


СОВЕТ

Еще одной причиной создания представлений является обеспечение безопасности исходных таблиц базы данных. Одни столбцы ваших таблиц можно открыть для просмотра, а другие, наоборот, скрыть. Таким образом, можно создать представление только с общедоступными столбцами, открыв к нему широкий доступ, и при этом ограничить доступ к исходным таблицам этого представления. В главе 14 будут рассмотрены вопросы защиты информации в базах данных, и вы узнаете, как управлять правами доступа пользователей к ресурсам.

Обновление представлений

Созданные таблицы автоматически поддерживают возможности вставки, обновления и удаления данных. А вот к представлениям этот тезис относится не всегда. Обновляя представление, вы в действительности обновляете исходную таблицу. Вот две потенциальные проблемы, возникающие при обновлении представлений.

- **Некоторые представления получают данные из двух и более таблиц.** Если вы обновляете такое представление, то его базовые таблицы не всегда обновляются корректно.
- **Инструкция `SELECT` представления может содержать выражения.** Так как выражения не соответствуют непосредственно полям таблицы, СУБД может не знать, как их обновлять.

Предположим, вы создаете представление, используя следующую инструкцию.

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

Не исключено, что вы допустили возможность обновить столбец `Pay` (оплата), используя такую инструкцию:

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

К сожалению, этот подход не сработает, потому что в таблице EMPLOYEE нет столбца Pay. Невозможно обновить в представлении то, чего нет в исходной таблице.

Собираясь обновлять представления, не забывайте следующее правило: столбец представления нельзя обновлять, если он не соответствует столбцу его базовой таблицы.

Добавление новых данных

В базе данных каждая таблица появляется на свет пустой, т.е. сразу после создания (с помощью SQL-инструкции или с помощью инструмента быстрой разработки) такая таблица является ничем иным, как структурной оболочкой, не содержащей данных. Чтобы таблица стала полезной, в нее необходимо поместить какие-нибудь данные. Эти данные могут уже храниться в компьютере в цифровом виде или вводиться пользователем вручную. Данные в таблицу можно ввести следующим образом.

- **Данные еще не преобразованы в цифровой формат.** Если данные еще не хранятся в цифровом виде, то кто-то должен ввести их построчно. Данные можно также вводить с помощью оптических сканеров и систем распознавания речи (правда, этот метод используется относительно редко).
- **Данные преобразованы в нужный цифровой формат.** Если данные уже находятся в цифровом виде, но, возможно, не в том формате, который используется в таблицах базы данных, то их нужно сначала преобразовать в соответствующий формат, а затем уже вставлять в базу данных.
- **Данные преобразованы в корректный цифровой формат.** Если данные уже находятся в нужном цифровом формате, то можно их скопировать в готовом виде в новую базу данных.

В следующем разделе вы узнаете о том, как добавить в таблицу данные, находящиеся в любой из трех названных форм. В зависимости от текущей формы данных их можно занести в базу одной операцией или пошагово, по одной записи за раз. Каждая вводимая запись соответствует одной строке таблицы базы данных.

Добавление данных в виде отдельных записей

В большинстве СУБД поддерживается ввод данных с помощью форм. Другими словами, вам предоставляется возможность создания экранной формы, в которой для каждого табличного столбца из базы данных будет существовать свое поле. По названию поля легко определить, какие данные следует вводить в него. Оператор базы данных вручную вводит в форму все данные, соответствующие одной строке. После того как СУБД примет заполненную строку, система очистит форму для ввода следующей. Таким образом, можно ввести в таблицу данные в виде отдельных строк, одну за другой.

Ввод данных с помощью форм прост и менее подвержен ошибкам, чем ввод списков значений, разделенных запятыми. Основная проблема ввода данных с помощью форм состоит в том, что сами формы не стандартизированы. В различных СУБД используются собственные методы создания форм. В то же время оператору, занятому вводом данных, совершенно безразлично, как именно создана форма ввода. Можно создать форму, которая будет выглядеть приблизительно одинаково в различных СУБД. (Для оператора это практически незаметно, но разработчику приложений придется учиться заново каждый раз, когда будут изменяться средства разработки.) В этом методе ввода данных существует еще одна проблема: в некоторых реализациях невозможно полностью проверить правильность вводимых данных.

Самый лучший способ поддержки высокого уровня целостности 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 входят в заданный диапазон. Естественно, применение ограничений не предотвратит все возможные ошибки, но позволит все же выявить некоторые из них.

СОВЕТ

Если средства разработки форм в вашей СУБД не позволяют применить полную проверку правильности ввода данных, гарантирующую их целостность, то вам, возможно, придется создать собственную экранную форму, использовать ее для ввода данных в переменные, а затем проверять корректность их значений с помощью программного кода приложения. И только убедившись, что все значения, принятые для записи строки таблицы, являются правильными, программа сможет добавить эту строку в таблицу с помощью SQL-инструкции `INSERT`.

Для ввода данных, предназначенных для одной строки таблицы, используется следующий синтаксис инструкции `INSERT`:

```
INSERT INTO таблица_1 [(столбец_1, столбец_2, ..., столбец_n)]
    VALUES (значение_1, значение_2, ..., значение_n) ;
```

Квадратные скобки ([]) означают, что заключенный в них 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, в том же порядке, в котором столбцы находятся в таблице, то эти элементы попадут в нужное место — неважно, указаны при этом названия столбцов явно или нет. Если же вы хотите расположить вводимые значения в порядке, который не совпадает с порядком следования столбцов в таблице, то последовательность имен столбцов необходимо указать явно, причем она должна совпадать с последовательностью значений, указанных в предложении VALUES.

Например, чтобы ввести запись в таблицу CUSTOMER, используйте следующий синтаксис.

```
INSERT INTO CUSTOMER (CustomerID, FirstName, LastName,
      Street, City, State, Zipcode, Phone)
    VALUES (:vcustid, 'Денис', 'Данилкин', 'Белорусская, 38',
      'Новосибирск', 'NS', '41156', '(046) 430-5555') ;
```

После ключевого слова VALUES первой в списке стоит vcustid — базовая переменная-счетчик, значение которой программа увеличивает на единицу после ввода в таблицу новой строки. Это гарантирует, что в столбце CustomerID не будет дублирования значений. CustomerID является первичным ключом таблицы, поэтому его значения должны быть уникальными. Остальные значения в списке VALUES являются не переменными, а элементами данных, хотя при желании данные для этих столбцов также можно поместить в переменные. Инструкция INSERT работает одинаково хорошо с аргументами ключевого слова VALUES, выраженными как в виде переменных, так и в виде конкретных значений.

Добавление данных только в выбранные столбцы

Иногда нужно где-то зафиксировать существование объекта, даже если по нему еще нет всех данных. Если для таких объектов у вас есть таблица базы данных, то строку по новому объекту можно вставить в нее, не заполняя значениями все столбцы этой строки. Чтобы таблица была в первой нормальной форме, обязательно нужно вставлять только те данные, которые позволяют отличить новую строку от всех остальных строк этой таблицы, в частности первичный ключ (о первой нормальной форме см. в главе 5). Кроме первичного ключа, можно вставить все данные, известные об этом объекте на текущий момент. В тех столбцах, в которые данные не вводятся, остаются пустые значения (NULL).

Ниже приведен пример частичного ввода строки.

```
INSERT INTO CUSTOMER (CustomerID, FirstName, LastName)
    VALUES (:vcustid, 'Денис', 'Данилкин') ;
```

При выполнении этой инструкции в таблицу базы данных вставляются только уникальный идентификатор клиента, а также его имя и фамилия. Остальные столбцы этой строки будут содержать значения NULL.

Добавление группы строк в таблицу

Добавлять в таблицу строки одну за другой с помощью инструкции INSERT — довольно утомительное занятие, и даже при использовании эргономичной экранной формы вы быстро устанете. Если же у вас есть надежное средство автоматического ввода данных, вы будете стараться по возможности использовать именно его.

Как правило, автоматический ввод используется тогда, когда данные уже существуют в электронном виде, т.е. кто-то предварительно ввел их в компьютер. Зачем же повторять эту рутинную работу? Перенести данные из одного файла в другой можно при минимальном участии человека. Если вам известны характеристики исходных данных и желаемая структура итоговой таблицы, то компьютер может (в принципе) выполнить такой перенос данных автоматически.

Копирование из внешнего файла

Предположим, 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 отличной от вашей СУБД. Данные могут быть представлены в коде ASCII, EBCDIC или в каком-нибудь другом закрытом внутреннем формате. Так что же делать?


СОВЕТ

Хорошо, если формат этих данных окажется одним из широко используемых. Если повезет, у вас будет хороший шанс раздобыть утилиту преобразования этого формата в приемлемый для вашей среды разработки. Если *сильно* повезет, то для преобразования текущего формата данных будет достаточно встроенных средств среды разработки. Пожалуй, самыми распространенными на персональных компьютерах форматами являются Access, SQL Server и MySQL. Если нужные вам данные находятся в одном из этих форматов, то преобразование должно пройти легко. Но если формат данных окажется не таким распространенным, то, скорее всего, преобразование придется провести в два этапа.

Перенос всех строк из одной таблицы в другую

Намного проще (чем импортировать внешние данные) извлечь содержимое одной таблицы вашей базы данных и объединить его с содержимым другой таблицы. Если структуры первой и второй таблиц идентичны (т.е. для каждого столбца первой таблицы существует соответствующий столбец во второй, а их типы данных совпадают), то задача решается очень просто. В этом случае содержимое двух таблиц можно объединить с помощью реляционного оператора `UNION`. В результате получится виртуальная таблица, содержащая данные из обеих исходных таблиц. О реляционных операторах, в том числе о `UNION`, вы узнаете из главы 11.

Перенос выбранных столбцов и строк из одной таблицы в другую

Нередко структура данных исходной таблицы не соответствует в точности структуре той таблицы, в которую вы собираетесь их поместить. Вполне вероятно совпадение некоторых столбцов — и это могут оказаться как раз те столбцы, которые нужно перенести. Комбинируя инструкции `SELECT` с помощью оператора `UNION`, можно указать, какие именно столбцы исходных таблиц должны войти в итоговую виртуальную таблицу. Включая в инструкции `SELECT` предложения `WHERE`, можно помещать в виртуальную таблицу только те строки, которые удовлетворяют заданным условиям. Предложения `WHERE` подробно описываются в главе 10.

Предположим, есть две таблицы, `PROSPECT` (потенциальный клиент) и `CUSTOMER` (покупатель), и нужно составить список всех жителей штата Мэн, данные о которых находятся в обеих таблицах. В этом случае можно получить виртуальную таблицу с нужной информацией, используя следующий запрос.

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

Рассмотрим этот код.

- Инструкции `SELECT` сообщают о том, что у созданной таблицы будут столбцы `FirstName` (имя) и `LastName` (фамилия).
- Предложения `WHERE` ограничивают состав строк в создаваемой таблице, отбирая лишь те, в столбце `State` (штат) которых содержится значение `'ME'` (штат Мэн).

- Столбца State в виртуальной таблице не будет, несмотря на то что он существует в двух исходных таблицах: PROSPECT и CUSTOMER.
- Оператор UNION объединяет результаты выполнения первого запроса SELECT к таблице PROSPECT с результатами выполнения второго запроса SELECT к таблице CUSTOMER, удаляет все строки-дубликаты, а затем выводит окончательный результат на экран.


СОВЕТ

Еще один способ копирования информации из одной таблицы в другую состоит во вложении инструкции SELECT в инструкцию INSERT. Этот метод (о подзапросах SELECT будет подробнее говориться в главе 12) не создает виртуальную таблицу, а просто дублирует отобранные данные. Например, можно извлечь все строки из таблицы CUSTOMER и вставить их в таблицу PROSPECT. Конечно, эта операция будет успешной только в том случае, если обе таблицы имеют одинаковую структуру. Но для отбора только тех покупателей, которые живут в штате Мэн, достаточно простой инструкции SELECT, имеющей в предложении WHERE всего лишь одно условие. Соответствующий код показан ниже.

```
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, PROSPECT и CUSTOMER), увеличивается скорость извлечения данных. Чтобы избежать избыточности и поддерживать согласованность данных, не вставляйте, не изменяйте и не удаляйте строки в одной таблице без вставки, изменения и удаления соответствующих строк в другой. Еще одна проблема состоит в возможности генерирования инструкцией INSERT дубликатов первичных ключей. Даже если существует единственный потенциальный клиент, первичный ключ (ProspectID) которого совпадает с первичным ключом (CustomerID) зарегистрированного покупателя, данные о котором вы пытаетесь вставить в таблицу PROSPECT, эта операция вставки не будет выполнена. Если обе таблицы имеют автоинкрементные первичные ключи, вы не должны устанавливать для счетчиков автоинкрементации в обеих таблицах одни и те же начальные значения. Позаботьтесь о том, чтобы обе группы порядковых номеров не пересекались.

Обновление существующих данных

Все течет, все меняется. Если вам не нравится нынешнее положение дел, то нужно просто немного подождать — через некоторое время все может измениться к лучшему. По мере изменений, происходящих в жизни, обычно приходится обновлять и базы данных, поскольку они, как правило, отражают наш изменчивый мир в тех или иных аспектах. Например, ваш клиент может поменять адрес. Количество товаров на складе меняется ежеминутно (будем надеяться, не в результате воровства, а потому, что товар хорошо продается). Это типичные примеры тех событий, из-за которых приходится постоянно обновлять базы данных.

В SQL для изменения данных, хранящихся в таблице, предусмотрена инструкция `UPDATE`. С помощью одной такой инструкции можно изменить в таблице одну либо несколько строк или даже все ее строки. В инструкции `UPDATE` используется следующий синтаксис.

```
UPDATE имя_таблицы
   SET столбец_1 = выражение_1, столбец_2 = выражение_2,
       ..., столбец_n = выражение_n
[WHERE предикаты] ;
```


СОВЕТ

Предложение `WHERE` не является обязательным. Оно указывает, какие строки должны обновляться. Если не использовать это предложение, то будут обновляться все строки таблицы. Предложение `SET` определяет новые значения изменяемых столбцов.

Рассмотрим таблицу `CUSTOMER` (покупатель), представленную в табл. 6.1 и содержащую столбцы `Name` (имя и фамилия), `City` (город), `AreaCode` (телефонный код региона) и `Telephone` (номер телефона).

Таблица 6.1. Таблица CUSTOMER

Name	City	AreaCode	Telephone
Денис Данилов	Мытищи	(495)	555-1111
Геннадий Гончаров	Люберцы	(495)	555-2222
Юрий Юрченко	Зеленоград	(495)	555-3333
Карина Новикова	Зеленоград	(495)	555-4444
Дина Бочкарева	Зеленоград	(495)	555-5555

Время от времени списки покупателей изменяются, по мере того как эти люди переезжают, изменяются их номера телефонов и т.п. Предположим, покупатель Юрий Юрченко переехал из Зеленограда в Мытищи. Тогда запись этого покупателя, находящуюся в таблице CUSTOMER, можно обновить с помощью следующей инструкции UPDATE.

```
UPDATE CUSTOMER
   SET City = 'Мытищи', Telephone = '777-7777'
 WHERE Name = 'Юрий Юрченко' ;
```

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

Таблица 6.2. Таблица CUSTOMER после обновления одной строки инструкцией UPDATE

Name	City	AreaCode	Telephone
Денис Данилов	Мытищи	(495)	555-1111
Геннадий Гончаров	Люберцы	(495)	555-2222
Юрий Юрченко	Мытищи	(495)	777-7777
Карина Новикова	Зеленоград	(495)	555-4444
Дина Бочкарева	Зеленоград	(495)	555-5555

Подобную инструкцию можно использовать для обновления сразу нескольких строк. Предположим, Зеленоград переживает резкий рост населения и ему теперь присвоен собственный телефонный код. Все строки покупателей, проживающих в этом городе, можно изменить с помощью одной инструкции UPDATE.

```
UPDATE CUSTOMER
   SET AreaCode = '(595)'
 WHERE City = 'Зеленоград' ;
```

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

Обновить в таблице все строки гораздо легче, чем только некоторые из них, ведь в таком случае не нужно использовать ограничивающее предложение WHERE. Предположим, что Москва значительно увеличилась в размерах и в ее состав вошли все населенные пункты, упомянутые в базе данных. Тогда все строки можно сразу изменить с помощью одной инструкции.

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

Таблица 6.3. Таблица CUSTOMER после обновления нескольких строк инструкцией UPDATE

Name	City	AreaCode	Telephone
Денис Данилов	Мытищи	(495)	555-1111
Геннадий Гончаров	Люберцы	(495)	555-2222
Юрий Юрченко	Мытищи	(495)	777-7777
Карина Новикова	Зеленоград	(595)	555-4444
Дина Бочкарева	Зеленоград	(595)	555-5555

Результат выполнения этой инструкции показан в табл. 6.4.

Таблица 6.4. Таблица CUSTOMER после обновления всех строк инструкцией UPDATE

Name	City	AreaCode	Telephone
Денис Данилов	Москва	(495)	555-1111
Геннадий Гончаров	Москва	(495)	555-2222
Юрий Юрченко	Москва	(495)	777-7777
Карина Новикова	Москва	(595)	555-4444
Дина Бочкарева	Москва	(595)	555-5555

В предложении `WHERE`, используемом для ограничения строк, к которым применяется инструкция `UPDATE`, может находиться подзапрос `SELECT`, результат которого используется в качестве входных данных для другой инструкции `SELECT`.

Предположим, вы оптовый продавец и ваша база данных включает таблицу `VENDOR` с названиями всех фирм-производителей, у которых вы покупаете товары. У вас также есть таблица `PRODUCT` с названиями всех продаваемых вами товаров и ценами, которые вы на них устанавливаете. В таблице `VENDOR` имеются столбцы `VendorID` (идентификатор поставщика), `VendorName` (название поставщика), `Street` (улица), `City` (город), `State` (штат) и `Zip` (почтовый индекс). А таблица `PRODUCT` содержит столбцы `ProductID` (идентификатор товара), `ProductName` (название товара), `VendorID` (идентификатор поставщика) и `SalePrice` (цена продажи).

Предположим, ваш поставщик, компания "Мегасервис Корпорейшн", приняла решение поднять цены на все виды товаров на 10%. И для того, чтобы сохранить маржу, вам также придется поднять на 10% цены товаров, получаемых от этого поставщика. Это можно сделать с помощью следующей инструкции UPDATE.

```
UPDATE PRODUCT
   SET SalePrice = (SalePrice * 1.1)
 WHERE VendorID IN
       (SELECT VendorID FROM VENDOR
         WHERE VendorName = 'Мегасервис Корпорейшн') ;
```

Подзапрос SELECT отбирает из столбца VendorID идентификатор, соответствующий компании "Мегасервис Корпорейшн". Затем полученное значение можно использовать для поиска в таблице PRODUCT тех строк, которые следует обновить. В результате цены всех товаров, полученных от компании "Мегасервис Корпорейшн", будут повышены на 10%, а цены остальных останутся прежними. О подзапросах SELECT мы подробно поговорим в главе 12.

Перенос данных

Чтобы добавить данные в таблицу или представление, помимо инструкций INSERT и UPDATE, можно использовать инструкцию MERGE. Инструкция MERGE позволяет объединить данные исходных таблиц или представлений с данными других таблиц или представлений. Эта же инструкция позволяет вставить новые строки в нужную таблицу или обновить существующие в ней. Таким образом, инструкция MERGE представляет собой весьма удобное средство копирования уже существующих данных из одного места в другое.

Возьмем в качестве примера базу данных VetLab, описанную в главе 5. Предположим, часть сотрудников, данные о которых занесены в таблицу EMPLOYEE, — это продавцы, которые уже приняли заказы. Другая часть — это сотрудники, не связанные напрямую с продажами, или продавцы, которые еще не имеют заказов. Только что закончившийся год был прибыльным, поэтому компания решила выдать премии по 100 долларов каждому, кто принял по крайней мере один заказ, и по 50 долларов всем остальным. Для начала создадим таблицу BONUS (бонус) и вставим в нее записи всех сотрудников, которые фигурируют хотя бы в одной строке таблицы ORDERS, устанавливая в каждой записи значение премии равным 100 долларам.

Затем воспользуемся инструкцией MERGE для вставки новых записей о тех сотрудниках, которые не имеют заказов, чтобы выдать им премию по

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 выполним запрос к таблице BONUS и посмотрим, что она содержит.

```
SELECT * FROM BONUS ;

EmployeeName                Bonus
---------------  ---------------
Алена Булина                  100
Диана Жень                    100
Николай Кукин                 100
Олег Донченко                 100
```

Затем выполним инструкцию MERGE, чтобы назначить премию в 50 долларов всем остальным сотрудникам.

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

Записи о сотрудниках из таблицы EMPLOYEE, которые не совпадают с записями в таблице BONUS, будут вставлены в последнюю таблицу. Теперь запрос к таблице BONUS даст следующий результат.

```
SELECT * FROM BONUS ;

EmployeeName                Bonus
-----------------  --------------
Алена Булина                  100
Диана Жень                    100
Николай Кукин                 100
Олег Донченко                 100
Наталия Гончарова              50
Михаил Бардин                  50
Олег Яненко                    50
Леонид Лоевский                50
```

Первые четыре записи, созданные с помощью инструкции INSERT, располагаются в алфавитном порядке по именам сотрудников. Остальные записи, добавленные с помощью инструкции MERGE, располагаются в том порядке, в котором они находились в таблице EMPLOYEE.

Инструкция MERGE — относительно новое добавление в SQL и может пока не поддерживаться некоторыми СУБД. В стандарт SQL:2011 добавлена еще одна возможность инструкции MERGE, которая может показаться где-то парадоксальной, поскольку она позволяет удалять записи.

Предположим, после выполнения инструкции INSERT вы решили совсем не премировать тех, кто принял хотя бы один заказ, но при этом выдать премию в 50 долларов всем остальным сотрудникам. Для реализации этого решения можно выполнить следующую инструкцию MERGE, которая позволит удалить бонусные записи о продавцах и добавить бонусные записи о тех, кто не связан с продажами.

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

Вот как выглядит результат выполнения этого кода.

```
SELECT * FROM BONUS;

EmployeeName                Bonus
-----------------     -----------
Наталия Гончарова              50
Михаил Бардин                  50
Олег Яненко                    50
Леонид Лоевский                50
```

Удаление устаревших данных

Со временем данные могут устаревать и становиться бесполезными. Ненужные данные, находясь в таблице, только замедляют работу системы, занимают память и путают пользователей. Наилучшим решением является перенос старых данных в архивную таблицу, а затем извлечение ее из базы данных. А если вдруг вам потребуется снова взглянуть на эти данные, то их можно будет восстановить. При этом они уже не будут замедлять ежедневную обработку данных. Впрочем, независимо от того, будете вы архивировать устаревшие

данные или нет, все же наступит время, когда их придется удалить. Для удаления строк из таблицы, находящейся в базе данных, в SQL предусмотрена инструкция DELETE.

С помощью всего одной инструкции DELETE можно удалить как все строки таблицы, так и некоторые из них. Строки, предназначенные для удаления, можно отобрать, используя в инструкции DELETE необязательное предложение WHERE. Синтаксис инструкции DELETE почти такой же, как у инструкции SELECT, за исключением того, что не нужно указывать столбцы. Ведь при удалении строки удаляются данные, находящиеся во всех ее столбцах.

Предположим, например, что ваш клиент Михаил Кадилов переехал в Швейцарию и больше ничего у вас покупать не собирается. Вы можете удалить его данные из таблицы CUSTOMER, используя следующую инструкцию.

```
DELETE FROM CUSTOMER
 WHERE FirstName = 'Михаил' AND LastName = 'Кадилов' ;
```

Если у вас только один покупатель, которого зовут Михаил Кадилов, то эта инструкция будет выполнена безупречно. Однако существует вероятность, что Михаилом Кадиловым зовут как минимум двух ваших покупателей. Чтобы удалить данные именно того из них, к кому вы потеряли интерес, добавьте в предложение WHERE дополнительные условия (указав, например, такие столбцы, как Street, Phone или CustomerID). Если не добавить предложение WHERE, то будут удалены все записи, относящиеся ко всем, кого зовут Михаилом Кадиловым.

Глава 7

Обработка темпоральных данных

В ЭТОЙ ГЛАВЕ...

- Определение моментов и промежутков времени
- Отслеживание событий
- Аудит изменений
- Управление событиями и их регистрацией
- Форматирование и анализ значений даты и времени

До появления SQL:2011 в стандарте ISO/IEC SQL не было механизма работы с данными, которые в один момент времени действительны, а в другой — нет. Без такого механизма не обойтись приложениям, которые должны вести журнал событий. Это означает, что обеспечивать регистрацию всего происходящего в базе данных в некоторый заданный момент времени должен был разработчик приложения, а не база данных. В результате такие приложения получались сложными, дорогостоящими и содержали ошибки, от которых нелегко избавиться.

В стандарт SQL:2011 был добавлен новый синтаксис, который позволяет выполнять обработку темпоральных (или временны*х*, т.е. связанных со временем) данных и при этом не влияет на код обработки данных, не имеющих отношения ко времени. Это хорошая новость для тех, кто хочет усилить новыми возможностями уже работающее с базой данных SQL-приложение. Но несмотря на то что с появлением SQL:2011 была стандартизирована обработка темпоральных данных, не был определен порядок форматирования или анализа шаблонов данных о времени. Этот недостаток был исправлен в SQL:2016.

Что следует понимать под *темпоральными данными*? В стандарте SQL:2011 такой термин не используется, но он широко распространен в среде разработчиков баз данных. В стандарте SQL:2011 под темпоральными понимаются данные, связанные с одним или несколькими промежутками времени, в течение которых эти данные считаются действительными. Проще говоря, это означает, что с помощью средств обработки темпоральных данных можно определять, когда тот или иной элемент данных является допустимым.

В этой главе будет введено понятие периода времени. Мы рассмотрим различные способы представления времени и узнаем, какое влияние оказывают темпоральные данные на определение первичных ключей и ограничений ссылочной целостности. Наконец, вы узнаете о том, как сохранять и обрабатывать очень сложные данные в битемпоральных таблицах.

Моменты и периоды времени

Несмотря на то что версии стандарта SQL, выпущенные до SQL:2011, включали такие типы данных, как `DATE`, `TIME`, `TIMESTAMP` и `INTERVAL`, они не использовали понятие *периода времени* с определенными начальным и конечным моментами. Коль возникла такая потребность, то, казалось бы, имеет смысл определить новый тип данных `PERIOD`. Однако в стандарте SQL:2011 такой тип данных отсутствует. Ввод нового типа данных в SQL на довольно позднем этапе разработки этого языка означал бы нанесение серьезного ущерба "экосистеме", которая сформировалась вокруг SQL. Для добавления нового типа данных потребовалось бы серьезное вмешательство практически во все существующие продукты, работающие с базами данных.

Вместо введения типа данных `PERIOD` в стандарте SQL:2011 эта проблема была решена путем добавления *определений периода* в виде метаданных для таблиц. Определение периода представляет собой именованный компонент таблицы, идентифицирующий пару столбцов, которые содержат моменты времени начала и конца периода. Для создания или удаления периодов, созданных с использованием этих новых определений, были обновлены (с учетом нового синтаксиса) инструкции `CREATE TABLE` и `ALTER TABLE`, с помощью которых создаются и модифицируются таблицы.

Таким образом, период определяется двумя столбцами (начало и конец). Эти столбцы не отличаются от обычных столбцов, в которых хранятся данные существующих типов, и каждому из них присвоено собственное уникальное имя. Как упоминалось выше, определение периода — это именованный компонент таблицы. Он находится в том же пространстве имен, что и имена

обычных столбцов, и поэтому его имя не должно совпадать ни с одним из существующих имен столбцов.

SQL придерживается закрыто-открытой модели для периодов, т.е. период включает начальное время, но не включает конечное. Для любой строки в таблице конечное время периода должно быть больше начального. Любая СУБД должна учитывать это ограничение.

ЗАПОМНИ!

Существуют два измерения времени, которые важно учитывать при работе с темпоральными данными:

- действительное время — это период времени, в течение которого строка в таблице базы данных корректно отражает реальность;
- транзакционное время — это период времени, в течение которого строка находится в базе данных.

Значения действительного и транзакционного времени для строки в таблице не обязательно должны совпадать. Например, в рабочей базе данных, фиксирующей периоды, в течение которых контракты действительны, информация о контракте может быть (и наверняка будет) внесена до того, как контракт вступит в силу.

Стандарт SQL:2011 для работы с двумя различными измерениями времени позволяет создавать как отдельные таблицы, так и одну битемпоральную таблицу. Информация о времени выполнения транзакций хранится в системно-версионных таблицах, которые содержат период системного времени, обозначенный ключевым словом `SYSTEM_TIME`. Информация о действительном времени находится в таблицах, которые содержат период прикладного времени, причем этому периоду можно присвоить любое имя (при условии, что оно нигде больше не используется). Стандарт позволяет определить только один период системного времени и один период прикладного времени.

Несмотря на то что поддержка темпоральных данных в SQL впервые нашла свое отражение лишь в стандарте SQL:2011, темпоральные данные использовались задолго до того, как эти новые конструкции стандарта SQL:2011 были включены в какие-либо СУБД. Обычно для этого в таблице определялись два столбца: один — для начального момента времени, другой — для конечного. Тот факт, что стандарт SQL:2011 не определяет новый тип данных `PERIOD` и вместо этого использует определения периода в виде метаданных, означает, что существующие таблицы с такими двумя столбцами (начала и конца) можно легко модернизировать в соответствии с новыми синтаксическими возможностями. Программную логику, которая раньше обеспечивала обработку данных о периодах, теперь можно легко удалить из уже существующих приложений, тем самым упростив их, ускорив их работу и сделав их более надежными.

Таблицы с периодами прикладного времени

Рассмотрим пример. Предположим, компания хочет отслеживать, к какому отделу относятся ее служащие в любой момент времени на протяжении их периода работы. Компания может сделать это путем создания таблиц с периодами прикладного времени для служащих и отделов.

```
CREATE TABLE employee_atpt(
   EmpID      INTEGER,
   EmpStart   DATE,
   EmpEnd     DATE,
   EmpDept    VARCHAR(30),
   PERIOD FOR EmpPeriod (EmpStart, EmpEnd)
) ;
```

Начальное значение даты/времени (`EmpStart` в данном примере) включается в период, а конечное (`EmpEnd`) — нет. Такая ситуация подпадает под определение закрыто-открытой семантики.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Я еще не назначил первичный ключ, поскольку использование темпоральных данных несколько усложняет задачу определения первичных ключей, но мы вернемся к этому чуть позже.

Пока же заполним нашу таблицу данными и посмотрим, как она будет выглядеть.

```
INSERT INTO employee_atpt
     VALUES (12345, DATE '2018-01-01', DATE '9999-12-31', 'Sales');
```

В результате получаем таблицу с одной строкой (табл. 7.1).

Таблица 7.1. Таблица с периодом прикладного времени, содержащая одну строку

EmpID	EmpStart	EmpEnd	EmpDept
12345	2018-01-01	9999-12-31	Sales

"Странная" конечная дата 9999-12-31 означает, что пребывание этого служащего в компании еще не завершено, т.е. он продолжает работать на компанию. Для простоты в этом и следующих примерах я отбросил часы, минуты, секунды и дробную часть секунд.

Теперь предположим, что 15 марта 2018 года служащий 12345 временно (до 15 июля 2018 года) переводится в инженерно-технический отдел, после чего

он должен вернуться в свой отдел сбыта. Эти события можно отразить в базе данных с помощью следующей инструкции UPDATE.

```
UPDATE employee_atpt
   FOR PORTION OF EmpPeriod
      FROM DATE '2018-03-15'
      TO DATE '2018-07-15'
   SET EmpDept = 'Engineering'
   WHERE EmpID = 12345;
```

После выполнения инструкции UPDATE наша таблица будет содержать три строки, как показано в табл. 7.2.

Таблица 7.2. Таблица с периодами прикладного времени после выполнения инструкции UPDATE

EmpID	EmpStart	EmpEnd	EmpDept
12345	2018-01-01	2018-03-15	Sales
12345	2018-03-15	2018-07-15	Engineering
12345	2018-07-15	9999-12-31	Sales

Если служащий 12345 все еще работает в отделе сбыта (Sales), таблица правильно отражает его принадлежность к этому отделу с первого дня 2018 года по настоящее время.

Если у вас есть право вставки новых записей в таблицу и обновления существующих в ней данных, то вам также стоит иметь право на удаление данных из этой таблицы. Однако удаление данных из таблицы с периодами прикладного времени происходит несколько сложнее, чем удаление строк из обычной (нетемпоральной) таблицы. В качестве примера предположим, что служащий 12345 вместо перехода в инженерно-технический отдел 15 марта 2018 года увольняется из компании и возвращается обратно 15 июля того же года. В исходном виде таблица будет содержать только одну строку, как показано в табл. 7.3.

Таблица 7.3. Таблица с периодами прикладного времени до обновления или удаления

EmpID	EmpStart	EmpEnd	EmpDept
12345	2018-01-01	9999-12-31	Sales

При выполнении инструкции DELETE наша таблица будет обновлена, чтобы отразить период, в течение которого служащий 12345 уволился.

```
DELETE employee_atpt
   FOR PORTION OF EmpPeriod
      FROM DATE '2018-03-15'
      TO DATE '2018-07-15'
   WHERE EmpID = 12345;
```

Результат выполнения этого кода показан в табл. 7.4.

Таблица 7.4. Таблица с периодами прикладного времени после удаления

EmpID	EmpStart	EmpEnd	EmpDept
12345	2018-01-01	2018-03-15	Sales
12345	2018-07-15	9999-12-31	Sales

Сейчас таблица отражает периоды времени, в течение которых служащий 12345 работал на компанию (и здесь четко виден перерыв в работе).

Возможно, вы заметили одну странность таблиц, приведенных в этом разделе. В обычной (нетемпоральной) таблице, содержащей список работников предприятия, идентификатор сотрудника (EmpID) вполне может служить в качестве первичного ключа таблицы, поскольку он уникальным образом определяет каждого сотрудника. Но таблица с периодами прикладного времени может содержать несколько записей для одного сотрудника. Идентификатор сотрудника сам по себе больше не годится на роль первичного ключа таблицы.

Назначение первичных ключей в таблицах с периодами прикладного времени

После изучения табл. 7.2 и 7.4 становится ясно, что идентификатор сотрудника (EmpID) не гарантирует уникальность записей, поскольку таблица содержит сразу несколько записей с одним и тем же значением EmpID. Чтобы обеспечить отсутствие строк-дубликатов, в первичный ключ должны быть добавлены даты начала (EmpStart) и конца (EmpEnd). Однако просто добавить эти поля в ключ еще недостаточно. Рассмотрим табл. 7.5, отражающую ситуацию, когда служащий 12345 просто перешел в инженерно-технический отдел на несколько месяцев, а затем вернулся на прежнее место работы.

Да, включение столбцов EmpStart и EmpEnd в первичный ключ гарантирует уникальность этих двух строк таблицы, но обратите внимание на перекрытие используемых в них периодов времени. Судя по приведенной в табл. 7.5

Таблица 7.5. Ситуация, которой хотелось бы избежать

EmpID	EmpStart	EmpEnd	EmpDept
12345	2018-01-01	9999-12-31	Sales
12345	2018-03-15	2018-07-15	Engineering

информации, с 15 марта по 15 июля 2018 года служащий 12345 является сотрудником отдела сбыта и инженерно-технического отдела одновременно. В некоторых организациях такая ситуация, может быть, и допустима, но она значительно усложняет разработку и может привести к нарушению целостности данных. Поэтому примем ограничение, с которым, вероятно, согласятся в большинстве организаций: каждый служащий одновременно может работать только в одном отделе. Такое ограничение можно дополнительно применить к таблице с помощью инструкции ALTER TABLE.

```
ALTER TABLE employee_atpt
ADD PRIMARY KEY (EmpID, EmpPeriod WITHOUT OVERLAPS);
```

Сначала создать таблицу, а затем (т.е. позже) добавить ограничение, связанное с построением первичного ключа, — не очень хороший стиль проектирования таблиц. Лучше сразу включить такое ограничение в исходную инструкцию CREATE. Вот пример такого кода.

```
CREATE TABLE employee_atpt
   EmpID INTEGER NOT NULL,
   EmpStart DATE NOT NULL,
   EmpEnd DATE NOT NULL,
   EmpDept VARCHAR(30),
      PERIOD FOR EmpPeriod (EmpStart, EmpEnd)
      PRIMARY KEY (EmpID, EmpPeriod WITHOUT OVERLAPS)
) ;
```

Теперь строки с перекрытиями запрещены. Кроме того, для всех элементов первичного ключа я добавил ограничения NOT NULL. Неопределенное значение в любом из этих полей могло бы стать источником ошибок в будущем. Обычно СУБД предупреждает появление подобных неприятностей, но... береженого Бог бережет, не так ли?

Применение ограничений ссылочной целостности к таблицам с периодами прикладного времени

Любая база данных, предназначенная для поддержки серьезной информационной системы (а не просто списка элементов), вероятно, включает не одну

таблицу. Если база данных содержит несколько таблиц, то необходимо определить отношения между ними и ввести ограничения для поддержки ссылочной целостности.

В примере, который рассматривается в этой главе, мы используем таблицы с периодами прикладного времени с данными о служащих и об отделах компании. Между таблицей отделов и таблицей сотрудников существует отношение "один ко многим", поскольку в одном отделе может работать множество сотрудников, но каждый сотрудник принадлежит к одному и только одному отделу. Это означает, что вам необходимо поместить внешний ключ в таблицу сотрудников, который будет ссылаться на первичный ключ таблицы отделов. С учетом этого снова создадим таблицу сотрудников, но на этот раз с использованием более полной инструкции CREATE, а затем аналогичным образом создадим таблицу отделов.

```
CREATE TABLE employee_atpt (
   EmpID       INTEGER          NOT NULL,
   EmpStart    DATE             NOT NULL,
   EmpEnd      DATE             NOT NULL,
   EmpName     VARACHAR (30),
   EmpDept     VARCHAR (30),
   PERIOD FOR EmpPeriod (EmpStart, EmpEnd)
   PRIMARY KEY (EmpID, EmpPeriod WITHOUT OVERLAPS)
   FOREIGN KEY (EmpDept, PERIOD EmpPeriod)
      REFERENCES dept_atpt (DeptID, PERIOD DeptPeriod)
) ;
CREATE TABLE dept_atpt (
   DeptID      VARCHAR (30)     NOT NULL,
   Manager     VARCHAR (40)     NOT NULL,
   DeptStart   DATE             NOT NULL,
   DeptEnd     DATE             NOT NULL,
   PERIOD FOR DeptTime (DeptStart, DeptEnd),
   PRIMARY KEY (DeptID, DeptTime WITHOUT OVERLAPS)
) ;
```

Создание запросов к таблицам с периодами прикладного времени

Теперь можно извлекать из нашей базы данных подробную информацию с помощью инструкций SELECT, в которых используются темпоральные данные.

Например, мы можем запросить список всех служащих, которые на данный момент являются сотрудниками организации. Даже до выхода стандарта SQL:2011 вы могли бы сделать это с помощью такой инструкции.

```
SELECT *
  FROM employee_atpt
```

```
 WHERE EmpStart <= CURRENT_DATE()
   AND EmpEnd > CURRENT_DATE();
```

Но с вводом нового синтаксиса тот же результат можно получить более простым способом.

```
SELECT *
  FROM employee_atpt
 WHERE EmpPeriod CONTAINS CURRENT_DATE();
```

Кроме того, теперь совсем нетрудно запросить список служащих, принятых на работу в течение заданного периода.

```
SELECT *
  FROM employee_atpt
 WHERE EmpPeriod OVERLAPS
       PERIOD (DATE ('2018-01-01'), DATE ('2018-09-16'));
```

Помимо `CONTAINS` и `OVERLAPS`, в этом контексте можно использовать такие предикаты, как `EQUALS`, `PRECEDES`, `SUCCEEDS`, `IMMEDIATELY PRECEDES` и `IMMEDIATELY SUCCEEDS`.

Эти предикаты имеют следующие значения.

- Если один период равен (`EQUALS`) другому, то они в точности совпадают.
- Если один период предшествует (`PRECEDES`) другому, то это значит, что первый заканчивается до начала второго.
- Если один период следует (`SUCCEEDS`) за другим, то это значит, что первый начинается после завершения второго.
- Если один период непосредственно предшествует (`IMMEDIATELY PRECEDES`) другому, то это значит, что первый заканчивается прямо перед началом второго и является смежным с ним.
- Если один период непосредственно следует (`IMMEDIATELY SUCCEEDS`) за другим, то это значит, что первый начинается сразу после завершения второго и является смежным с ним.

Работа с системно-версионными таблицами

Назначение системно-версионных таблиц отличается от назначения таблиц с периодами прикладного времени, и, соответственно, таблицы двух видов функционируют по-разному. Таблицы с периодами прикладного времени позволяют определять периоды времени и служат для хранения данных, которые относятся к этим периодам. В отличие от них, системно-версионные таблицы предназначены для создания записей с точными данными о том, когда

некоторый элемент данных был добавлен в базу данных, изменен или удален из нее. Например, для банка важно знать, когда депозит был внесен или снят, причем эта информация должна сохраняться в банке в течение периода времени, оговоренного законом. Аналогично биржевым маклерам необходимо отслеживать, когда в точности была совершена конкретная сделка купли-продажи. Существует множество подобных ситуаций, в которых очень важно знать, когда точно (вплоть до долей секунды) произошло определенное событие.

К приложениям, предназначенным для обслуживания банковских или биржевых процессов, предъявляются строгие требования.

- Любая операция обновления или удаления должна сохранять исходное состояние строки, в котором она пребывала до выполнения этой операции.
- Система, а не пользователь, поддерживает начальное и конечное значения времени для всех периодов в строке.

  Исходные строки, которые были подвергнуты операции обновления или удаления, остаются в таблице и с этого момента становятся ретроспективными строками. Пользователям не разрешается модифицировать содержимое ретроспективных строк или периодов, связанных с любой из них. Только система, а не пользователь, может обновить периоды строк в системно-версионной таблице. Это реализуется путем обновления столбцов, не содержащих значений периодов, или в результате удаления соответствующих строк.

  Эти ограничения гарантируют, что история изменения данных защищена от фальсификаций, что отвечает стандартам контроля и удовлетворяет соответствующим законодательным актам.

Системно-версионные таблицы отличаются от таблиц с периодами прикладного времени двумя следующими аспектами в инструкциях `CREATE`, с помощью которых они создаются.

- В то время как в таблице с периодами прикладного времени пользователь может присвоить периоду любое имя, в системно-версионной таблице период должен носить имя `SYSTEM_TIME`.
- При создании системно-версионной таблицы инструкция `CREATE` должна содержать ключевые слова `WITH SYSTEM VERSIONING`. Несмотря на то что стандарт SQL:2011 позволяет для моментов начала и конца периода использовать либо тип данных `DATE`, либо один из типов `TIMESTAMP`, практически во всех ситуациях стоит отдавать предпочтение типу `TIMESTAMP`, который предоставляет возможность оперировать более высоким уровнем точности, чем день. Но какой бы тип данных вы ни выбрали для столбца начала, такой же тип данных должен иметь и столбец конца периода.

Для иллюстрации использования таблиц с системно-версионными данными воспользуемся примером со служащими и отделами компании. Системно-версионную таблицу можно создать с помощью следующего кода.

```
CREATE TABLE employee_sys (
   EmpID     INTEGER,
   Sys_Start TIMESTAMP(12) GENERATED ALWAYS AS ROW START,
   Sys_End   TIMESTAMP(12) GENERATED ALWAYS AS ROW END,
   EmpName   VARCHAR(30),
   PERIOD FOR SYSTEM_TIME (SysStart, SysEnd)
) WITH SYSTEM VERSIONING;
```

Любая строка в системно-версионной таблице считается текущей системной строкой, если текущее время входит в период системного времени. В противном случае она считается ретроспективной системной строкой.

Системно-версионные таблицы имеют много общего с таблицами с периодами прикладного времени, но нам сейчас важно рассмотреть различия между ними.

- Пользователи не могут присваивать или изменять значения в столбцах `Sys_Start` и `Sys_End`, поскольку это делается автоматически средствами СУБД благодаря использованию ключевых слов `GENERATED ALWAYS`.
- Если вы используете инструкцию `INSERT` для добавления данных в системно-версионную таблицу, то значение в столбце `Sys_Start` автоматически устанавливается равным временной метке, которая соответствует каждой транзакции. В столбец `Sys_End` записывается самое большое значение, которое может быть выражено типом данных, определенным для этого столбца.
- В системно-версионных таблицах инструкции `UPDATE` и `DELETE` выполняются только для строк, относящихся к текущей системной информации. Пользователи не могут обновлять или удалять ретроспективные системные строки.
- Пользователи не могут модифицировать значения начального или конечного времени любого периода, определяющего работу системы, как в строках, относящихся к текущей системной информации, так и в ретроспективных системных строках.
- При использовании инструкций `UPDATE` или `DELETE` для строк, относящихся к текущей системной информации, автоматически выполняется вставка ретроспективной системной строки.

  В начале выполнения инструкции `UPDATE` в системно-версионную таблицу вставляется копия старой строки, в которой время конца системного периода устанавливается равным времени выполнения

(значению TIMESTAMP) данной транзакции. Это означает, что в соответствующий момент времени данная строка перестает быть текущей. Затем СУБД выполняет обновление данных, одновременно заменяя время начала системного периода временем выполнения (значением TIMESTAMP) данной транзакции. И теперь, т.е. с момента выполнения транзакции, обновленная строка является текущей системной строкой. При этом триггеры инструкции UPDATE для рассматриваемых строк сработают, а триггеры INSERT — нет, хотя и выполняется вставка данных в таблицу ретроспективных строк (но как часть всей операции). О триггерах мы подробно поговорим в главе 23.

Инструкция DELETE, выполняемая над системно-версионной таблицей, в действительности не удаляет указанные строки, а заменяет в них системное время конца периода значением TIMESTAMP текущего системного времени. Это означает, что с момента выполнения транзакции эти строки перестают быть текущими. Отныне эти строки становятся частью измерения ретроспективного, а не текущего времени. При выполнении инструкции DELETE все триггеры для соответствующих строк должны сработать.

Назначение первичных ключей в системно-версионных таблицах

Назначать первичные ключи в системно-версионных таблицах проще, чем в таблицах периодов прикладного времени. Дело в том, что вам и не нужно иметь дело с периодами времени. В системно-версионных таблицах ретроспективные строки изменять нельзя. Будучи текущими системными строками, они прошли проверку на уникальность. Поскольку сейчас их изменить нельзя, их и не нужно больше проверять на уникальность.

Если в существующую системно-версионную таблицу добавить ограничение на первичный ключ с помощью инструкции ALTER, то, поскольку оно применяется только к текущим строкам, вам не нужно включать в инструкцию информацию о периоде. Рассмотрим пример.

```
ALTER TABLE employee_sys
  ADD PRIMARY KEY (EmpID);
```

Коротко и ясно.

Применение ограничений ссылочной целостности к системно-версионным таблицам

Применение ссылочных ограничений к системно-версионным таблицам тоже не составляет труда (по тем же причинам). Рассмотрим пример.

```
ALTER TABLE employee_sys
  ADD FOREIGN KEY (EmpDept)
      REFERENCES dept_sys (DeptID);
```

Поскольку действие здесь направлено только на текущие строки, вам не нужно включать в инструкцию столбцы начала и конца периода.

Создание запросов к системно-версионным таблицам

Большинство запросов к системно-версионным таблицам связано с событиями, которые были истинными на некоторый момент времени в прошлом или в течение некоторого периода времени в прошлом. Для решения таких задач в стандарт SQL:2011 были добавлены новые элементы синтаксиса. Чтобы извлечь из таблицы данные о том, что было истинным в заданный момент времени, следует использовать синтаксис `SYSTEM_TIME AS OF`. Предположим, вам нужно узнать, какие служащие числились в штате компании на 15 июля 2017 года. Это можно сделать с помощью следующего запроса.

```
SELECT EmpID, EmpName, Sys_Start, Sys_End
  FROM employee_sys FOR SYSTEM_TIME AS OF
       TIMESTAMP '2017-07-15 00:00:00';
```

При выполнении этой инструкции вы получите все строки, в которых начальный момент времени совпадает с заданным значением `TIMESTAMP` или предшествует ему, а оно (в свою очередь) предшествует конечному моменту времени.

Чтобы узнать, что было истинным в течение некоторого периода времени, можно использовать похожую инструкцию (с новым синтаксисом).

```
SELECT EmpID, EmpName, Sys_Start, Sys_End
  FROM employee_sys FOR SYSTEM_TIME FROM
       TIMESTAMP '2017-07-01 00:00:00' TO
       TIMESTAMP '2017-08-01 00:00:00';
```

Результат выполнения этого запроса будет включать все строки, относящиеся к промежутку времени, который начинается с первого значения `TIMESTAMP` и заканчивается вторым значением `TIMESTAMP`, *не* включая его.

Если запрос к системно-версионной таблице не включает спецификацию `TIMESTAMP`, то по умолчанию запрос возвращает только текущие системные строки. Код такого запроса может иметь следующий вид.

```
SELECT EmpID, EmpName, Sys_Start, Sys_End
  FROM employee_sys;
```

Если нужно извлечь из системно-версионной таблицы строки обоих видов, т.е. как ретроспективные, так и текущие, можно использовать следующий синтаксис.

```
SELECT EmpID, EmpName, Sys_Start, Sys_End
  FROM employee_sys FOR SYSTEM_TIME FROM
       TIMESTAMP '2013-07-01 00:00:00' TO
       TIMESTAMP '9999-12-31 24:59:59';
```

Отслеживание данных с помощью битемпоральных таблиц

Иногда нужно знать, когда в действительности произошло некоторое событие и когда оно было записано в базу данных. Для подобных ситуаций можно использовать таблицу, которая одновременно является как системно-версионной, так и таблицей с периодами прикладного времени. Такие таблицы получили название *битемпоральных*.

Существует ряд случаев, когда битемпоральные таблицы очень нужны. Предположим, один из служащих решил переехать из штата Орегон в штат Вашингтон. Вы должны учесть этот факт, поскольку с момента официальной даты переезда из одного штата в другой сумма удержания подоходного налога штата должна измениться. Однако весьма маловероятно, что такое изменение будет внесено в базу данных в день переезда. Поэтому необходимо зарегистрировать оба значения даты/времени, и битемпоральная таблица подходит для этого как нельзя лучше. Запись периода системно-версионного времени выполняется тогда, когда об изменении стало известно базе данных, а запись периода прикладного времени — когда переезд вступил в силу с юридической точки зрения. Вот пример кода, создающего такую таблицу.

```
CREATE TABLE employee_bt (
   EmpID          INTEGER,
   EmpStart       DATE,
   EmpEnd         DATE,
   EmpDept        Integer
   PERIOD FOR EmpPeriod (EmpStart, EmpEnd),
   Sys_Start TIMESTAMP (12) GENERATED ALWAYS
      AS ROW START,
   Sys_End TIMESTAMP (12) GENERATED ALWAYS
      AS ROW END,
   EmpName        VARCHAR (30),
   EmpStreet      VARCHAR (40),
   EmpCity        VARCHAR (30),
   EmpStateProv   VARCHAR (2),
   EmpPostalCode VARCHAR (10),
   PERIOD FOR SYSTEM_TIME (Sys_Start, Sys_End),
   PRIMARY KEY (EmpID, EPeriod WITHOUT OVERLAPS),
```

```
   FOREIGN KEY (EDept, PERIOD EPeriod)
   REFERENCES Dept (DeptID, PERIOD DPeriod)
) WITH SYSTEM VERSIONING;
```

Назначение битемпоральных таблиц заключается в создании как системно-версионных таблиц, так и таблиц с периодами прикладного времени. Значения для столбцов начала и конца периодов прикладного времени предоставляет пользователь. Любая инструкция `INSERT`, выполняющая вставку данных в такую таблицу, автоматически устанавливает значение периода системного времени в соответствии с меткой `TIMESTAMP` транзакции.

Значение столбца, содержащего конец периода системного времени, автоматически устанавливается равным самому большому значению, разрешенному для типа данных этого столбца. Инструкции `UPDATE` и `DELETE` работают так, как это определено для стандартных таблиц с периодами прикладного времени. Но в случае системно-версионных таблиц инструкции `UPDATE` и `DELETE` оказывают влияние только на строки текущего времени, и при выполнении каждой такой инструкции автоматически вставляется ретроспективная строка.

В запросе к битемпоральной таблице можно задавать период прикладного времени, период системно-версионного времени или оба периода. Вот пример последнего случая (когда задаются оба периода).

```
SELECT EmpID
  FROM employee_bt FOR SYSTEM TIME AS OF
       TIMESTAMP '2017-07-15 00:00:00'
 WHERE EmpID = 314159 AND
       EmpPeriod CONTAINS DATE '2017-06-20 00:00:00';
```

Форматирование и анализ значений даты и времени

Стандарт языка, в частности, международный стандарт SQL, должен описывать его официальный синтаксис, а также порядок форматирования элементов языка, таких как значения даты и времени. Но, что удивительно, до появления версии SQL:2016 ничего такого не было. Значения даты и времени выражались различными способами, и стандартный формат даты/времени, приятый в США, например, отличается от стандартного европейского формата даты/времени.

В стандарте SQL:2016 определяются способы представления единиц времени с помощью шаблона, что позволяет понять, как выглядят фактические единицы даты и времени. Например, дата 16 сентября 2018 года может быть представлена в виде `'09-16-2018'` в инструкции SQL. Шаблон, который

соответствует ожидаемому формату даты, может выглядеть как 'MM-DD-YYYY'. Он сообщает разработчику о том, что сначала нужно представить данные в виде месяца, затем — дня, а затем — года. Альтернативный шаблон 'DD-MM-YYYY' сообщает, что нужно сначала указать день, затем — месяц, а затем — год. Записи MM, DD и YYYY являются заполнителями, которые должны быть заменены в инструкции SQL конкретным значением месяца, дня и года.

Помимо заполнителей MM, DD и YYYY, существует ряд других заполнителей, которые имеют специальное значение. Эти заполнители приведены в табл. 7.6.

Таблица 7.6. Заполнители шаблона и соответствующие им значения

Заполнитель	Значение
YYYY | YYY | YY | Y	Год
RRRR | RR	Округленный год
MM	Месяц
DD	День месяца
DDD	День года
HH | HH12	Час, по 12-часовой шкале
HH24	Час, по 24-часовой шкале
MI	Минута
SS	Секунда минуты
SSSS	Секунда дня
FF1 | FF2 | .... | FF9	Доля секунды
A.M. | P.M.	Время AM или PM
TZH	Час часового пояса
TZM	Минута часового пояса

Шаблоны формата вместе с указанными заполнителями можно использовать при определении даты/времени с помощью спецификации пути JSON, как описано в главе 19, и в выражении CAST, как описано в главе 9.

Глава 8

Обработка значений

В ЭТОЙ ГЛАВЕ...

- Использование переменных для устранения избыточного кодирования
- Получение часто запрашиваемой информации из табличных полей базы данных
- Объединение простых значений для формирования сложных выражений

В книге постоянно подчеркивается, насколько важна структура базы данных для поддержки ее целостности. Впрочем, несмотря на то что важность структуры базы данных часто недооценивается, не стоит забывать, что наибольшую ценность все же представляют сами данные. В конце концов, именно значения, хранящиеся на пересечении строк и столбцов в таблице базы данных, являются тем "сырьем", "переработка" которого позволит сделать выводы о взаимосвязях и тенденциях.

Значения можно представить несколькими способами: непосредственно либо с помощью функций или выражений. В этой главе описываются различные виды значений, а также функции и выражения.

ЗАПОМНИ!

Функции получают данные и на их основе вычисляют значения. *Выражение* представляет собой цепочку элементов данных, после вычисления которой SQL выдает единственное значение.

Значения

В SQL различаются следующие виды значений:

- записи;
- литералы;
- переменные;
- специальные переменные;
- ссылки на столбцы.

АТОМЫ ТОЖЕ НЕ ЯВЛЯЮТСЯ НЕДЕЛИМЫМИ

В XIX веке ученые считали, что атом является той наименьшей частью материи, какая только возможна. Поэтому они и назвали эту часть *атомом* — словом, происходящим от греческого "атомос", что означает "неделимый". Сегодня 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. Значение типа `DATE` состоит из следующих компонентов: год, месяц и день. А компонентами значения типа `TIMESTAMP` являются час, минута, секунда и т.д. Значения типов `REAL` и `FLOAT` в качестве компонентов включают *экспоненту* и *мантиссу*. В значении типа `CHAR` есть компоненты, к которым можно получить доступ с помощью функции выделения подстроки (`SUBSTRING`). Поэтому называть значения полей в базах данных *атомарными* (по аналогии с атомами материи) вполне резонно. Впрочем, если исходить из первоначального значения этого слова, то ни одно из современных применений термина *атомарный* не является правильным.

Записи

Самыми заметными значениями в базе данных являются табличные *записи*. Это содержимое каждой строки, хранимой в таблице базы данных. Значение этого типа обычно состоит из множества компонентов, поскольку каждый столбец в строке имеет свое значение. На пересечении одного столбца и одной строки находится *поле*. В поле содержится *скалярное* (или *атомарное*) значение, у которого имеется только один компонент.

Литералы

В SQL значение может быть представлено либо переменной, либо константой. Вполне логично, что значение *переменной* время от времени может изменяться, а значение *константы* (т.е. постоянной величины) не изменяется никогда. Важной разновидностью констант является *литерал*, само представление которого и является значением.

В SQL предусмотрено много разных типов данных и, соответственно, много разных типов литералов. Некоторые примеры литералов приведены в табл. 8.1. Обратите внимание на то, что литералы нечисловых типов заключены в одинарные кавычки. Эти знаки помогают избежать путаницы, хотя, впрочем, могут и вызывать проблемы.

Таблица 8.1. Примеры литералов различных типов

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)	'ГРЕЦИЯ '
Примечание: в предыдущей строке в одинарные кавычки заключено пятнадцать символов, в том числе пробелы	
VARCHAR (CHARACTER VARYING)	'лептон'
NATIONAL CHARACTER (15)	'ELLAS '[1]
Примечание: в предыдущей строке в одинарные кавычки заключено пятнадцать символов, в том числе пробелы	
NATIONAL CHARACTER VARYING(15)	'lepton'[2]
CHARACTER LARGE OBJECT(512) (CLOB(512))	*Очень длинная символьная строка*

Окончание табл. 8.1

Тип данных	Пример литерала
`BINARY(4)`	`'01001100011100001111000111001010'`
`VARBINARY(4)` (`BINARY VARYING(4)`)	`'0100110001110000'`
`BINARY LARGE OBJECT (512)` (`BLOB(512)`)	*Очень длинная строка, состоящая из нулей и единиц*
`DATE`	`DATE '1969-07-20'`
`TIME(2)`	`TIME '13.41.32.50'`
`TIMESTAMP(0)`	`TIMESTAMP '2018-02-25-13.03.16.000000'`
`TIME WITH TIMEZONE(4)`	`TIME '13.41.32.5000-08.00'`
`TIMESTAMP WITH TIMEZONE(0)`	`TIMESTAMP '2018-02-25-13.03.16.0000+02.00'`
`INTERVAL DAY`	`INTERVAL '7' DAY`

[1] Этим словом греки называют Грецию на своем языке. (Если написать его по-английски, то получится "Hellas", а по-русски — "Эллада".)

[2] Это слово "lepton" (лепта — разменная монета Греции до введения евро), написанное по-гречески.

А что если литерал является символьной строкой, содержащей символ одинарной кавычки? В таком случае вместо одного этого символа в литерале должны стоять две одинарные кавычки подряд, чтобы показать, что кавычка является частью строки и не указывает на ее завершение. Таким образом, чтобы получился символьный литерал `'Паб О'Брайенс'`, необходимо ввести `'Паб О''Брайенс'`.

Переменные

Прекрасно, когда при работе с базами данных можно манипулировать литералами и другими константами. Однако полезно иметь и переменные. Во многих ситуациях без них пришлось бы выполнять гораздо больше работы. *Переменная* — это величина, значение которой может изменяться. Чтобы понять, почему переменные полезны, рассмотрим пример.

Предположим, вы розничный продавец, у которого есть покупатели нескольких категорий. Тем из них, кто покупает в больших объемах, вы продаете товары по самым низким ценам. Тем же, кто покупает в средних объемах, вы

продаете товары по более высоким ценам. И наконец, те, кто ограничивается покупкой небольшого количества товаров, платят самую высокую цену. Обычно все розничные цены имеют определенные коэффициенты по отношению к оптовой стоимости их закупки. Предположим, что для товара F-35 вы решили, что крупнооптовые покупатели (покупатели класса C) будут платить за него в 1,4 раза больше, чем платите за этот товар вы. Среднеоптовые покупатели (покупатели класса B) будут уже платить в 1,5 раза больше. И наконец, мелкооптовые покупатели (покупатели класса A) — в 1,6 раза больше.

Вы храните значения стоимости товаров и устанавливаемых вами цен в таблице `PRICING` (ценообразование). Среди ее полей можно найти следующие: `Price` (цена), `Cost` (стоимость), `Product` (продукт) и `Class` (класс). Чтобы реализовать новую структуру ценообразования, достаточно выполнить следующие SQL-инструкции.

```
UPDATE PRICING
   SET Price = Cost * 1.4
 WHERE Product = 'F-35'
   AND Class = 'C' ;
UPDATE PRICING
   SET Price = Cost * 1.5
 WHERE Product = 'F-35'
   AND Class = 'B' ;
UPDATE PRICING
   SET Price = Cost * 1.6
 WHERE Product = 'F-35'
   AND Class = 'A' ;
```

Этот код неплох и, в принципе, решает ваши задачи. Но что делать, если происки конкурентов начинают подрывать ваш сектор рынка? Чтобы остаться "на плаву", вам, скорее всего, придется уменьшить установленную вами маржу между ценами закупки и продажи. Тогда потребуется ввести нечто похожее на такую последовательность инструкций.

```
UPDATE PRICING
   SET Price = Cost * 1.25
 WHERE Product = 'F-35'
   AND Class = 'C' ;
UPDATE PRICING
   SET Price = Cost * 1.35
 WHERE Product = 'F-35'
   AND Class = 'B' ;
UPDATE PRICING
   SET Price = Cost * 1.45
 WHERE Product = 'F-35'
   AND Class = 'A' ;
```

Если рынок изменчив, вам придется регулярно переписывать свой SQL-код. Это потребует немалых усилий, особенно если цены указаны во многих местах кода. Усилия можно свести к минимуму, если заменить литералы (такие, как `1.45`) переменными (такими, например, как `:multiplierA`). Тогда операции обновления можно выполнять следующим образом.

```
UPDATE PRICING
   SET Price = Cost * :multiplierC
 WHERE Product = 'F-35'
   AND Class = 'C' ;
UPDATE PRICING
   SET Price = Cost * :multiplierB
 WHERE Product = 'F-35'
   AND Class = 'B' ;
UPDATE PRICING
   SET Price = Cost * :multiplierA
 WHERE Product = 'F-35'
   AND Class = 'A' ;
```

Теперь, когда ситуация на рынке заставит вас пересмотреть политику ценообразования, вам придется всего лишь изменить значения переменных `:multiplierC`, `:multiplierB` и `:multiplierA`. Эти переменные являются параметрами, передаваемыми SQL-коду, который использует их для формирования новых цен.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Иногда переменные, используемые таким образом, называют *параметрами*, а иногда — *базовыми переменными*. Переменные называются параметрами, если они находятся в приложениях, написанных на модульном варианте SQL, и базовыми — если используются во встроенном SQL.

ЗАПОМНИ!

Термин *встроенный SQL* подразумевает, что SQL-инструкции встроены в код приложения, написанного на некотором процедурном языке. В качестве альтернативы можно использовать модульный SQL для написания целого модуля, который затем будет вызываться приложением, написанным на процедурном языке. Каждый из этих двух подходов имеет свои преимущества и недостатки. Какой из них выбрать, зависит от конкретной реализации SQL.

Специальные переменные

Как только пользователь на клиентском компьютере подключится к базе данных, находящейся на сервере, устанавливается сеанс связи. Если пользователь соединяется с несколькими базами данных, то сеанс, связанный с последним соединением, считается *текущим*, а предыдущие — *неактивными*.

Стандарт SQL определяет несколько специальных переменных, применяемых в многопользовательских системах. Эти переменные содержат данные о различных пользователях.

- SESSION_USER. Эта специальная переменная содержит идентификатор авторизации пользователя текущего сеанса SQL. Используя переменную SESSION_USER, можно написать программу мониторинга, определяющую, кто выполняет SQL-инструкции.
- CURRENT_USER. Любой модуль SQL может иметь связанный с ним идентификатор авторизации, определяемый пользователем. Его значение хранится в переменной CURRENT_USER. Если такого идентификатора у модуля нет, то переменная CURRENT_USER содержит то же значение, что и SESSION_USER.
- SYSTEM_USER. В данной переменной хранится идентификатор пользователя операционной системы. Он может отличаться от идентификатора того же пользователя, хранящегося в SQL-модуле. К примеру, пользователь может зарегистрироваться в системе как LARRY, а в модуле — как PLANT_MGR, в результате чего в переменной SESSION_USER будет храниться значение PLANT_MGR. Если этот пользователь явно не определяет идентификатор модуля и, следовательно, переменная CURRENT_USER также получает значение PLANT_MGR, то в переменной SYSTEM_USER будет храниться значение LARRY.


СОВЕТ

Специальные переменные SYSTEM_USER, SESSION_USER и CURRENT_USER используются для сбора данных о пользователях, работающих в системе. Вы можете создать специальную таблицу регистрации (журнал) и периодически вставлять в нее значения, содержащиеся в этих переменных. Как это сделать, показано в следующем примере.

```
INSERT INTO USAGELOG (SNAPSHOT)
     VALUES ('User ' || SYSTEM_USER ||
             ' with ID ' || SESSION_USER ||
             ' active at ' || CURRENT_TIMESTAMP) ;
```

При выполнении этой инструкции создаются примерно такие записи журнала:

```
User LARRY with ID PLANT_MGR active at 2018-04-07-23.50.00
```

Ссылки на столбцы

Каждый столбец содержит в каждой строке таблицы всего одно значение. Инструкции SQL часто ссылаются на эти значения. Полностью определенная

ссылка на столбец состоит из имени таблицы, символа точки и имени столбца (например, PRICING.Product). Рассмотрим следующую инструкцию.

```
SELECT PRICING.Cost
  FROM PRICING
 WHERE PRICING.Product = 'F-35' ;
```

Здесь PRICING.Product — ссылка на столбец. Эта ссылка имеет значение 'F-35'. Элемент PRICING.Cost — это также ссылка на столбец, но нам не будет известно ее значение, пока не будет выполнена инструкция SELECT.


СОВЕТ

Поскольку имеет смысл делать ссылки на столбцы, находящиеся только в текущей таблице, как правило, эти ссылки полностью определять необязательно. Например, следующая инструкция равнозначна предыдущей.

```
SELECT Cost
  FROM PRICING
 WHERE Product = 'F-35' ;
```

В то же время иногда приходится работать одновременно с разными таблицами. В базе данных у каких-либо двух таблиц могут быть определены столбцы с одинаковыми именами. И тогда ссылки на такие столбцы необходимо определять полностью, чтобы получить именно то, что вам нужно.

Предположим, например, что некоторая компания имеет филиалы, расположенные в Кингстоне и Джефферсон-Сити, и отдельно для каждого из этих филиалов вы отслеживаете данные о сотрудниках. Таблица сотрудников, работающих в Кингстоне, называется EMP_KINGSTON, а работающих в Джефферсон-Сити — EMP_JEFFERSON. Допустим, нам необходим список всех сотрудников, которые одновременно работают в обоих филиалах. Таким образом, нам следует найти всех сотрудников, данные о которых находятся в обеих таблицах. Для решения задачи создадим следующую инструкцию SELECT.

```
SELECT EMP_KINGSTON.FirstName, EMP_KINGSTON.LastName
  FROM EMP_KINGSTON, EMP_JEFFERSON
 WHERE EMP_KINGSTON.EmpID = EMP_JEFFERSON.EmpID ;
```

Поскольку идентификатор сотрудника является уникальным и имеет одно и то же значение независимо от филиала, в котором работает сотрудник, этот номер можно использовать для связи между таблицами (в каждой из них он находится в столбце EmpID). В результате выполнения приведенной инструкции возвращаются имена (FirstName) и фамилии (LastName) только тех сотрудников, чьи данные хранятся в обеих таблицах.

Выражения

Выражения могут быть как простыми, так и очень сложными. Они могут содержать литералы, имена столбцов, параметры, базовые переменные, подзапросы, логические и арифметические операторы. Впрочем, каким бы сложным ни было выражение, оно обязательно должно выдавать в результате одиночное значение.

По этой причине SQL-выражения обычно называют *выражениями со значением*. Объединение нескольких таких выражений в одно возможно тогда, когда составные части (т.е. отдельные выражения) приводятся к значениям, имеющим совместимые типы данных.

В SQL определено пять различных типов выражений:

- строковые;
- числовые;
- даты/времени;
- интервальные;
- условные.

Строковые выражения

Простейшим *строковым выражением* является одиночное строковое значение. Более сложные варианты могут включать ссылки на столбцы, итоговые функции, скалярные подзапросы (описаны в главе 12), выражения с использованием ключевых слов `CASE` и `CAST` (описаны в главе 9) или составные строковые выражения.

В строковых выражениях можно использовать только *оператор конкатенации*. С его помощью можно объединить любые допустимые выражения для получения более сложного строкового выражения. Оператор конкатенации обозначается двумя вертикальными линиями (`||`). Некоторые примеры строковых выражений показаны в следующей таблице.

Выражение	Результат
`'Хрустящий' || 'арахис'`	`'Хрустящий арахис'`
`'Шарики' || ' ' || 'из желе'`	`'Шарики из желе'`
`FIRST_NAME || ' ' || LAST_NAME`	`'Джон Смит'`
`B'1100111' || B'01010011'`	`B'110011101010011'`

Выражение	Результат
`'' || 'Спаржа'`	`'Спаржа'`
`'Спаржа' || ''`	`'Спаржа'`
`'С' || '' || 'пар' || '' || 'жа'`	`'Спаржа'`

Как видно из таблицы, если объединять какую-либо строку со строкой нулевой длины, то результатом будет сама исходная строка.

Числовые выражения

Числовое выражение состоит из данных числового типа, к которым могут быть применены операторы сложения, вычитания, умножения и деления. При вычислении такого выражения обязательно должно получиться числовое значение. Компоненты, составляющие числовое выражение, могут иметь разные типы данных, но *все* они должны быть числовыми. Тип данных результата зависит от типов данных его компонентов. В стандарте SQL нет жесткого определения, как именно тип данных исходных компонентов влияет на тип результата выражения. Все зависит от конкретной аппаратной платформы. Поэтому, если вы используете в одном числовом выражении разные типы данных, для получения четкого представления о типе результата обратитесь к документации по той платформе, на которой работаете.

Ниже приведены некоторые примеры числовых выражений:

- `-27`
- `49 + 83`
- `5 * (12 - 3)`
- `PROTEIN + FAT + CARBOHYDRATE`
- `FEET/5280`
- `COST * :multilierA`

Выражения со значениями даты/времени

Выражения со значенимями даты/времени выполняют операции с данными, имеющими отношение к дате и времени. Компоненты этих выражений могут иметь типы `DATE`, `TIME`, `TIMESTAMP` и `INTERVAL`. Результат такого выражения 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
```

Время задается по шкале UTC (Всемирное координированное время), ранее известной как время по Гринвичу. Но при желании можно указать смещение, чтобы время соответствовало нужному часовому поясу. Для местного часового пояса, используемого в вашей системе, можно использовать следующий простой синтаксис:

```
TIME '22.55.00' AT LOCAL
```

Кроме того, это значение можно задать и в развернутом виде:

```
TIME '22.55.00' AT TIME ZONE INTERVAL '-08.00' HOUR TO MINUTE
```

Последнее выражение определяет местное время часового пояса, в котором находится Портленд (штат Орегон). Этот часовой пояс отстоит от Гринвича на восемь часов.

Интервальные выражения

Если взять два значения даты/времени и из одного вычесть другое, то получится *интервал*. А вот сложение таких значений не имеет смысла, поэтому SQL не поддерживает такую операцию. Если же сложить два интервала или вычесть один из другого, то в результате снова получится интервал. Кроме того, интервал можно умножать или делить на числовую константу.

В SQL существуют два типа интервалов: *год–месяц* и *день–время*. Чтобы избежать двусмысленности, в интервальном выражении необходимо 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 - CURRENT_DATE) YEAR TO MONTH
```

А это выражение возвращает интервал в 40 дней:

```
INTERVAL '17' DAY + INTERVAL '23' DAY
```

Ниже приблизительно подсчитывается общее количество месяцев, в течение которых мать пятерых детей была беременна, при условии, что сейчас она не ждет шестого ребенка.

```
INTERVAL '9' MONTH * 5
```

Интервалы могут быть как положительными, так и отрицательными и состоять из любого выражения со значением или из объединения таких выражений, результатом вычисления которых является интервал.

Условные выражения

Значение условного выражения 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 речь о них пойдет в главе 9.

Функции

Функция — это простая (или не очень) операция, которую обычные SQL-инструкции не могут выполнить, но которая тем не менее достаточно часто встречается на практике. В SQL существуют функции, выполняющие действия, которые в их отсутствие пришлось бы выполнять приложению, написанному на языке высокого уровня (в такое приложение как раз и вставляются SQL-инструкции). Существуют две основные разновидности функций: *итоговые* (или агрегированные) и *функции преобразований*.

Статистические вычисления с помощью итоговых функций

Итоговые функции применяются к *наборам строк* из таблицы, а не к отдельной строке. Эти функции обрабатывают текущий набор строк и вычисляют некоторые его обобщенные характеристики. В такой набор могут входить все строки таблицы или только те из них, которые отфильтровываются предложением `WHERE` (подробно о предложении `WHERE` рассказывается в главе 10). Программисты называют такие функции *итоговыми*, поскольку они берут информацию из целого набора строк, определенным образом ее обрабатывают и выдают результат в виде одной строки. Этот итоговый результат представляет собой *обобщение* (агрегирование) данных из всех строк, составляющих набор.

Для того чтобы получить представление о применении итоговых функций, рассмотрим табл. 8.2, в которой представлены сведения о компонентах, содержащихся в продуктах питания.

Таблица 8.2. Компоненты продуктов питания (в 100 граммах)

Продукт питания	Калории	Белки, г	Жиры, г	Углеводы, г
Жареный миндаль	627	18,6	57,7	19,6
Спаржа	20	2,2	0,2	3,6
Сырые бананы	85	1,1	0,2	22,2

Окончание табл. 8.2

Продукт питания	Калории	Белки, г	Жиры, г	Углеводы, г
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	111	7,6	0,5	19,8
Кола	39			10,0
Белый хлеб	269	8,7	3,2	50,4
Цельнозерновой 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
Жевательные конфеты	367		0,5	93,1
Хрустящий арахис	421	5,7	10,4	81,0

Информация из табл. 8.2 хранится в таблице FOODS (продукты питания) базы данных. В пустых полях находятся значения NULL. С помощью итоговых функций COUNT, AVG, MAX, MIN и SUM мы можем узнать важные для здоровья факты.

Функция COUNT

Функция COUNT сообщает, сколько строк находится в таблице или сколько строк таблицы удовлетворяют заданным условиям. Вот пример простого применения этой функции.

```
SELECT COUNT (*)
  FROM FOODS ;
```

Функция возвращает результат, равный общему количеству строк таблицы FOODS, в данном случае это число 15. Тот же результат мы получим и при выполнении следующей инструкции.

```
SELECT COUNT (Calories)
  FROM FOODS ;
```

Так как значение было введено в каждую строку столбца Calories (калории), подсчет строк даст тот же результат. Если бы в некоторых строках этого столбца находились пустые значения, они не были бы включены в результат.

Следующий запрос к таблице FOODS возвращает число 11, так как в четырех из пятнадцати строк столбца Carbohydrate (углеводы) находится значение NULL.

```
SELECT COUNT (Carbohydrate)
  FROM FOODS ;
```


СОВЕТ

Поле таблицы базы данных может содержать пустое значение по разным причинам. Самыми распространенными являются следующие: значение неизвестно (или пока неизвестно) либо известно, но еще не введено. Иногда, если значение какого-либо поля равно нулю, оператор базы данных, вводящий данные, обходит это поле стороной и, таким образом, оставляет в нем значение NULL. Так поступать не следует, поскольку нуль (число 0) все же является определенным значением, которое учитывается при подсчетах. А NULL *не* является значением, и SQL не включает его в процесс вычислений.

Чтобы узнать, сколько в столбце различных значений, можно использовать функцию COUNT с ключевым словом DISTINCT. Рассмотрим следующую инструкцию.

```
SELECT COUNT (DISTINCT Fat)
  FROM FOODS ;
```

Она возвращает значение, равное 12. В таблице мы видим, что в 100-граммовой порции спаржи содержится столько же жиров (0,2 г), сколько и в 100 г бананов, а в 100-граммовой порции фасоли — ровно столько же жиров (0,5 г), сколько в 100 г желе. Таким образом, в таблице находится всего 12 разных значений, имеющих отношение к содержанию жиров.

Функция AVG

Функция AVG вычисляет и возвращает среднее арифметическое всех значений, хранящихся в заданном столбце. Разумеется, эту функцию можно применять только к столбцам с числовыми данными, как в следующем примере.

```
SELECT AVG (Fat)
  FROM FOODS ;
```

Результатом станет среднее содержание жиров во всех продуктах, равное 15,37 г. Это число достаточно высокое, так как общую картину портит информация о сливочном масле. Возможно, вы зададите себе вопрос: "А каким было

бы среднее содержание жиров, если бы не учитывалось масло?" Чтобы ответить на него, в наш запрос можно включить предложение WHERE.

```
SELECT AVG (Fat)
  FROM FOODS
 WHERE FOOD <> 'Butter' ;
```

В этом случае среднее содержание жиров в 100 г пищевых продуктов падает до 10,32 г.

Функция MAX

Возвращает максимальное значение из всех, содержащихся в заданном столбце. Следующая инструкция вернет значение, равное 81 (количество жиров в 100 г сливочного масла).

```
SELECT MAX (Fat)
  FROM FOODS ;
```

Функция MIN

Возвращает минимальное значение из всех, содержащихся в заданном столбце. Следующая инструкция вернет значение, равное 0,4, так как функция не учитывает пустые значения.

```
SELECT MIN (Carbohydrate)
  FROM FOODS ;
```

Функция SUM

Возвращает сумму всех значений заданного столбца. Следующая инструкция возвращает число 3924, которое является общим количеством калорий во всех пятнадцати продуктах.

```
SELECT SUM (Calories)
  FROM FOODS ;
```

Функция LISTAGG

В SQL:2016 появилась новая итоговая функция LISTAGG, которая агрегирует значения группы строк таблицы в список значений с разделителями, такими как запятая, которые можно указать. Обычно эта возможность используется для преобразования агрегированных табличных значений в строку значений, разделенных запятыми (как в файле CSV). Синтаксис функции LISTAGG таков:

```
LISTAGG (<выражение>, <разделитель>) WITHIN GROUP (ORDER BY имя_поля, ...)
```

В качестве примера предположим, что в базе данных находится таблица EMPLOYEE, которая, в свою очередь, включает записи EmployeeID, FirstName,

LastName и DepartmentID для каждого из сотрудников. Дополнительно предположим, что нужно вывести информацию по всем сотрудникам, сгруппированную по отделам и выводимую в алфавитном порядке по фамилии каждого сотрудника. Подобный листинг можно создать с помощью следующего запроса.

```
SELECT DepartmentID,
       LISTAGG(LastName, ',') WITHIN GROUP (ORDER BY LastName)
       AS Employees
  FROM EMPLOYEE
 GROUP BY DepartmentID ;
```

В результате выполнения этого запроса отобразится табличный набор данных, включающий столбцы DepartmentID и Employees. Каждая строка в результирующем наборе данных будет содержать идентификатор DepartmentID отдела, за которым следует разделенный запятыми список сотрудников этого отдела. Строки будут упорядочены в алфавитном порядке по идентификатору DepartmentID, и имена сотрудников в каждой строке тоже будут упорядочены в алфавитном порядке.

Функции преобразований

В разных ситуациях порой используются одинаковые наборы операций. Поскольку к этим операциям приходится прибегать довольно часто, стандартом ISO/IEC SQL они были определены в виде *функций преобразования*. Конечно, по сравнению с такими СУБД, как Access, Oracle или SQL Server, в SQL этих функций довольно-таки мало, но те немногие, которые есть, вы будете применять очень часто. В SQL имеются четыре категории таких функций:

- строковые;
- числовые;
- даты/времени;
- интервальные.

Строковые функции

Строковые функции получают значение одной символьной строки и возвращают в качестве результата другую символьную строку. В SQL определено десять таких функций:

- SUBSTRING;
- SUBSTRING SIMILAR;
- SUBSTRING_REGEX;
- TRANSLATE_REGEX;

- OVERLAY;
- UPPER;
- LOWER;
- TRIM;
- TRANSLATE;
- CONVERT.

Функция SUBSTRING

Используется для извлечения подстроки из исходной строки. Тип извлеченной подстроки будет совпадать с типом исходной строки. Например, если исходная строка имеет тип CHARACTER VARYING, то и полученная в результате подстрока будет иметь тип CHARACTER VARYING. Вот как выглядит синтаксис функции SUBSTRING.

```
SUBSTRING (строковое_значение FROM начало [FOR длина])
```

Предложение в квадратных скобках ([]) является необязательным. Подстрока, извлекаемая из элемента *строковое_значение*, начинается с символа, порядковый номер которого указан в элементе *начало*. Нумерация начинается с первого символа. Длина извлекаемой подстроки указывается в элементе *длина*. Если предложение FOR отсутствует, то подстрока извлекается от символа, соответствующего элементу *начало*, до конца строки. Рассмотрим следующий пример.

```
SUBSTRING ('Цельнозерновой хлеб' FROM 16 FOR 4)
```

Извлеченной подстрокой является 'хлеб'. Она начинается с 16-го символа исходной строки и имеет длину 4 символа. На первый взгляд, функция SUBSTRING не кажется такой уж ценной. Чтобы найти слово 'хлеб' в литерале 'Цельнозерновой хлеб', можно было бы обойтись и без функции выделения подстроки. Но в качестве элемента *строковое_значение* не всегда используется литерал. Это значение может быть любым выражением, при вычислении которого получается символьная строка. Например, элемент *строковое_значение* может быть выражен переменной fooditem, которая каждый раз может иметь разные значения. При выполнении следующей функции будет извлечена фиксированная подстрока независимо от того, какую символьную строку хранит переменная fooditem.

```
SUBSTRING (:fooditem FROM 16 FOR 4)
```

Все функции преобразования объединяет то, что они могут оперировать как литералами, так и выражениями, после вычисления которых получаются значения соответствующего типа.

ВНИМАНИЕ!

При использовании функции SUBSTRING не следует забывать о следующем. Извлекаемая подстрока обязательно должна быть частью исходной строки. Если вы задаете извлекаемую подстроку, которая начинается с 16-го символа, а в исходной строке их только четыре, результатом станет значение NULL. Поэтому, используя функцию SUBSTRING, необходимо иметь некоторое представление о структуре данных. Кроме того, не следует указывать отрицательную длину подстроки, так как конец не может быть перед началом.

Если столбец имеет тип данных VARCHAR, то ширина поля конкретной строки не будет известна. Для функции SUBSTRING отсутствие такой информации не важно. Если вы укажете длину подстроки, при поиске которой произойдет выход за правый край поля, функция SUBSTRING вернет то, что найдет, а не ошибку.

Рассмотрим следующую инструкцию.

```
SELECT * FROM FOODS
 WHERE SUBSTRING (Food FROM 7 FOR 7) = 'хлеб' ;
```

Даже несмотря на то что значение, находящееся в столбце Food таблицы FOODS ('Белый хлеб'), имеет длину менее 14 символов, эта инструкция все равно вернет строку с данными о белом хлебе.

СОВЕТ

Если какой-либо *операнд* (т.е. значение, из которого с помощью определенного оператора получается другое значение) функции SUBSTRING приводится к значению NULL, результатом выполнения этой функции будет NULL.

Функция SUBSTRING SIMILAR

Функция SUBSTRING SIMILAR работает с тремя параметрами: исходной символьной строкой, строкой шаблона и управляющим символом. Для извлечения результирующей строки из исходной символьной строки проверяется *совпадение с шаблоном* (на базе POSIX-ориентированных регулярных выражений).

Для разделения строки шаблона на три части используются два управляющих символа, каждый из которых сопровождается символом двойной кавычки.

Предположим, например, что исходная символьная строка S содержит такой текст: 'Белый хлеб выпекают из пшеничной муки'. Предположим также, что строка шаблона R содержит текст 'хлеб '/"'выпекают'/"' из', где косая черта (/) — это управляющий символ.

Таким образом, следующий код вернет в результате среднюю часть строки шаблона, в данном случае — подстроку 'выпекают'.

```
SUBSTRING S SIMILAR TO R ;
```

Функция SUBSTRING_REGEX

Функция SUBSTRING_REGEX ищет в строке шаблон, заданный в виде регулярного выражения XQuery, и возвращает одно вхождение совпадающей подстроки.

Согласно международному стандарту JTC 1/SC 32 синтаксис функции извлечения подстроки, заданной в виде регулярного выражения, имеет следующий вид.

```
SUBSTRING_REGEX <левая круглая скобка>
   <шаблон XQuery>[ FLAG <флаг параметра XQuery> ]
   IN <предметная строка регулярного выражения>
   [ FROM <начальная позиция> ]
   [ USING <единицы символьной длины> ]
   [ OCCURRENCE <вхождение регулярного выражения>]
   [ GROUP <группа фиксации регулярного выражения>]
   <правая круглая скобка>
```

- *<шаблон XQuery>* — строковое выражение, значением которого является регулярное выражение XQuery;
- *<флаг параметра XQuery>* — необязательная символьная строка, соответствующая аргументу `$flags` функции `fn:match` XQuery (из раздела описания функций и операторов — XQuery F&O);
- *<предметная строка регулярного выражения>* — символьная строка, проверяемая на совпадение с *<шаблон XQuery>*;
- *<начальная позиция>* — необязательное точное числовое значение, указывающее символьную позицию начала поиска (по умолчанию используется значение 1);
- *<единицы символьной длины>* — значение, указывающее единицу измерения для определения элемента *<начальная позиция>*: CHARACTERS или OCTETS (значение по умолчанию — CHARACTERS);
- *<вхождение регулярного выражения>* — необязательное точное числовое значение, указывающее, какое вхождение совпадения необходимо использовать (значение по умолчанию — 1);
- *<группа фиксации регулярного выражения>* — необязательное точное числовое значение, указывающее требуемую группу фиксации соответствия (по умолчанию используется значение 0, означающее все вхождение).

Ниже приведено несколько примеров использования функции SUBSTRING_REGEX.

```
SUBSTRING_REGEX ('\p{L}*' IN 'Just do it.') = 'Just'
SUBSTRING_REGEX ('\p{L}*' IN 'Just do it.' FROM 2) = 'ust'
SUBSTRING_REGEX ('\p{L}*' IN 'Just do it.' OCCURRENCE 2) = 'do'
SUBSTRING_REGEX ('(do) (\p{L}*' IN 'Just do it.' GROUP 2) = 'it'
```

Функция TRANSLATE_REGEX

Функция TRANSLATE_REGEX ищет в строке шаблон регулярного выражения XQuery и возвращает строку с одним или всеми вхождениями регулярного выражения, замененными строкой замены XQuery. Согласно международному стандарту JTC 1/SC 32 синтаксис функции трансляции регулярных выражений имеет следующий вид.

```
TRANSLATE_REGEX <левая круглая скобка>
   <шаблон XQuery>[ FLAG <флаг параметра XQuery>]
   IN <предметная строка регулярного выражения>
   [ WITH <строка замены регулярного выражения>]
   [ FROM <начальная позиция>]
   [ USING <единицы символьной длины>]
   [ OCCURRENCE <вхождение трансляции регулярного выражения>]
   <правая круглая скобка>
   <вхождение трансляции регулярного выражения> ::=
   <вхождение регулярного выражения>
   | ALL
```

- *<строка замены регулярного выражения>* — символьная строка, значение которой соответствует аргументу `$replacement` функции `fn:replace` XQuery (по умолчанию используется строка нулевой длины);
- *<вхождение трансляции регулярного выражения>* — это или ключевое слово `ALL`, или точное числовое значение, указывающее, какое вхождение совпадения необходимо искать (значение по умолчанию — `ALL`).

Приведем несколько примеров без строк замены.

```
TRANSLATE_REGEX ('i' IN 'Bill did sit.') = 'Bll dd st.'
TRANSLATE_REGEX ('i' IN 'Bill did sit.' OCCURRENCE ALL) = 'Bll dd st.'
TRANSLATE_REGEX ('i' IN 'Bill did sit.' FROM 5) = 'Bill dd st.'
TRANSLATE_REGEX ('i' IN 'Bill did sit.' Occurrence 2) = 'Bill dd sit.'
```

А вот несколько примеров с использованием строк замены.

```
TRANSLATE_REGEX ('i' IN 'Bill did sit.' WITH 'a') = 'Ball dad sat.'
TRANSLATE_REGEX ('i' IN 'Bill did sit.' WITH 'a' OCCURRENCE ALL) =
 'Ball dad sat.'
TRANSLATE_REGEX ('i' IN 'Bill did sit.' WITH 'a' OCCURRENCE 2) =
 'Bill dad sit.'
TRANSLATE_REGEX ('i' IN 'Bill did sit.' WITH 'a' FROM 5) =
 'Bill dad sat.'
```

Функция OVERLAY

Заменяет в строке указанную подстроку (определенную числовыми значениями исходной позиции и длины) строкой замены. Если для подстроки

указана нулевая длина, из исходной строки ничего не удаляется, а строка замены вставляется в исходную строку начиная с заданной начальной позиции.

Функция UPPER

Преобразует все символы строки в символы верхнего регистра, как показано ниже.

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.

Функция LOWER

Преобразует все символы строки в символы нижнего регистра, как показано ниже.

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.

Функция TRIM

Чтобы исключить из символьной строки ведущие или замыкающие (или одновременно и те, и другие) пробелы (или другие символы), используйте функцию `TRIM`.

Выражение	Результат
`TRIM (LEADING ' ' FROM ' ангел ')`	`'ангел '`
`TRIM (TRAILING ' ' FROM ' ангел ')`	`' ангел'`
`TRIM (BOTH ' ' FROM ' ангел ')`	`'ангел'`
`TRIM (BOTH 'а' FROM 'америка')`	`'мерик'`

Символом, удаляемым по умолчанию, для этой функции является пробел, поэтому следующий синтаксис также допустим.

```
TRIM (BOTH FROM ' ангел ')
```

В этом случае получится тот же результат, что и в третьем примере, а именно: 'ангел'.

Функции TRANSLATE и CONVERT

Берут исходную строку, составленную из символов одного набора, и преобразуют ее в строку, составленную из символов другого набора. Примером может служить преобразование символов из английского набора в армянский или из иврита во французский. Функции преобразования, выполняющие эти действия, зависят от реализации SQL. За подробностями обратитесь к документации к установленной у вас СУБД.


ЗАПОМНИ!

Если бы перевод с одного языка на другой был таким легким, как вызов SQL-функции TRANSLATE, то это можно было бы назвать магией. К сожалению, такая задача — не из простых. Все, что делает функция TRANSLATE, — это преобразует символ из одного символьного набора в соответствующий символ другого набора. Она может, например, преобразовать 'Ellas' в 'Ellas' и в то же время не сможет преобразовать 'Ellas' в 'Греция'.

Числовые функции

Числовые функции могут получать на вход данные разных типов, но возвращают всегда числовое значение. В SQL существует почти два десятка групп таких функций:

- позиция подстроки (POSITION);
- вхождения регулярного выражения (OCCURRENCES_REGEX);
- позиция регулярного выражения (POSITION_REGEX);
- извлечение подстроки (EXTRACT);
- символьная длина (CHARACTER_LENGTH, OCTET_LENGTH);
- кардинальное число (CARDINALITY, ARRAY_MAX_CARDINALITY);
- усечение массива (TRIM_ARRAY);
- абсолютное значение (ABS);
- остаток от целочисленного деления (MOD);
- геометрические функции (SIN, COS, TAN, ASIN, ACOS, ATAN, SINH, COSH, TANH);
- логарифмические функции (LOG, LOG10, LN);
- экспоненциальная функция (EXP);
- возведение в степень (POWER);
- квадратный корень (SQRT);

- округление до ближайшего целого в меньшую сторону (FLOOR);
- округление до ближайшего целого в большую сторону (CEIL, CEILING);
- определение интервального номера (WIDTH_BUCKET).

Функция POSITION

Ищет заданную целевую строку в исходной строке и возвращает позицию символа, с которого начинается целевая строка. Для символьной строки синтаксис выглядит следующим образом.

```
POSITION (целевая_строка IN исходная_строка
  [USING единицы_символьной_длины])
```

Вы можете указать *единицы_символьной_длины*, если значение этого элемента отличается от варианта CHARACTER, но такая потребность возникает в редких случаях. Если используются символы Unicode, то (в зависимости от типа данных) длина символа может быть 8, 16 или 32 бита. В случае, если символ имеет длину 16 или 32 бита, можно явно указать длину 8 битов с помощью ключевых слов USING OCTETS.

Для двоичной строки синтаксис выглядит следующим образом.

```
POSITION (целевая_строка IN исходная_строка)
```

Если значение целевой строки равно совпадающей по длине (в октетах) подстроке исходной строки, то результат будет на единицу большим, чем количество октетов, предшествующих началу первой такой подстроки. Ниже представлено несколько примеров использования функции POSITION.

Выражение	Результат
POSITION ('Б' IN 'Белый хлеб')	1
POSITION ('Бел' IN 'Белый хлеб')	1
POSITION ('хл' IN 'Белый хлеб')	7
POSITION ('пш' IN 'Белый хлеб')	0
POSITION ('' IN 'Белый хлеб')	1
POSITION ('01001001' IN '001100010100100100100110')	2

Если функция не находит целевую строку, то (как для символьных, так и для двоичных строк) возвращается значение 0. Если длина целевой строки нулевая (как в предпоследнем примере), то функция POSITION всегда возвращает

единицу. Если любой из операндов этой функции имеет значение NULL, то в результате также будет NULL.

Функция OCCURRENCES_REGEX

Возвращает количество совпадений для регулярного выражения в строке. Она имеет следующий синтаксис.

```
OCCURRENCES_REGEX <левая круглая скобка>
<шаблон XQuery>[ FLAG <флаг параметра XQuery>]
IN <предметная строка регулярного выражения>
[ FROM <начальная позиция>]
[ USING <единицы символьной длины>] <правая круглая скобка>
```

Вот несколько примеров.

```
OCCURRENCES_REGEX ( 'i' IN 'Bill did sit.' ) = 3
OCCURRENCES_REGEX ( 'i' IN 'Bill did sit.' FROM 5) = 2
OCCURRENCES_REGEX ( 'I' IN 'Bill did sit.' ) = 0
```

Функция POSITION_REGEX

Возвращает позицию начала совпадения или на единицу большее значение конца совпадения для регулярного выражения в строке. Ее синтаксис приведен ниже.

```
POSITION_REGEX <левая круглая скобка>[ <START | AFTER> ]
<шаблон XQuery>[ FLAG <флаг параметра XQuery>]
IN <предметная строка регулярного выражения>
[ FROM <начальная позиция>]
[ USING <единицы символьной длины>]
[ OCCURRENCE <вхождение регулярного выражения>]
[ GROUP <группа фиксации регулярного выражения>]
 <правая круглая скобка>
```

Приведем несколько примеров, которые помогут прояснить ситуацию.

```
POSITION_REGEX ( 'i' IN 'Bill did sit.' ) = 2
POSITION_REGEX ( START 'i' IN 'Bill did sit.' ) = 2
POSITION_REGEX ( AFTER 'i' IN 'Bill did sit.' ) = 3
POSITION_REGEX ( 'i' IN 'Bill did sit.' FROM 5) = 7
POSITION_REGEX ( 'i' IN 'Bill did sit.' OCCURRENCE 2 ) = 7
POSITION_REGEX ( 'I' IN 'Bill did sit.' ) = 0
```

Функция EXTRACT

Извлекает один заданный компонент из значения типа даты/времени или интервала. Например, следующее выражение вернет значение 08.

```
EXTRACT (MONTH FROM DATE '2013-08-20')
```

Функция CHARACTER_LENGTH

Возвращает количество символов в заданной строке. Например, следующее выражение вернет число 15.

```
CHARACTER_LENGTH ('Жареный опоссум')
```


ЗАПОМНИ!

Эта функция (как и функция SUBSTRING) не особенно полезна, если ее аргументом является литерал (такой, как 'Жареный опоссум'). Ведь вместо выражения CHARACTER_LENGTH ('Жареный опоссум') можно сразу написать число 15, и дело с концом. В использовании функции CHARACTER_LENGTH будет больше смысла, если в качестве ее аргумента задается не литерал, а выражение.

Функция OCTET_LENGTH

Практически во всех современных компьютерах для представления одного алфавитно-цифрового символа используется 8 битов, или 1 байт. В более сложных наборах символов (таких, например, как китайский) для этого используется уже 16 битов (2 байта). Функция OCTET_LENGTH подсчитывает и возвращает количество *октетов* (байтов) в строке. Если строка является битовой, то эта функция возвращает число октетов, необходимое для хранения данного числа битов. Если строка составлена из символов англоязычного набора (с одним октетом на символ), то функция OCTET_LENGTH вернет количество символов, содержащихся в строке. Если же строка состоит из символов китайского набора, то число, возвращаемое функцией, в два раза превысит количество китайских символов.

```
OCTET_LENGTH ('Beans, Lima')
```

Эта функция возвращает число 11, поскольку каждый символ помещается в октете.


ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

В некоторых наборах для разных символов используется переменное число октетов. В частности, в тех из них, которые поддерживают смешанное использование символов кандзи (японское иероглифическое письмо) и латиницы, для перехода из одного набора символов в другой используются управляющие последовательности. Например, для строки, состоящей из 30 латинских символов, потребуется 30 октетов. А если все ее 30 символов взяты из кандзи, то для нее нужно 62 октета (60 октетов плюс ведущий и замыкающий символы переключения). И наконец, если в этой строке символы латиницы и кандзи чередуются попеременно, то для нее требуется 150 октетов (поскольку для каждого символа кандзи требуется два октета,

а также по одному октету для ведущего и замыкающего символов переключения). Функция OCTET_LENGTH возвращает количество октетов, которое требуется для текущего значения заданной строки.

Функция CARDINALITY

Работает с такими коллекциями элементов, как массивы или мультимножества, каждый элемент которых является значением определенного типа. Количество содержащихся в коллекции элементов называется *кардинальным числом коллекции* (или ее *мощностью*). Рассмотрим один из примеров использования функции CARDINALITY.

```
CARDINALITY (TeamRoster)
```

Если в списке некоторой команды указано двенадцать человек, то эта функция вернет значение 12. Столбец TeamRoster таблицы TEAM может быть как массивом, так и мультимножеством (*массив* — это упорядоченная, а *мультимножество* — неупорядоченная коллекция элементов). Для списка команды, который может изменяться достаточно часто, разумнее использовать именно мультимножество.

Функция ARRAY_MAX_CARDINALITY

Функция CARDINALITY возвращает количество элементов в заданном массиве или мультимножестве, но не сообщает максимальное значение мощности, которое было назначено для данного массива, а ведь иногда это полезно знать. Поэтому в стандарт SQL:2011 была добавлена новая функция: ARRAY_MAX_CARDINALITY. Нетрудно догадаться, что она возвращает заданную максимальную мощность массива. Для мультимножества максимальная мощность не задается.

Функция TRIM_ARRAY

В то время как функция TRIM отсекает первые или последние символы в строке, функция TRIM_ARRAY исключает последние элементы массива.

Чтобы удалить последние три элемента массива TeamRoster, используйте следующий синтаксис.

```
TRIM_ARRAY (TeamRoster, 3)
```

Функция ABS

Возвращает абсолютное значение заданного числового выражения.

```
ABS (-273)
```

Результатом будет число 273.

Функция MOD

Возвращает остаток от целочисленного деления первого числового выражения на второе.

```
MOD (3, 2)
```

В данном случае функция возвращает 1 — остаток, получаемый при делении нацело числа 3 на 2.

Функция SIN

Возвращает синус числового выражения.

```
SIN (числовое выражение)
```

Функция COS

Возвращает косинус числового выражения.

```
COS (числовое выражение)
```

Функция TAN

Возвращает тангенс числового выражения.

```
TAN (числовое выражение)
```

Функция ASIN

Возвращает арксинус числового выражения.

```
ASIN (числовое выражение)
```

Функция ACOS

Возвращает арккосинус числового выражения.

```
ACOS (числовое выражение)
```

Функция ATAN

Возвращает арктангенс числового выражения.

```
ATAN (числовое выражение)
```

Функция SINH

Возвращает гиперболический синус числового выражения.

```
SINH (числовое выражение)
```

Функция COSH

Возвращает гиперболический косинус числового выражения.

```
COSH (числовое выражение)
```

Функция TANH

Возвращает гиперболический тангенс числового выражения.

```
TANH (числовое выражение)
```

Функция LOG

Возвращает логарифм числового выражения по заданному основанию.

```
LOG (основание, числовое выражение)
```

Функция LOG10

Возвращает десятичный логарифм числового выражения.

```
LOG10 (числовое выражение)
```

Функция LN

Возвращает натуральный логарифм числового выражения.

```
LN (числовое выражение)
```

Например, для функции `LN (9)` возвращаемое значение будет приблизительно равно 2,197224577. Количество знаков после запятой зависит от конкретной реализации SQL.

Функция EXP

Возводит основание натурального логарифма *e* в степень, заданную числовым выражением.

```
EXP (2)
```

В данном случае функция возвращает значение, которое приблизительно равно 7,389056. Количество знаков после запятой зависит от конкретной реализации SQL.

Функция POWER

Возводит первое числовое значение в степень, заданную вторым числовым выражением.

```
POWER (2, 8)
```

В этом примере функция вернет 256 (два в восьмой степени).

Функция SQRT

Возвращает квадратный корень числового выражения.

```
SQRT (4)
```

Эта функция возвращает 2 (квадратный корень из четырех).

Функция FLOOR

Округляет числовое значение до наибольшего целого числа, не превышающего данное значение.

```
FLOOR (3,141592)
```

Эта функция возвращает 3.

Функция CEIL (CEILING)

Округляет числовое значение до наименьшего целого числа, которое не меньше данного значения.

```
CEIL (3,141592)
```

Эта функция возвращает число 4.

Функция WIDTH_BUCKET

Используется в приложениях, выполняющих оперативную обработку данных (On-line Analytical Processing — OLAP). Получая четыре аргумента, она возвращает целое число между нулем (0) и увеличенным на единицу значением четвертого аргумента. Возвращаемое число означает, в какой по счету части всего диапазона находится первый аргумент функции. Сам диапазон определяется вторым и третьим аргументами функции. В качестве делителя диапазона на равновеликие части используется последний аргумент. Для значений, выпадающих из этого диапазона, возвращается результат, равный либо нулю (0), либо увеличенному на единицу значению четвертого аргумента.

```
WIDTH_BUCKET (PI, 0, 10, 5)
```

Предположим, что `PI` — числовое выражение со значением 3,141592. В данном примере интервал значений между нулем и 9,99999[9] (`0` и `10` — 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 представляет собой диапазон значений от 2 до 3,999999.

Функции даты/времени

В SQL существуют три функции, которые возвращают информацию о текущей дате, текущем времени или и о том и другом. Так, функция `CURRENT_DATE` возвращает текущую дату, `CURRENT_TIME` — текущее время, `CURRENT_TIMESTAMP` — текущие дату и время. Первая из этих функций не имеет аргументов, а у остальных по одному аргументу. Этот единственный аргумент

определяет точность секундной части возвращаемого функцией значения времени. (О типах даты/времени и о том, что такое точность, см в главе 2.)

Ниже приведено несколько примеров использования функций даты/времени.

Выражение	Результат
CURRENT_DATE	2012-12-31
CURRENT_TIME (1)	08:36:57.3
CURRENT_TIMESTAMP (2)	2012-12-31 08:36:57.38

Дата, возвращаемая функцией CURRENT_DATE, имеет тип DATE. Время, возвращаемое функцией CURRENT_TIME (*p*), имеет, в свою очередь, тип TIME, а значение даты и времени, возвращаемое функцией CURRENT_TIMESTAMP (*p*), — тип TIMESTAMP. Поскольку информация о дате и времени исходит от системных часов компьютера, она соответствует тому часовому поясу, в котором находится компьютер.

В некоторых приложениях значения даты и времени требуется представлять в виде символьных строк. Преобразование типов данных можно выполнять с помощью выражения CAST, которое будет описано в главе 9.

Интервальные функции

Интервальная функция ABS появилась в стандарте SQL:1999. Она подобна числовой функции ABS, но работает с интервальными данными, а не с данными числового типа. Функция ABS имеет один операнд и возвращает интервал идентичной точности, гарантируя отсутствие отрицательного значения. Рассмотрим пример:

```
ABS ( TIME '11:31:00' - TIME '12:31:00' )
```

Результат будет таким:

```
INTERVAL +'1:00:00' HOUR TO SECOND
```

Табличные функции

Табличная функция возвращает целую таблицу, а не одиночное значение. Существуют два типа таких функций: обычные и полиморфные (определены в стандарте SQL:2016).

Обычные табличные функции

Обычная табличная функция получает в качестве аргумента одну или несколько таблиц, выполняет над ними ту или иную операцию и возвращает результирующую таблицу. При создании такой функции необходимо задавать имена и типы возвращаемых ею столбцов.

Полиморфные табличные функции

Полиморфная табличная функция возвращает таблицу, тип записи которой не был объявлен при создании функции. Тип записи может зависеть от аргументов вызова функции. Полиморфные функции могут содержать обобщенные табличные параметры, не требуя предварительного объявления типа записи. Более того, тип результата может зависеть от типов записей входных таблиц. На момент выхода книги полиморфные табличные функции еще не поддерживались популярными СУБД.

Глава 9

Использование сложных выражений

В ЭТОЙ ГЛАВЕ...

- Использование условных выражений CASE
- Преобразование данных из одного типа в другой
- Ускорение ввода данных с помощью выражений с записями

В главе 2 SQL был назван *подъязыком данных*. Фактически его единственное назначение — работать с информацией, хранящейся в базе данных. SQL не обладает многими возможностями, присущими процедурным языкам, поэтому разработчикам приложений приходится объединять SQL-инструкции с кодом, написанным на процедурном языке. Эти постоянные переходы с одного языка на другой усложняют процесс разработки программ и отрицательно сказываются на их производительности.

Довольно низкий уровень производительности, вызванный ограниченными возможностями SQL, стимулирует постоянное обновление международных стандартов. Одним из относительно новых средств языка является выражение CASE, с помощью которого можно создать условную конструкцию с множеством вариантов выбора. Еще одно нововведение — выражение CAST, которое позволяет преобразовать табличную информацию из одного типа в другой. В качестве третьей новинки SQL можно назвать выражение со значением типа записи, которое дает возможность работать со списком значений там, где раньше можно было работать только с одним значением. Например, если элементами списка значений являются столбцы таблицы, то вы сможете выполнить операцию со всеми этими столбцами, используя довольно простой синтаксис.

Условные выражения CASE

В каждом полноценном компьютерном языке имеется какая-либо (и часто не одна) условная инструкция. Пожалуй, самой распространенной среди них является конструкция IF...THEN...ELSE...ENDIF. Если условие, следующее за ключевым словом IF, при вычислении дает значение True, то выполняется блок команд, который следует за ключевым словом THEN. Если же вычисленное значение условия не равно True, то выполняется блок команд, следующий за ключевым словом ELSE. О завершении конструкции свидетельствует ключевое слово ENDIF. Эта конструкция прекрасно подходит для принятия решения о выборе одного из двух вариантов. В то же время она слишком неуклюжая, если необходимо сделать выбор из множества вариантов.


ЗАПОМНИ!

В большинстве компьютерных языков существует также инструкция CASE, предназначенная для ситуаций, в которых требуется решить одну из множества задач, в зависимости от того, какому из множества значений равен результат вычисления условия.

В SQL одновременно существует и инструкция CASE, и одноименное выражение. Выражение CASE является составной частью инструкции, а не самостоятельной конструкцией. В SQL выражение CASE можно использовать практически в любом месте, допустимом для значений. Во время выполнения программы это выражение вычисляется с получением единственного значения. А инструкция CASE не вычисляет значение — она обеспечивает выполнение блока инструкций.

Выражение CASE просматривает таблицу строка за строкой, принимая значение заданного результата каждый раз, когда одно из условий в списке возвращает значение True. Если строка не удовлетворяет первому условию, проверяется второе условие. Если оно выполняется, то заданный для него результат становится значением выражения, и так далее, пока не будут обработаны все условия. Если никаких совпадений не обнаружится, то выражение получит значение NULL, после чего начнет обрабатываться следующая строка.

Выражение CASE можно использовать двумя способами.

- **Проверка условий отбора.** Выражение CASE находит в таблице такие строки, для которых выполняются заданные условия. Если для какой-либо строки результат вычисления условия оказывается равным True, то инструкция, содержащая это выражение CASE, выполняет заданное изменение в этой строке.

» **Сравнение содержимого поля таблицы с заданным значением.** Результат действия инструкции, содержащей выражение CASE, зависит от того, какое из нескольких заданных значений совпадает с содержимым поля в каждой строке таблицы.

Эти объяснения станут более понятными после того, как вы прочитаете следующие два раздела. В первом из них приводятся два примера использования выражения CASE с условиями отбора. В одном из этих примеров выполняется просмотр всей таблицы и на основе проверки заданных условий вносятся изменения в таблицу. Во втором разделе приведены два примера использования выражения CASE с проверкой значений.

Использование выражения CASE с условиями отбора

Эффективным способом использования выражения CASE является поиск строк таблицы, в которых выполняется заданное условие отбора. В таком случае синтаксис выражения CASE должен иметь следующий вид.

```
CASE
   WHEN условие_1 THEN результат_1
   WHEN условие_2 THEN результат_2
   ...
   WHEN условие_n THEN результат_n
   ELSE результат_x
END
```

Выражение CASE проверяет, является ли истинным *условие_1* в первой *оцениваемой строке* (т.е. в первой из тех строк, которые соответствуют условиям предложения WHERE, если оно задано). Если *условие_1* оказалось равным True, то выражение CASE принимает значение элемента *результат_1*. А если *условие_1* не равно True, то для той же строки проверяется *условие_2*. Если оно выполняется, то выражение CASE принимает значение элемента *результат_2* и т.д. Если же ни одно из заданных условий не выполнено, то выражение CASE принимает значение элемента *результат_x*. Предложение ELSE не является обязательным. В случае, если этого предложения нет и не выполняется ни одно из указанных условий, то выражение принимает значение NULL. После того как SQL-инструкция, содержащая выражение CASE, полностью обработает первую оцениваемую строку таблицы и выполнит соответствующее действие, она перейдет к следующей оцениваемой строке. Выполнение этой последовательности действий продолжается до тех пор, пока не закончится обработка всей таблицы.

Обновление значений на основе условия

Выражение CASE можно использовать почти в любом месте инструкции SQL, где только может находиться значение. И это открывает широкие возможности. Например, можно использовать выражение CASE в инструкции обновления UPDATE, чтобы на основе определенных условий изменять табличные значения. Рассмотрим следующий пример.

```
UPDATE FOODS
   SET RATING = CASE
                   WHEN FAT < 1
                      THEN 'очень мало жира'
                   WHEN FAT < 5
                      THEN 'мало жира'
                   WHEN FAT < 20
                      THEN 'среднее содержание жира'
                   WHEN FAT < 50
                      THEN 'высокое содержание жира'
                   ELSE 'сплошной жир'
                END ;
```

Эта инструкция проверяет по порядку условия WHEN, пока не встретится первое истинное значение, после чего оставшиеся условия игнорируются.

В табл. 8.2 было приведено содержание жира в 100 граммах некоторых продуктов питания. Таблица из базы данных, содержащая эту информацию, может также включать столбец RATING, который дает словесную оценку значению жирности. Если выполнить приведенную выше инструкцию UPDATE для таблицы FOODS из главы 8, то у спаржи будет оценка "очень мало жира", у цыплят — "мало жира", а сливочное масло попадет в категорию "сплошной жир".

Обход условий, вызывающих ошибки

Еще одним ценным применением выражения CASE является обход *исключений* — проверка условий, которые вызывают ошибки.

В некоторых компаниях новым работникам (продавцам), нанятым с условием оплаты труда на комиссионной основе, выдают аванс в счет будущих комиссионных. Рассмотрим выражение CASE, определяющее размер такого аванса, с учетом того, что со временем (при достаточном увеличении комиссионных) выплата аванса прекращается.

```
UPDATE SALES_COMP
   SET COMP = COMMISSION + CASE
                              WHEN COMMISSION > DRAW
                                 THEN 0
                              WHEN COMMISSION < DRAW
                                 THEN DRAW
                           END ;
```

Если у продавца нет комиссионных, 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 и их, и аванс, размер которого уменьшается пропорционально размеру комиссионных.

Все выражения THEN в общем выражении CASE должны быть одного и того же типа — или числовые, или символьные, или типа даты/времени. И результат выражения CASE будет иметь тот же тип.

Использование выражения CASE со значениями

Для сравнения тестового значения с другими значениями некоторой последовательности можно использовать более компактную форму выражения CASE. Эта форма весьма полезна в инструкции SELECT или UPDATE, когда в столбце таблицы содержится ограниченное число разных значений и с каждым из них нужно связать соответствующее значение результата. В этом случае выражение CASE будет иметь следующий синтаксис.

```
CASE тестовое_значение
   WHEN значение_1 THEN результат_1
   WHEN значение_2 THEN результат_2
   ...
   WHEN значение_n THEN результат_n
   ELSE результат_x
END
```

Если *тестовое_значение* и *значение_1* равны, то выражение получает значение *результат_1*. А если *тестовое_значение* и *значение_1* не равны, но равны *тестовое_значение* и *значение_2*, то выражение получает значение *результат_2*. Все значения, представленные для сравнения, проверяются по очереди, сверху вниз, по направлению к элементу *значение_n*, пока не будет найдено совпадение с тестовым значением. Если же такое значение (совпадающее с тестовым) найдено не будет, то выражение получит значение *результат_x*. Опять-таки, если необязательное предложение ELSE отсутствует и ни одно из значений последовательности, представленных для сравнения, не равно тестовому, то выражение получает значение NULL.

Чтобы понять, как работает конструкция CASE со значениями, рассмотрим пример с таблицей, которая содержит фамилии и звания офицеров и генералов. Нам нужно получить их список, в котором перед фамилиями стояли бы аббревиатуры их званий. Для сравнения будут использоваться такие звания: генерал, полковник, подполковник, майор, капитан, старший лейтенант и лейтенант. Список создается с помощью следующей инструкции.

```
SELECT CASE RANK
          WHEN 'генерал'            THEN 'ген.'
          WHEN 'полковник'          THEN 'полк.'
          WHEN 'подполковник'       THEN 'подп.'
          WHEN 'майор'              THEN 'м-р'
          WHEN 'капитан'            THEN 'кап.'
          WHEN 'старший лейтенант'  THEN 'ст. л-т'
          WHEN 'лейтенант'          THEN 'л-т'
          ELSE NULL
       END,
          LAST_NAME
       FROM OFFICERS ;
```

Результат должен быть примерно таким.

```
кап.  Иванов
полк. Дударев
ген.  Макаев
м-р   Красновский
      Цибаев
```

Если имеющееся в столбце звание (например, адмирал) отсутствует в выражении CASE, то при выполнении приведенной выше инструкции SELECT предложение ELSE не присвоит такому военнослужащему звания вообще (в нашем примере без звания остался Цибаев).

Вот еще пример. Предположим, капитан Иванов повышен в звании и становится майором. Требуется внести соответствующие изменения в базу данных OFFICERS (офицеры). Предположим, переменная officer_last_name (фамилия офицера) содержит значение 'Иванов', а переменная new_rank (новое звание) — значение 4, которое, согласно следующей таблице, соответствует новому званию Иванова.

new_rank	Звание
1	генерал
2	полковник
3	подполковник
4	майор
5	капитан
6	старший лейтенант
7	лейтенант
8	NULL

Теперь данные о повышении можно ввести с помощью следующей инструкции SQL.

```
UPDATE OFFICERS
   SET RANK = CASE :new_rank
                 WHEN 1 THEN 'генерал'
                 WHEN 2 THEN 'полковник'
                 WHEN 3 THEN 'подполковник'
                 WHEN 4 THEN 'майор'
                 WHEN 5 THEN 'капитан'
                 WHEN 6 THEN 'старший лейтенант'
                 WHEN 7 THEN 'лейтенант'
                 WHEN 8 THEN NULL
              END
   WHERE LAST_NAME = :officer_last_name ;
```

Для выражения CASE со значениями существует еще один синтаксис.

```
CASE
   WHEN тестовое_значение = значение_1 THEN результат_1
   WHEN тестовое_значение = значение_2 THEN результат_2
   ...
   WHEN тестовое_значение = значение_n THEN результат_n
   ELSE результат_x
END
```

Специальное выражение CASE — NULLIF

Мир изменяется постоянно — это его основное свойство. Предметы переходят из одного состояния в другое, и иногда вам кажется, что вы что-то знаете, но в результате выясняется, что это далеко не так. Классическая термодинамика, как и современная теория хаоса, утверждает, что системам свойственно переходить из хорошо известного, упорядоченного состояния в состояние хаоса, которое никто не может предсказать. Если кто-нибудь видел состояние комнаты подростка всего лишь через неделю после проведения генеральной уборки, то он легко согласится с правильностью этой теории.

Поля таблиц в базах данных хранят значения, соответствующие известной информации. Если же значение поля неизвестно, в него, как правило, записывают пустое значение (NULL). SQL позволяет с помощью выражения CASE заменять определенное значение табличного поля пустым. И тогда специальное значение NULL будет означать, что информация, содержащаяся в этом поле, вам больше не известна. Рассмотрим следующий пример.

Представьте, что вы владеете небольшой авиакомпанией, которая выполняет рейсы между Ростовом и Хабаровском. До недавних пор во время некоторых рейсов делалась промежуточная посадка в аэропорту Екатеринбурга

для дозаправки. Но в какой-то момент вы лишились разрешения на посадку в Екатеринбурге, и теперь дозаправку приходится проводить в одном из двух аэропортов: или Челябинска, или Перми. И теперь вам не известно, во время какого рейса в каком из аэропортов будут садиться самолеты для дозаправки, ясно лишь, что не в Екатеринбурге. В вашей базе данных есть таблица FLIGHT, в которой хранится важная информация о каждом из рейсов вашей авиакомпании (данные о посадке для дозаправки хранятся в столбце RefuelStop). Эту таблицу нужно обновить, чтобы удалить из нее все упоминания о Екатеринбурге. Один из способов осуществления такой операции показан в следующем примере.

```
UPDATE FLIGHT
   SET RefuelStop = CASE
                       WHEN RefuelStop = 'Екатеринбург'
                          THEN NULL
                       ELSE RefuelStop
                    END ;
```

СОВЕТ

Поскольку ситуации, когда нужно заменить известное значение пустым (NULL), случаются часто, для решения таких задач в SQL предусмотрен специальный упрощенный синтаксис. Вот как выглядит предыдущий пример, переписанный с использованием этого синтаксиса.

```
UPDATE FLIGHT
   SET RefuelStop = NULLIF(RefuelStop, 'Екатеринбург') ;
```

Это выражение можно перевести следующим образом: "Обновить таблицу FLIGHT путем замены в столбце RefuelStop значения 'Екатеринбург' значением NULL. Остальные значения заменять не надо".

В синтаксисе предложения NULLIF еще больше пользы, если требуется преобразовать данные, собранные для обработки программами, написанными на таких языках, как C++ или Java. В стандартных языках программирования значение NULL не применяется, поэтому очень часто для выражения понятия "неизвестно" или "не применимо" используются специальные значения. Например, значение "неизвестно" в поле зарплаты (Salary) может быть представлено отрицательным числом –1, а в поле кода должности — символьной строкой '***'. Если требуется в реляционной базе данных представить состояния "неизвестно" или "не применимо" с помощью значения NULL, то эти специальные значения (–1 и '***') придется преобразовать в пустые. 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

Выражение COALESCE, как и NULLIF, является сокращенной формой выражения CASE. Оно работает со списком значений, которые могут быть определенными или пустыми.

- **Если в списке только одно из значений не является пустым**, то оно и становится значением выражения COALESCE.
- **Если список содержит не одно, а несколько непустых значений**, то значением выражения становится первое из них.
- **Если все значения списка пустые**, то выражение COALESCE равно NULL.

Аналогичное выражение CASE имело бы следующий вид.

```
CASE
   WHEN значение_1 IS NOT NULL
      THEN значение_1
   WHEN значение_2 IS NOT NULL
      THEN значение_2
   ...
   WHEN значение_n IS NOT NULL
      THEN значение_n
   ELSE NULL
END
```

Соответствующий упрощенный синтаксис выражения COALESCE выглядит так:

```
COALESCE(значение_1, значение_2, ..., значение_n)
```

Возможно, вам придется использовать выражение COALESCE после выполнения операции внешнего объединения OUTER JOIN (о которой мы поговорим в главе 11). В этом случае использование выражения COALESCE позволит существенно сократить объем кода.

Преобразование типов данных с помощью выражения CAST

В главе 2 описывались различные типы данных, поддерживаемые в SQL. В идеальном случае каждый столбец таблицы должен иметь подходящий тип данных. Однако на практике не всегда ясно, каким же он должен быть. Предположим, что, определяя в базе данных таблицу, вы присваиваете ее столбцу тип данных, который замечательно подходит для текущего приложения. Однако позднее вам, возможно, потребуется расширить область действия вашего приложения или написать полностью новое приложение, в котором данные используются по-другому. В таком случае может потребоваться тип данных, отличный от выбранного вами ранее.

Иногда нужно 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 строк, а в другой — в виде значений типа `DATE`. Даже если в обоих столбцах содержится один и тот же вид информации (например, даты), различие в типах данных не позволяет провести сравнение. В стандартах SQL-86 и SQL-89 несовместимость типов данных представляла большую проблему. Однако с выходом стандарта SQL-92 появилась довольно простая возможность решить ее с помощью выражения `CAST`.

Выражение `CAST` преобразует табличные данные или переменные из одного типа в другой. После такого преобразования становится возможным выполнение многих операций.


ЗАПОМНИ!

Перед использованием выражения `CAST` следует ознакомиться с некоторыми его ограничениями. Нельзя бездумно преобразовывать данные одного типа в любой другой. Преобразуемые данные должны быть совместимы с новым типом. Например, с помощью выражения `CAST` можно преобразовать символьную строку `'2007-04-26'` типа `CHAR(10)` в тип `DATE`. В то же время символьную строку `'гип-попотам'`, также имеющую тип `CHAR(10)`, преобразовать с помощью выражения `CAST` в тип `DATE` уже невозможно. Нельзя выполнить преобразование значений из типа `INTEGER` в тип `SMALLINT`, если значение типа `INTEGER` превышает максимально допустимое значение, определенное для типа `SMALLINT`.

Элемент данных любого из символьных типов можно преобразовать в любой другой тип (например, числовой или тип даты) при условии, что значение этого элемента имеет форму литерала нового типа. И наоборот, элемент

данных любого типа можно преобразовать в любой из символьных типов при условии, что значение этого элемента имеет форму литерала исходного типа.

Другие возможные преобразования типов перечислены ниже.

- Любой числовой тип — в любой другой числовой. При преобразовании в тип с меньшей дробной частью система округляет или усекает результат.
- Любой точный числовой тип — в интервал, состоящий из одного компонента, такой, например, как `INTERVAL DAY` или `INTERVAL SECOND`.
- Любой тип `DATE` — в тип `TIMESTAMP`. Часть значения типа `TIMESTAMP`, предназначенная для отображения времени, заполняется нулями.
- Любой тип `TIME` — в тип `TIME` с другим размером дробной части для секунд или в тип `TIMESTAMP`. Часть значения типа `TIMESTAMP`, предназначенная для отображения даты, заполняется значением текущей даты.
- Любой тип `TIMESTAMP` — в тип `DATE`, `TIME` или `TIMESTAMP` с другим размером дробной части для секунд.
- Любой тип `INTERVAL` категории "год-месяц" — в точный числовой тип или тип `INTERVAL` категории "год-месяц" с другим видом представления.
- Любой тип `INTERVAL` категории "день-время" — в точный числовой тип или тип `INTERVAL` категории "день-время" с другим видом представления.

Использование выражения CAST в SQL

Предположим, вы работаете в торговой компании, которая собирает данные о потенциальных и уже нанятых сотрудниках. Данные о потенциальных сотрудниках содержатся в таблице `PROSPECT`. Они включают уникальный идентификатор, который хранится в поле `SSN` типа `CHAR(9)`. Данные об уже нанятых сотрудниках находятся в другой таблице, `EMPLOYEE`, и различаются они также по своему идентификатору, имеющему тип `INTEGER`. Теперь представьте, что вам нужно получить список только тех людей, данные о которых содержатся в обеих таблицах. Чтобы решить эту задачу, используйте выражение `CAST`.

```
SELECT * FROM EMPLOYEE
 WHERE EMPLOYEE.SSN =
       CAST(PROSPECT.SSN AS INTEGER) ;
```

Использование выражения CAST при взаимодействии SQL и языка приложения

Главное назначение выражения CAST — работать с такими типами данных, которые поддерживаются в SQL, но отсутствуют в языках программирования высокого уровня. Ниже приведены примеры таких типов.

- FORTRAN и Pascal не имеют типов данных DECIMAL и NUMERIC, а в SQL они есть.
- Стандартный COBOL не содержит типов данных FLOAT и REAL, а в SQL они есть.
- Ни в каком языке, кроме SQL, нет типа данных DATETIME.

Предположим, для доступа к таблицам, в которых есть столбцы с типом данных DECIMAL(5, 3), вы собираетесь использовать язык FORTRAN или Pascal. При этом необходимо избежать вероятных неточностей перевода значений столбцов в тип REAL, используемый в указанных двух языках. Эту задачу можно решить с помощью выражения CAST для перевода данных в переменные приложения строкового типа и обратно. Например, вы извлекли *числовое* значение зарплаты 198.37 и преобразовали его в *строку* '0000198.37' типа CHAR(10). При повышении зарплаты работника до 203,74 необходимо поместить это число в строку '0000203.74' типа CHAR(10). Для этого сначала воспользуемся выражением CAST и преобразуем данные SQL из типа DECIMAL(5, 3) в тип CHAR(10) для сотрудника с идентификатором, хранящимся в переменной приложения :emp_id_var.

```
SELECT CAST(Salary AS CHAR(10)) INTO :salary_var
  FROM EMP
 WHERE EmpID = :emp_id_var ;
```

Затем приложение FORTRAN или Pascal проверяет значение символьной строки в переменной :salary_var и присваивает этой переменной новое значение ('0000203.74'), после чего с помощью следующего SQL-кода 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 неудобно, но для выполнения нужных операций можно написать набор специальных процедур. И тогда вы сможете извлекать и обновлять любые SQL-данные из приложения, написанного на каком-то процедурном языке, а также получать и устанавливать точные значения.

Выражение CAST целесообразно использовать для "двунаправленного" преобразования типов данных процедурного языка в типы, определенные для работы с базой данных (т.е. для взаимодействия разных языков программирования), а не для преобразования типов в самой базе данных.

Выражения с записями

Стандартами SQL-86 и SQL-89 было предписано, что большинство операций должно выполняться с одиночным значением или с одиночным столбцом в табличной строке. Чтобы работать со множеством значений, необходимо было с помощью *логических операторов* строить сложные выражения (о логических операторах вы узнаете в главе 10).

С выходом стандарта SQL-92 в SQL появились *выражения с записями*, которые работают не с одиночными значениями, а со списками значений или столбцов. Выражение с записью представляет собой список выражений, заключенных в круглые скобки и разделенных запятыми. Выражение такого типа позволяет работать сразу с целой строкой или с некоторым ее подмножеством.

В главе 6 вы узнали, как с помощью инструкции INSERT добавить в таблицу новую строку. Для этого как раз и используется выражение с записью.

Рассмотрим следующий пример. Данные о сыре "чеддер" вводятся в поля FOODNAME (название продукта питания), CALORIES (калории), PROTEIN (белки), FAT (жиры), CARBOHYDRATE (углеводы) таблицы FOODS.

```
INSERT INTO FOODS
        (FOODNAME, CALORIES, PROTEIN, FAT, CARBOHYDRATE)
    VALUES
        ('Сыр чеддер', 398, 25, 32.2, 2.1) ;
```

В этом примере выражение ('Сыр чеддер', 398, 25, 32.2, 2.1) содержит значение типа записи. При использовании в инструкции INSERT выражение с записью может содержать пустые значения (NULL) и значения по умолчанию (DEFAULT). (Под значением по умолчанию понимается значение, которое подставляется в столбец, если не вводится ничего другого в явном виде.) В следующей строке представлено допустимое выражение с записью:

```
('Сыр чеддер', 398, NULL, 32.2, DEFAULT)
```

В таблицу можно добавить сразу несколько строк, вставив в предложение VALUES несколько выражений с записями.

```
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 с записями иногда позволяют сократить объем кода, вводимого вручную. Предположим, у вас есть две таблицы с данными о продуктах питания, одна из которых составлена на английском языке, а другая — на испанском. Требуется найти те строки первой таблицы, которые соответствуют строкам второй. Если не использовать выражения с записями, то, возможно, придется написать такой код.

```
SELECT * FROM FOODS
 WHERE FOODS.CALORIES = COMIDA.CALORIA
   AND FOODS.PROTEIN = COMIDA.PROTEINAS
   AND FOODS.FAT = COMIDA.GRASAS
   AND FOODS.CARBOHYDRATE = COMIDA.CARBOHIDRATO ;
```

А вот код, выполняющий те же действия, но составленный с помощью выражения с записями.

```
SELECT * FROM FOODS
 WHERE (FOODS.CALORIES, FOODS.PROTEIN, FOODS.FAT,
        FOODS.CARBOHYDRATE)
     =
       (COMIDA.CALORIA, COMIDA.PROTEINAS, COMIDA.GRASAS,
        COMIDA.CARBOHIDRATO) ;
```


СОВЕТ

Этот пример не демонстрирует слишком большую экономию кода. Выигрыш станет заметнее, если число сравниваемых столбцов будет больше. В случаях "граничной выгоды", как в приведенном примере, лучше придерживаться старого синтаксиса, так как он более понятен.

Следует отметить, что использование выражения с записью имеет дополнительное преимущество перед его аналогом с развернутым синтаксисом. Дело в том, что это выражение работает намного быстрее. В принципе, достаточно "умная" реализация могла бы выявлять такой развернутый синтаксис, анализировать его и выполнять в виде выражения с записью. Но на практике на такую хитроумную оптимизацию пока что не способна ни одна из существующих СУБД.

Глава 10

Выбор нужных данных

В ЭТОЙ ГЛАВЕ...

- Выбор таблиц для извлечения данных
- Отделение нужных строк от остальных
- Создание эффективных предложений `WHERE`
- Работа с пустыми значениями
- Создание составных выражений с логическими операторами
- Группирование результатов запроса по столбцу
- Упорядочение результатов запроса
- Обработка связанных строк

Основное назначение СУБД — хранение данных и обеспечение удобного доступа к ним. В хранении данных нет ничего особенного, ту же работу может выполнять и картотека. Что действительно сложно, так это обеспечить удобный доступ к данным. Для этого не обойтись без инструмента отбора небольшого (как правило) объема нужных данных из множества ненужных.

SQL позволяет использовать некоторые характеристики самих данных для определения, представляет ли конкретная строка интерес для вас. Инструкции `SELECT`, `DELETE` и `UPDATE` сообщают *ядру* базы данных (т.е. той части СУБД, которая непосредственно взаимодействует с данными), какие именно строки

нужно извлекать, удалять или обновлять. Для улучшения качества получаемого результата в эти инструкции добавляются уточняющие предложения.

Уточняющие предложения

В SQL определены следующие уточняющие предложения: FROM, WHERE, HAVING, GROUP BY и ORDER BY. Предложение FROM сообщает ядру базы данных, с какой таблицей (или таблицами) будет работать инструкция. Предложения WHERE и HAVING задают характеристики данных, определяющие, включать или нет конкретные строки в текущую операцию. И наконец, предложения GROUP BY и ORDER BY содержат указания, каким образом отображать строки, извлеченные из базы данных. Основные сведения об уточняющих предложениях приведены в табл. 10.1.

Таблица 10.1. Уточняющие предложения и их назначение

Уточняющее предложение	Назначение
FROM	Указывает, из каких таблиц извлекать данные
WHERE	Отсеивает строки, которые не соответствуют условию отбора
GROUP BY	Группирует строки в соответствии со значениями в столбцах группировки
HAVING	Отсеивает группы, которые не соответствуют условию отбора
ORDER BY	Сортирует результаты предшествующих предложений для вывода окончательного результата

ЗАПОМНИ!

Если используется не одно, а несколько предложений, то располагаться они должны в следующем порядке.

```
SELECT список_столбцов
  FROM список_таблиц
  [WHERE условие_отбора]
  [GROUP BY столбец_группировки]
  [HAVING условие_отбора]
  [ORDER BY условие_сортировки] ;
```

О каждом из уточняющих предложений необходимо знать следующее.

- Предложение WHERE — это фильтр, который отбирает строки, удовлетворяющие условию, и отсеивает все остальные.
- Предложение GROUP BY создает группы из строк, отобранных с помощью предложения WHERE, в соответствии со значениями заданных столбцов группировки.
- Предложение HAVING — это еще один фильтр, обрабатывающий группы, сформированные предложением GROUP BY. Он отбирает те из них, которые удовлетворяют условию, отбрасывая все остальные.
- Предложение ORDER BY упорядочивает все данные, которые остались после обработки таблицы (или таблиц) предыдущими предложениями.

СОВЕТ

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, обрабатывает этот результат и передает то, что получилось, следующему предложению. Если этот порядок выполнения предложений представить в виде функциональной записи, то он будет выглядеть следующим образом:

```
SELECT(HAVING(GROUP BY(WHERE(FROM...))))
```

Предложение ORDER BY выполняется уже после отбора данных предложением SELECT. Оно может обращаться только к тем столбцам, которые перечислены в списке, указанном в предложении SELECT. К другим же столбцам из таблиц, перечисленных в предложении FROM, предложение ORDER BY обращаться не может.

Предложение FROM

Предложение FROM легко понять, если в нем указана только одна таблица, как в следующем примере.

```
SELECT * FROM SALES ;
```

Эта инструкция возвращает все данные, находящиеся во всех строках всех столбцов таблицы SALES. Но в предложении FROM можно указать не одну, а несколько таблиц.

```
SELECT *
  FROM CUSTOMER, SALES ;
```

Эта инструкция создает виртуальную таблицу, в которой данные из таблицы CUSTOMER объединены с данными из таблицы SALES. (О виртуальных таблицах см. в главе 6.) Для формирования такой таблицы каждая строка таблицы CUSTOMER объединяется с каждой строкой таблицы SALES. Следовательно, в создаваемой таким способом новой виртуальной таблице количество строк будет равно количеству строк первой таблицы, умноженному на количество строк второй. И если в таблице CUSTOMER десять строк, а в SALES — сто, то в новой таблице их будет тысяча.


СОВЕТ

Такая операция называется *декартовым произведением* двух исходных таблиц. Декартово произведение представляет собой разновидность операции объединения (JOIN). Операции объединения мы подробно рассмотрим в главе 11.

Во многих приложениях большинство строк, образованных в результате применения к двум таблицам декартова произведения, не имеет никакого смысла. Это справедливо и для нашего примера создания виртуальной таблицы из таблиц CUSTOMER и SALES. В данном случае интерес представляют только те строки, в которых значение CustomerID из таблицы CUSTOMER равно значению CustomerID из таблицы SALES. Все остальные строки можно отсеять с помощью предложения WHERE.

Предложение WHERE

В предыдущих главах предложение WHERE использовалось множество раз, причем без объяснений, потому что его смысл и способ применения были очевидны. Любая инструкция (такая, как SELECT, DELETE или UPDATE) выполняет операцию только с теми табличными строками, для которых заданное в предложении WHERE условие оказывается истинным. Предложение WHERE имеет следующий синтаксис.

```
SELECT список_столбцов
  FROM имя_таблицы
 WHERE условие;

DELETE FROM имя_таблицы
 WHERE условие;

UPDATE имя_таблицы
   SET столбец_1=значение_1, столбец_2=значение_2, ...,
       столбец_n=значение_n
 WHERE условие;
```

Условие в предложении WHERE может быть или простым, или сколь угодно сложным. Несколько простых условий можно объединить в одно сложное. Для этого используются логические операторы AND, OR и NOT. Мы еще вернемся к ним позже.

Вот несколько типичных примеров предложений WHERE.

```
WHERE CUSTOMER.CustomerID = 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, определяемые в предложении WHERE, называются предикатами. *Предикат* — это выражение, содержащее определенное утверждение о значениях, которое может быть истинным или ложным.

Например, предикат FOODS.Calories < 219 является истинным, если значение столбца FOODS.Calories в текущей строке меньше 219. Если утверждение, заданное предикатом, истинно, то условие считается выполненным. Утверждение может быть истинным (т.е. равным True), ложным (равным False) или неопределенным (NULL). Неопределенное значение возникает тогда, когда в утверждении какие-либо элементы содержат NULL. Наиболее распространенными являются предикаты сравнения (=, <, >, <>, <= и >=), но в SQL существуют и другие предикаты, которые значительно расширяют возможности отбора нужных данных. Список предикатов фильтрации приведен ниже:

- 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 типичное использование предикатов, в которых одно значение сравнивается с другим.

Для каждой строки, в которой результат сравнения равен `True` (что означает выполнение условия предложения `WHERE`), выполняется заданная операция (`SELECT`, `UPDATE`, `DELETE` и др.). Строки, в которых в результате сравнения получается значение `False`, пропускаются. Для примера рассмотрим следующую инструкцию SQL.

```
SELECT * FROM FOODS
 WHERE Calories < 219 ;
```

Эта инструкция отображает все строки таблицы `FOODS`, в которых значение, хранящееся в столбце `Calories`, меньше числа 219.

В табл. 10.2 приведены шесть предикатов сравнения.

Таблица 10.2. Предикаты сравнения SQL

Сравнение	Символ
Равно	=
Не равно	<>
Меньше	<
Меньше или равно	<=
Больше	>
Больше или равно	>=

Предикат `BETWEEN`

Иногда нужно выбрать строку, в которой значение определенного столбца попадает в заданный диапазон. Такой отбор строк позволяют сделать предикаты сравнения. Можно, например, составить предложение `WHERE` для отбора всех строк таблицы `FOODS`, в которых значение, хранящееся в столбце `CALORIES`, больше 100 и меньше 300.

```
WHERE FOODS.Calories > 100 AND FOODS.Calories < 300
```

Такое сравнение не позволит включить продукты питания с числом калорий, в точности равным 100 или 300, — в выборку попадут только те из них, количество калорий в которых находится *в промежутке между* этими числами. Чтобы в выборку попали и эти два граничных значения, можно использовать такую инструкцию:

```
WHERE FOODS.Calories >= 100 AND FOODS.Calories <= 300
```

Альтернативный способ определения диапазона, включающего границы, — использовать предикат BETWEEN.

```
WHERE FOODS.Calories BETWEEN 100 AND 300
```


СОВЕТ

Это предложение с функциональной точки зрения идентично предыдущему примеру. Как видите, эта формулировка короче и более наглядна, чем та, в которой используются два предиката сравнения, соединенные логическим оператором AND.


ВНИМАНИЕ!

Ключевое слово BETWEEN может привести к путанице, потому что из текста не ясно, включены ли в предложение границы диапазона. А ведь они включены. Кроме того, необходимо помнить, что первое граничное значение не должно быть больше второго. Если, например, в столбце FOODS.Calories содержится значение 200, то следующее предложение вернет значение True:

```
WHERE FOODS.Calories BETWEEN 100 AND 300
```

А вот следующее предложение, казалось бы, эквивалентное предыдущему примеру, в реальности возвращает противоположный результат (False):

```
WHERE FOODS.Calories BETWEEN 300 AND 100
```


ЗАПОМНИ!

Если вы используете предикат BETWEEN, не забывайте, что первая граница диапазона обязательно должна быть меньше или равна второй.

Предикат BETWEEN можно использовать со следующими типами данных: символьными, битовыми, даты/времени, а также с числовыми. Вам могут встретиться примеры, подобные следующему.

```
SELECT FirstName, LastName
  FROM CUSTOMER
 WHERE CUSTOMER.LastName BETWEEN 'A' AND 'Мяяя' ;
```

При его выполнении возвращаются данные обо всех покупателях, фамилии которых (вернее, их первые буквы) относятся к первой половине алфавита.

Предикаты IN и NOT IN

Предикаты IN и NOT IN используются для работы с любыми указанными значениями из наперед заданного набора значений (например, списка штатов США). Предположим, у вас есть таблица, в которой перечислены поставщики товаров, регулярно закупаемых вашей компанией. Вам нужно узнать телефонные номера тех поставщиков, которые находятся в северо-западной части

Тихоокеанского побережья Америки. Эти номера можно найти с помощью предикатов сравнения, как показано в следующем примере.

```
SELECT Company, Phone
  FROM SUPPLIER
 WHERE State = 'OR' OR State = 'WA' OR State = 'ID' ;
```

В качестве альтернативы для выполнения той же задачи можно использовать предикат IN.

```
SELECT Company, Phone
  FROM SUPPLIER
 WHERE State IN ('OR', 'WA', 'ID') ;
```

Это чуть более компактная форма записи, чем та, в которой используются предикаты сравнения и логические операторы OR. Кроме того, используя предикат IN, вы уже не спутаете оператор OR с аббревиатурой штата Орегон.

Таким же образом работает и второй вариант этого предиката — NOT IN. Предположим, 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, представительства которых находятся за пределами этих трех штатов. В таком случае можно использовать следующий запрос.

```
SELECT Company, Phone
  FROM SUPPLIER
 WHERE State NOT IN ('CA', 'AZ', 'NM') ;
```

Предикаты IN и NOT IN позволяют избежать ввода лишнего кода. Впрочем, примеры, приведенные в этом разделе, не демонстрируют такой уж большой экономии.

СОВЕТ

Несмотря на то что использование предиката IN не дает значительных преимуществ в объеме кода, у вас все равно есть веская причина применять именно его, а не предикаты сравнения. Ваша СУБД, скорее всего, поддерживает оба этих метода по-разному, и один из них может работать значительно быстрее другого. Эти два метода включения объектов в группу (или исключения из нее) можно проверить на практике, а затем использовать тот из них, который покажет самый быстрый результат. Но если ваша СУБД оснащена хорошим оптимизатором, то, независимо от используемых вами предикатов, будет выбран наиболее эффективный метод и без вашего участия.

Предикат IN представляет ценность и в другой области. Являясь частью подзапроса, он позволяет извлечь нужную информацию из двух таблиц в случае, когда ее не удается получить из одной. Подробнее о подзапросах мы

поговорим в главе 12, а сейчас рассмотрим пример использования в них предиката IN.

Предположим, вам нужно вывести имена всех клиентов, кто за последние 30 дней купил товар марки F-35. Имена и фамилии покупателей находятся в таблице CUSTOMER, а данные о покупках — в таблице TRANSACT. Для получения такой информации можно использовать следующий запрос.

```
SELECT FirstName, LastName
  FROM CUSTOMER
 WHERE CustomerID IN
       (SELECT CustomerID
          FROM TRANSACT
         WHERE ProductID = 'F-35'
           AND TransDate >= (CurrentDate - 30)) ;
```

Внутренняя инструкция SELECT, работающая с таблицей TRANSACT, вложена во внешнюю, работающую с таблицей CUSTOMER. Внутренняя инструкция ищет в столбце CustomerID идентификаторы всех тех, кто за последние 30 дней купил товар марки F-35, а внешняя инструкция выводит имена и фамилии всех покупателей, номера которых получены с помощью внутренней инструкции SELECT.

Предикаты LIKE и NOT LIKE

Предикат LIKE используют для сравнения двух строк на предмет частичного совпадения. К частичному совпадению прибегают в случае, если неизвестна точная форма искомой строки. Предикат LIKE позволяет извлечь из таблицы множество строк, содержащих в одном из столбцов таблицы сходные символьные значения.

Для идентификации частичных совпадений в SQL используются два метасимвола. Знак процента (%) означает строку, состоящую из любого (даже нулевого) количества символов. Знак подчеркивания (_) означает любой одиночный символ. Несколько примеров использования предиката LIKE приведены в табл. 10.3.

Таблица 10.3. Примеры использования предиката LIKE

Выражение	Возвращаемое значение
WHERE WORD LIKE 'город%'	город
	городок
	городовой
	городничий

Окончание табл. 10.3

Выражение	Возвращаемое значение
`WHERE WORD LIKE '%мир%'`	мировая война
	Казимир Пруткевич
`WHERE WORD LIKE 'б_р_'`	боря
	буря
	бура
	баро

Предикат `NOT LIKE` позволяет получить все строки, которые не имеют частичного совпадения. При этом также можно использовать метасимволы, как в следующем примере:

```
WHERE Phone NOT LIKE '503%'
```

В этом примере будут возвращены все строки таблицы, в которых телефонный номер, содержащийся в столбце `Phone`, не начинается с 503.

СОВЕТ

Иногда требуется выполнить поиск строк, содержащих символы процента или подчеркивания. В таком случае нужно, чтобы SQL интерпретировал знак процента именно как знак процента, а не как метасимвол. Это реализуемо, если перед символом, который при поиске должен восприниматься буквально, поставить управляющий символ. В качестве такового можно назначить любой символ, который отсутствует в проверяемой строке, как показано в следующем примере.

```
SELECT Quote
  FROM BARTLETTS
 WHERE Quote LIKE '20#%'
ESCAPE '#' ;
```

Символ `%` превращается в обычный благодаря стоящему перед ним символу `#`. Точно таким же способом можно отключить и символ подчеркивания, а также сам управляющий символ. Например, приведенный выше запрос может найти фразу `"20% продавцов обеспечивают 80% продаж"`. Запрос найдет также строку `"20%"`.

Предикат SIMILAR

В стандарт SQL:1999 был введен предикат SIMILAR, позволявший находить частичное совпадение более эффективно, чем предикат LIKE. Впрочем, в стандарте SQL:2011 он был признан устаревшим. Это означает, что его не следует задействовать в программных проектах, так как со временем он будет исключен из стандарта. Существующие программы, в которых он используется, следует переписать.

Предикат NULL

С помощью предиката NULL выполняется поиск всех строк, в которых заданный столбец содержит пустое значение. Именно такие значения содержал столбец Carbohydrate (углеводы) в нескольких строках таблицы FOODS (продукты питания), как указывалось в главе 8. Названия продуктов из этих строк можно получить с помощью такой инструкции.

```
SELECT (Food)
  FROM FOODS
 WHERE Carbohydrate IS NULL ;
```

Этот запрос вернет следующие значения.

```
Гамбургер с нежирной говядиной
Нежное мясо цыплят
Жареный опоссум
Свиной окорок
```

Нетрудно предположить, что, если перед этим предикатом вставить ключевое слово NOT, можно получить 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.

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 обоих столбцах, Carbohydrate и Protein, являются пустыми. Из этого можно сделать несколько выводов:

- выражение Carbohydrate IS NULL истинно;
- выражение Protein IS NULL истинно;
- выражение Carbohydrate IS NULL AND Protein IS NULL истинно;

- истинность выражения `Carbohydrate = Protein` не может быть определена;
- сравнение `Carbohydrate = NULL` недопустимо.

Использовать ключевое слово `NULL` в сравнениях бессмысленно, так как результат всегда будет неопределенным.

Почему же сравнение `Carbohydrate = Protein` дает неопределенный результат несмотря на то, что `Carbohydrate` и `Protein` имеют одно и то же пустое значение? Дело в том, что пустое значение означает ответ "я не знаю". Вы не знаете, каким должно быть значение в столбце `Carbohydrate`, и каким — в столбце `Protein`. Следовательно, вам неизвестно, являются ли эти значения одинаковыми. Возможно, для столбца `Carbohydrate` следует ввести число 37, а для столбца `Protein` — 14, а может быть, в каждом из них должно находиться число 93. Если вам неизвестно количество углеводов и белков, то нельзя сказать, являются ли эти величины одинаковыми.

Предикаты ALL, SOME и ANY

Несколько тысячелетий назад великий греческий философ Аристотель сформулировал систему логики, которая легла в основу Западной философии. Ее суть: следует начать с набора предпосылок, о которых известно, что они истинные, затем применить к ним допустимые операции и вывести, таким образом, новые истины. Вот пример такой процедуры.

> *Предпосылка 1.* Все греки — люди.
>
> *Предпосылка 2.* Все люди смертны.
>
> *Заключение.* Все греки смертны.

А вот еще пример.

> *Предпосылка 1.* Некоторые греки — женщины.
>
> *Предпосылка 2.* Все женщины — люди.
>
> *Заключение.* Некоторые греки — люди.

Еще один способ выражения идеи второго примера состоит в следующем.

> Если какие-то греки — женщины и все женщины — люди, то некоторые греки — люди.

В первом примере в обеих предпосылках используется *квантор всеобщности* `ALL` (все), который позволяет сделать в итоге разумный вывод обо всех греках. Во втором примере в одной из предпосылок используется *квантор существования* `SOME` (некоторые), который позволяет, применив метод дедукции,

сделать вывод о некоторых греках. А в третьем примере, чтобы сделать то же заключение, что и во втором примере, используется квантор существования ANY (какие-то) — синоним квантора SOME.

Теперь посмотрим, как кванторы SOME, ANY и ALL применяются в SQL.

Рассмотрим пример с бейсбольной статистикой. Бейсбол — это вид спорта, требующий значительных физических нагрузок, особенно для питчера, т.е. игрока, подающего мяч. Ему за время игры приходится от 90 до 150 раз бросать мяч со своей возвышенности до основной базы, или "дома", — 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.

ВОЗМОЖНАЯ ДВУСМЫСЛЕННОСТЬ КВАНТОРА ANY

Первоначально в SQL в качестве квантора существования использовалось ключевое слово ANY. Но оно часто сбивало с толку и приводило к ошибкам, так как в английском языке слово *any* иногда означает "любой", а иногда — "некоторый".

- "Do *any* of you know where Baker Street is?" (*Кто-нибудь* из вас знает, где находится Бейкер-стрит?)
- "I can eat more eggs than *any* of you". (Я могу съесть больше яиц, чем *любой* из вас.)

В первом предложении задается вопрос, существует ли хотя бы один человек, который знает, где находится Бейкер-стрит. Здесь слово *any* используется как квантор существования. Второе предложение — это хвастливое заявление о том, что я могу съесть больше яиц, чем любой из присутствующих. В этом случае слово *any* используется как квантор всеобщности.

По этой причине разработчики стандарта SQL-92, хотя и оставили слово ANY для совместимости с имеющимися продуктами, ввели слово SOME как синоним, устраняющий неоднозначность. Стандарт SQL продолжает поддерживать оба квантора существования.

Предположим, вы собираете данные о количестве тех игр, в которых питчеры Главной лиги бессменно подавали мяч. В одной из ваших таблиц перечислены все питчеры Американской лиги, а в другой — питчеры 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.

В Американской лиге разрешается, чтобы назначенный хиттер (игрок нападения, которому не требуется играть в оборонительной позиции) мог бить битой по мячу вместо одного из девяти игроков, играющих в обороне. В то же время Национальная лига не признает назначенных хиттеров, но признает пинч-хиттеров (заменяющих хиттеров). Когда пинч-хиттер вступает в игру вместо питчера, замененный питчер не имеет права играть до конца матча. Обычно назначенные хиттеры играют вместо питчеров, потому что те, как известно, плохие хиттеры. Питчерам приходится тратить так много времени и усилий на совершенствование своего броска, что у них остается мало времени для тренировок с битой, как это делают остальные игроки.

Скажем, вы предполагаете, 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 ваших наблюдениях относительно того, что назначенные хиттеры позволяют хорошо бросающим, но слабым в обращении с битой питчерам Американской лиги долго оставаться на подаче (пока они в силе), даже если игра идет на равных. Так как назначенные хиттеры уже работают с битой вместо них, становится не важным, что питчеры — слабые хиттеры. Однако в Национальной лиге питчеру все-таки приходится брать в руки биту. Поэтому большинство менеджеров заменяют питчера хиттером, считая его функцию более важной для исхода игры, чем функцию пусть даже очень эффективного питчера. Чтобы проверить свою теорию, вы составляете следующий запрос.

```
SELECT FirstName, LastName
  FROM AMERICAN_LEAGUER
 WHERE CompleteGames > ALL
       (SELECT CompleteGames
          FROM NATIONAL_LEAGUER) ;
```

Подзапрос (внутренняя инструкция SELECT) возвращает список, показывающий для каждого питчера Национальной 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 любой питчер Национальной лиги. В этом запросе используется квантор всеобщности ALL, и, следовательно, весь запрос вернет имена и фамилии тех питчеров Американской лиги, которые провели больше игр с бессменной подачей мяча, чем самый лучший по этому показателю питчер Национальной лиги.

Рассмотрим аналогичный запрос.

```
SELECT FirstName, LastName
  FROM AMERICAN_LEAGUER
```

```
WHERE CompleteGames > ANY
      (SELECT CompleteGames
         FROM NATIONAL_LEAGUER) ;
```

В данном случае вместо квантора всеобщности ALL используется квантор существования ANY. Подзапрос здесь тот же, что и в предыдущем примере. В результате его выполнения мы получаем тот же полный список, показывающий для каждого питчера Национальной лиги количество тех игр, в которых он бессменно подавал 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 здесь как раз и нужен квантор существования ANY. Поскольку вы уже знаете, что хотя бы один из питчеров Национальной лиги ни одной игры не провел бессменно на подаче, результат всего запроса, 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, результат будет точно таким же. И если истинно утверждение "Хотя бы один питчер Национальной лиги ни одной игры не провел на подаче бессменно ", то будет истинным и следующее: "Какой-то (SOME) питчер Национальной лиги ни одной игры не провел на подаче бессменно".

Предикат EXISTS

Для определения того, возвращает ли подзапрос какие-либо строки, можно использовать предикат EXISTS (существует). Если подзапрос возвращает хотя бы одну строку, то текущий результат удовлетворяет условию EXISTS, что позволяет выполниться внешнему запросу. Ниже приведен пример использования предиката EXISTS.

```
SELECT FirstName, LastName
  FROM CUSTOMER
 WHERE EXISTS
       (SELECT DISTINCT CustomerID
          FROM SALES
         WHERE SALES.CustomerID = CUSTOMER.CustomerID) ;
```

В таблице SALES хранятся данные обо всех продажах, выполненных компанией. Столбец CustomerID этой таблицы содержит идентификаторы покупателей, которые участвовали в сделках купли-продажи. В таблице CUSTOMER хранятся имена и фамилии всех покупателей, но нет никакой информации о конкретных покупках.

Подзапрос в последнем примере возвращает строку для каждого покупателя, который сделал хотя бы одну покупку. Внешний запрос возвращает имена и фамилии тех, кто участвовал в сделках, зарегистрированных в таблице SALES.

Предикат EXISTS, как показано в следующем запросе, эквивалентен сравнению результата функции COUNT с нулем.

```
SELECT FirstName, LastName
  FROM CUSTOMER
 WHERE 0 <>
       (SELECT COUNT(*)
          FROM SALES
         WHERE SALES.CustomerID = CUSTOMER.CustomerID);
```

Для каждой строки таблицы SALES, в которой значение CustomerID равно какому-либо значению CustomerID из таблицы CUSTOMER, эта инструкция выводит значения столбцов FirstName (имя) и LastName (фамилия) из таблицы CUSTOMER. Поэтому для каждой покупки, отмеченной в таблице SALES, эта инструкция выводит имя и фамилию покупателя, который ее совершил.

Предикат UNIQUE

Подобно предикату EXISTS, т.е. вместе с подзапросом, можно использовать и предикат уникальности UNIQUE. Но если предикат EXISTS является истинным в случае, когда подзапрос возвращает хотя бы одну строку, то предикат UNIQUE получает значение True лишь тогда, когда среди возвращаемых подзапросом строк не найдется двух одинаковых. Другими словами, предикат UNIQUE будет истинным, если все возвращаемые подзапросом строки будут уникальными. Рассмотрим следующий пример.

```
SELECT FirstName, LastName
  FROM CUSTOMER
 WHERE UNIQUE
       (SELECT CustomerID FROM SALES
         WHERE SALES.CustomerID = CUSTOMER.CustomerID);
```

Эта инструкция возвращает имена и фамилии всех новых покупателей, для которых в таблице SALES зарегистрирована только одна покупка. Два значения NULL *не считаются* равными друг другу (поскольку они попросту неизвестны). Поэтому, когда ключевое слово UNIQUE применяется к таблице (результату выполнения подзапроса), которая содержит только две строки с неопределенными значениями, результат будет истинным.

Предикат DISTINCT

Предикат DISTINCT (отличающийся) напоминает UNIQUE, за исключением отношения к пустым значениям. Если в таблице, полученной в результате выполнения подзапроса, все значения являются уникальными, то можно предполагать, что они отличаются друг от друга. Однако, в отличие от результата

предиката UNIQUE, если к таблице (результату выполнения подзапроса), которая содержит только две строки с неопределенными значениями, применить ключевое слово DISTINCT (а не UNIQUE), то результат будет ложным. Два значения NULL считаются уникальными, но *не отличающимися* друг от друга.

Такая странная ситуация выглядит противоречиво, но этому есть свое объяснение. Дело в том, что в некоторых ситуациях два значения NULL нужно считать отличающимися друг от друга, а в других — одинаковыми. И тогда в первом случае достаточно использовать предикат UNIQUE, а во втором — DISTINCT.

Предикат OVERLAPS

Предикат OVERLAPS позволяет определить, не перекрываются ли два промежутка времени (это очень полезно при составлении корректного расписания). Если два интервала времени перекрываются, то предикат OVERLAPS возвращает значение True, в противном случае — значение False.

Промежуток времени можно указать двумя способами: в виде начального момента времени и конечного или в виде начала интервала и его длительности. Рассмотрим несколько примеров.

```
(TIME '2:55:00', INTERVAL '1' HOUR)
OVERLAPS
(TIME '3:30:00', INTERVAL '2' HOUR)
```

В этом примере будет получено значение True, так как момент времени 3:30 наступает после 2:55, т.е. меньше, чем через час. А вот второй пример.

```
(TIME '9:00:00', TIME '9:30:00')
OVERLAPS
(TIME '9:29:00', TIME '9:31:00')
```

И здесь предикат OVERLAPS вернет значение True, потому что указанные два промежутка времени содержат перекрытие в одну минуту. Следующий пример.

```
(TIME '9:00:00', TIME '10:00:00')
OVERLAPS
(TIME '10:15:00', INTERVAL '3' HOUR)
```

В этом примере будет получено значение False, так как эти два промежутка времени не перекрываются. И наконец, четвертый пример.

```
(TIME '9:00:00', TIME '9:30:00')
OVERLAPS
(TIME '9:30:00', TIME '9:35:00')
```

Здесь предикат вернет значение False: несмотря на то что заданные два промежутка времени являются смежными, они не перекрываются.

Предикат MATCH

В главе 5 речь шла о ссылочной целостности, которая подразумевает поддержание согласованности в многотабличной базе данных. Целостность может быть нарушена, если в дочернюю таблицу добавить строку, для которой нет соответствующей строки в родительской таблице. Подобные проблемы могут возникнуть, если удалить из родительской таблицы строку, но оставить в дочерней таблице строки, связанные с удаленной таблицей.

Предположим, например, что, ведя бизнес, вы собираете данные о своих покупателях в таблицу CUSTOMER, а данные о продажах заносите в таблицу SALES. Вам не следует добавлять строку в таблицу SALES до тех пор, пока вы не введете данные о покупателе, участвующем в соответствующей сделке, в таблицу CUSTOMER. Вам также не следует удалять из таблицы CUSTOMER данные о покупателе, если он участвовал в сделках, информация о которых все еще хранится в таблице SALES.


ЗАПОМНИ!

Перед тем как выполнять вставку или удаление, вам, скорее всего, имеет смысл проверить, не приведет ли эта операция к нарушению целостности данных. Такую проверку поможет выполнить предикат MATCH.

Итак, в качестве примера мы снова используем таблицы CUSTOMER и SALES. Поле CustomerID — это первичный ключ таблицы CUSTOMER и одновременно внешний ключ таблицы SALES. В каждой строке таблицы CUSTOMER должно содержаться уникальное значение CustomerID, не равное NULL. А в таблице SALES ключ CustomerID не является уникальным, потому что у постоянных покупателей его значения повторяются (при многократных покупках). Это обычная ситуация, не угрожающая целостности, потому что в этой таблице поле CustomerID является не первичным, а внешним ключом.


СОВЕТ

Иногда в столбце CustomerID таблицы SALES могут появляться значения NULL, потому что кто-то может зайти в ваш магазин с улицы, быстро купить что-то и выйти еще до того, как вы успеете ввести в таблицу CUSTOMER его имя, фамилию и адрес. Тогда в дочерней таблице может образоваться строка, для которой нет соответствующей строки в родительской таблице. Для решения этой проблемы можно включить в таблицу CUSTOMER строку для "анонимного" пользователя, с которым затем связываются все "бесхозные" продажи.

Предположим, к кассиру подходит покупатель и утверждает, что 18 декабря 2017 года он купил товар F-35 и теперь хочет вернуть его. Его заявление может быть подтверждено путем проверки вашей таблицы SALES с помощью

предиката MATCH. Прежде всего вы должны найти в столбце CustomerID идентификатор покупателя и присвоить его значение переменной :vcustid. Затем можно использовать следующий синтаксис, в котором, помимо поля идентификатора покупателя, задействованы столбцы ProductID (идентификатор товара) и SaleDate (дата продажи).

```
... WHERE (:vcustid, 'F-35', '2017-12-18')
          MATCH
          (SELECT CustomerID, ProductID, SaleDate
             FROM SALES)
```

Если существует запись о продаже с этим идентификатором пользователя, товаром и датой, то предикат MATCH вернет значение True. В таком случае вы возвращаете покупателю деньги. (*Примечание*: если одно из значений в первом аргументе предиката MATCH окажется неопределенным, то он вернет значение True.)

Разработчики SQL добавили предикаты MATCH и UNIQUE по одной и той же причине: чтобы явно выполнять проверки, которые определены для неявных ограничений, связанных со ссылочной целостностью и уникальностью.

Общий синтаксис предиката MATCH выглядит так.

```
выражение_типа_записи MATCH [UNIQUE] [SIMPLE | PARTIAL | FULL ]
                        Подзапрос
```

Ключевые слова UNIQUE, SIMPLE, PARTIAL и FULL связаны с правилами, которые применяются в случае, если *выражение_типа_записи* содержит столбцы с неопределенными значениями. Правила для предиката MATCH совпадают с соответствующими правилами поддержки ссылочной целостности.

Правила ссылочной целостности и предикат MATCH

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) таблицы EMPLOYEE можно объявить внешним ключом, который ссылается на столбец DeptNo (номер отдела) таблицы DEPT (отделы). Это соответствие гарантирует, что при занесении в таблицу EMPLOYEE информации о служащем, работающем в отделе 123, в таблице DEPT должна быть запись, в столбце DeptNo которой содержится значение 123.

Если внешний и первичный ключи состоят из одного столбца, то ситуация тривиальна. Однако оба этих ключа могут состоять из нескольких столбцов. Например, значение в столбце DeptNo (номер отдела) может быть уникальным только в пределах одного региона, т.е. для некоторого значения в столбце

Location (представительство). Поэтому, чтобы однозначно определить строку из таблицы DEPT, необходимо указать значение и столбца Location, и столбца DeptNo. Если, например, отдел 123 существует в двух представительствах, расположенных в Москве и в Твери соответственно, то отделы необходимо указывать как ('Москва', '123') и ('Тверь', '123'). В таком случае для точного указания в таблице EMPLOYEE отдела, где работает сотрудник (строки из таблицы DEPT), необходимо использовать два столбца. Их можно назвать EmpLoc и EmpDeptNo. Если сотрудник работает в московском отделе 123, то значениями столбцов EmpLoc и EmpDeptNo будут соответственно 'Москва' и '123'. Таким образом, объявить внешний ключ в таблице EMPLOYEE можно так.

```
FOREIGN KEY (EmpLoc, EmpDeptNo)
   REFERENCES DEPT (Location, DeptNo)
```

СОВЕТ

Вывод правильных заключений на основе ваших данных значительно усложняется, если в этих данных содержатся пустые значения. Данные с такими значениями в разных ситуациях интерпретируются по-разному. Возможность изменять интерпретацию данных, в которых встречаются значения NULL, как раз и реализуется с помощью ключевых слов UNIQUE, SIMPLE, PARTIAL и FULL. Если в ваших данных нет пустых значений, то можете не морочить себе голову, пропустить оставшуюся часть раздела и сразу перейти к следующему. Если же в ваших данных есть такие значения, то приведенная здесь информация важна. Постарайтесь не торопясь, внимательно ознакомиться с приведенным ниже списком. В каждом его пункте описана отдельная ситуация, связанная со значениями NULL, а также представлен вариант решения конкретной задачи с помощью предиката MATCH.

Ниже приведено несколько сценариев, иллюстрирующих правила работы с пустыми значениями и предикатом MATCH.

- **Оба значения являются или не являются пустыми.** Если значения EmpLoc и EmpDeptNo вместе являются или не являются пустыми, то правила ссылочной целостности будут теми же, что и для ключей, состоящих из одного столбца с неопределенными или определенными значениями.
- **Одно значение пустое, а другое — нет.** Если, например, значение EmpLoc пустое, а EmpDeptNo — нет или наоборот, то нужны новые правила. Когда в таблицу EMPLOYEE при вставке или обновлении ее строк вводятся значения EmpLoc и EmpDeptNo, такие как (NULL, '123') или ('Москва', NULL), существует шесть вариантов, при которых используются правила SIMPLE, PARTIAL и FULL вместе с ключевым словом UNIQUE или без него.

СТАНДАРТНЫЕ ПРАВИЛА

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

В соответствии со стандартом SQL-89 по умолчанию использовалось правило UNIQUE. Позже начали обсуждаться и другие варианты. Альтернативные предложения рассматривались уже во время разработки стандарта SQL-92. Некоторые упорно настаивали на правиле PARTIAL и считали, что оно должно быть единственно возможными. С их точки зрения, правила SQL-89 (UNIQUE) были настолько неприемлемы, что их стоит считать ошибкой, которую необходимо исправить с помощью правила PARTIAL. Были и те, кто предпочитал использовать правило UNIQUE, а правило PARTIAL считал непонятным, двусмысленным и неэффективным. Впрочем, были также предложения ввести дополнительные соглашения, описываемые правилом FULL. В конце концов, эта дискуссия была разрешена следующим образом: пользователи получили в свое распоряжение все три ключевых слова и теперь могли выбрать нужный вариант. Впоследствии, с появлением стандарта SQL:1999, добавилось еще и правило SIMPLE. В целом увеличение числа правил делает работу с пустыми значениями какой угодно, но только не простой. В настоящее время, если не указаны ключевые слова SIMPLE, PARTIAL или FULL, будет выполняться правило SIMPLE.

- **Используется ключевое слово UNIQUE.** Чтобы предикат MATCH был истинным, совпадающая строка, найденная в результате выполнения подзапроса к таблице, должна быть уникальной.
- **Оба компонента в выражении с записью являются пустыми.** В этом случае предикат MATCH возвращает значение True, каким бы ни было содержимое сравниваемой таблицы, полученной при выполнении подзапроса.
- **В выражении с записью и ключевым словом SIMPLE, но без UNIQUE, ни один из компонентов не является пустым, и при этом хотя бы одна строка в таблице, полученной при выполнении подзапроса, совпадает с этим выражением.** В таком случае предикат MATCH возвращает значение True, иначе — False.
- **В выражении с записью и ключевыми словами SIMPLE и UNIQUE ни один из компонентов не является пустым, и при этом хотя бы одна строка в таблице, полученной при выполнении подзапроса, уникальна и совпадает с этим выражением.** В таком случае предикат MATCH возвращает значение True, иначе — False.
- **В выражении с записью и ключевым словом SIMPLE какой-либо из компонентов является неопределенным.** В этом случае предикат MATCH возвращает значение True.

- **В выражении с записью и ключевым словом PARTIAL, но без UNIQUE, какой-либо из компонентов не является пустым, и при этом непустая часть хотя бы одной строки в таблице, полученной при выполнении подзапроса, совпадает с этим выражением.** В таком случае предикат MATCH возвращает значение True, иначе — значение False.
- **В выражении с записью и ключевыми словами PARTIAL и UNIQUE какой-либо из компонентов не является пустым, и при этом непустые части этого выражения совпадают с непустыми частями хотя бы одной уникальной строки в таблице, полученной при выполнении подзапроса.** В таком случае предикат MATCH возвращает значение True, иначе — значение False.
- **Ни один из компонентов выражения с записью и ключевым словом FULL, но без UNIQUE, не является пустым, и при этом хотя бы одна строка в таблице, полученной при выполнении подзапроса, совпадает с этим выражением.** В таком случае предикат MATCH возвращает значение True, иначе — значение False.
- **Ни один из компонентов выражения с записью и ключевыми словами FULL и UNIQUE не является пустым, и при этом хотя бы одна строка в таблице, полученной при выполнении подзапроса, уникальна и совпадает с этим выражением.** В таком случае предикат MATCH возвращает значение True, иначе — значение False.
- **Один из компонентов выражения типа записи является неопределенным, причем указано ключевое слово FULL.** В этом случае предикат MATCH возвращает значение False.

Логические операторы

Как видно из предыдущих примеров, чтобы извлечь из таблицы нужные строки, одного условия в запросе обычно недостаточно. В одних случаях условий, которым должны удовлетворять строки, должно быть не меньше двух. В других же случаях требуется, чтобы выбранные строки удовлетворяли *одному из* нескольких условий. А иногда нужно получить только те строки, которые *не* удовлетворяют указанному условию. Для реализации этих требований в SQL существуют логические операторы AND, OR и NOT.

AND

Если для извлечения строки необходимо, чтобы *все* заданные условия выполнялись, т.е. были истинными, используйте логический оператор AND.

Рассмотрим следующий пример, в котором используются поля `InvoiceNo` (номер счета-фактуры), `SaleDate` (дата продажи), `SalesPerson` (продавец), `TotalSale` (всего продано) из таблицы `SALES`.

```
SELECT InvoiceNo, SaleDate, Salesperson, TotalSale
  FROM SALES
 WHERE SaleDate >= '2017-12-14'
   AND SaleDate <= '2017-12-20' ;
```

Здесь предложение `WHERE` предполагает выполнение двух условий:

- дата `SaleDate` должна быть не раньше 14 декабря 2017 года;
- дата `SaleDate` должна быть не позже 20 декабря 2017 года.

Таким образом, обоим условиям будут удовлетворять только те строки, которые содержат данные о продажах в течение недели, начавшейся 14 декабря 2017 года. Запрос вернет именно такие строки.


ВНИМАНИЕ!

Оператор `AND` имеет чисто логический смысл. Такое ограничение иногда может привести к путанице, потому что союз "и" люди обычно используют в более широком смысле. Предположим, например, что ваш начальник говорит: "Мне нужны данные о продажах, выполненных 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 начальнику. Ведь 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, зарегистрированных в таблице `SALES`, Фергюсон и Форд не провели вместе. Второй же запрос вернет информацию обо всех продажах, сделанных *либо* Фергюсоном, *либо* Фордом. Скорее всего, именно это и требовалось вашему начальнику.

OR

Если для извлечения строки необходимо, чтобы из нескольких условий эта строка удовлетворяла хотя бы одному, то используйте логический оператор `OR`.

```
SELECT InvoiceNo, SaleDate, SalesPerson, TotalSale
  FROM SALES
 WHERE SalesPerson = 'Ford'
    OR TotalSale > 200 ;
```

В результате выполнения этого запроса будут получены данные обо всех продажах Форда (которые были сделаны на любую сумму), а также о продажах всех остальных продавцов, но только в случае, если эти другие сделки превысили сумму в 200 долларов.

NOT

Для отрицания условия служит оператор `NOT`. Если к условию, которое возвращает значение `True`, добавить `NOT`, то результатом станет значение `False`. Если же к условию, которое возвращает значение `False`, добавить `NOT`, то оно будет возвращать `True`. Рассмотрим следующий пример.

```
SELECT InvoiceNo, SaleDate, SalesPerson, TotalSale
  FROM SALES
 WHERE NOT (SalesPerson = 'Ford') ;
```

Этот запрос возвращает данные по всем продажам, совершенным всеми продавцами, кроме Форда.

ВНИМАНИЕ!

Иногда при использовании логического оператора (`AND`, `OR` или `NOT`) неясно, какова его область действия. Чтобы точно указать применение оператора к конкретному предикату, заключите его в круглые скобки. В последнем примере оператор `NOT` применяется ко всему предикату `(SalesPerson = 'Ford')`, а не к какой-то его части.

Предложение GROUP BY

Иногда возникает потребность не просто вывести отдельные записи, а оценить и как-то обобщить данные, содержащиеся в некоторой группе записей. В этом случае используется предложение `GROUP BY`.

Предположим, что вы, будучи начальником отделом сбыта, хотите оценить эффективность работы менеджеров по продажам. Для этого можно построить простой запрос на основе инструкции `SELECT`.

```
SELECT InvoiceNo, SaleDate, SalesPerson, TotalSale
  FROM SALES;
```

В этом случае вы получите результат, подобный показанному на рис. 10.1.

Рис. 10.1. Результат отбора информации о продажах с 12.01.2017 по 12.07.2017

Этот результат, конечно, дает некоторое представление о том, как работают ваши продавцы, поскольку здесь выводится информация о небольшом количестве продаж. Однако на практике компании могут иметь гораздо бо́льшие объемы продаж, и в этом случае уже непросто понять, достигнуты ли цели, поставленные руководством компании. Чтобы проанализировать эффективность работы продавцов, можно объединить предложение `GROUP BY` с одной из функций агрегирования (которые также называют итоговыми функциями), что позволит получить количественное представление о результатах торговой деятельности компании. Например, чтобы узнать, кто из продавцов работает с более дорогостоящими и прибыльными товарами, используйте функцию среднего значения (`AVG`).

```
SELECT SalesPerson, AVG(TotalSale)
  FROM SALES
 GROUP BY SalesPerson;
```

Результат этого запроса, полученный в Microsoft Access 2016, показан на рис. 10.2. При выполнении этого запроса с использованием другой СУБД вы получите такой же результат, но таблица результатов может выглядеть чуть по-другому.

Как видно на рис. 10.2, средние продажи Беннетта значительно выше, чем у двух других продавцов. Чтобы сравнить общие объемы продаж по каждому продавцу, выполните следующий запрос.

```
SELECT SalesPerson, SUM(TotalSale)
  FROM SALES
 GROUP BY SalesPerson;
```

Результат выполнения этого запроса приведен на рис. 10.3.

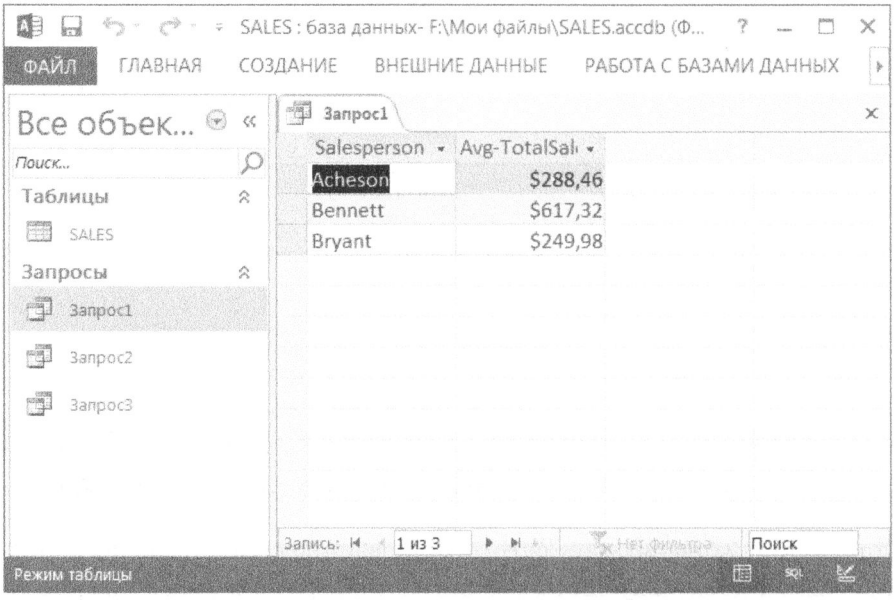


Рис. 10.2. Средний уровень продаж по каждому продавцу

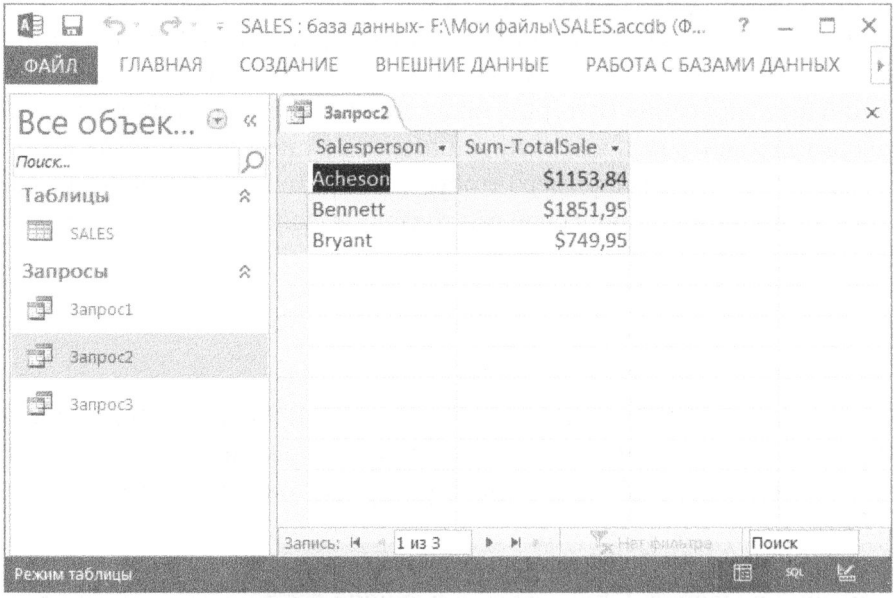


Рис. 10.3. Общие объемы продаж по каждому продавцу

У Беннетта также самый высокий объем продаж, что согласуется с самым высоким значением среднего уровня продаж.

Предложение HAVING

Предложение HAVING позволяет еще глубже проанализировать сгруппированные данные. Это фильтр, который по своему действию похож на предложение WHERE, но в отличие от последнего работает не с отдельными строками, а с их группами. Проиллюстрируем действие предложения HAVING, используя следующий пример. Предположим, менеджер по продажам, высоко оценивая заслуги Беннетта, хочет сосредоточиться на работе других продавцов. Для этого ему необходимо исключить из общих данных объемы продаж Беннетта, поскольку те находятся "в другой весовой категории". Чтобы сделать это, воспользуемся предложением HAVING.

```
SELECT Salesperson, SUM(TotalSale)
  FROM SALES
 GROUP BY Salesperson
HAVING Salesperson <> 'Bennett';
```

Результат выполнения этого запроса показан на рис. 10.4. Строка, относящаяся к продавцу по фамилии "Bennett", на экран не выводится.

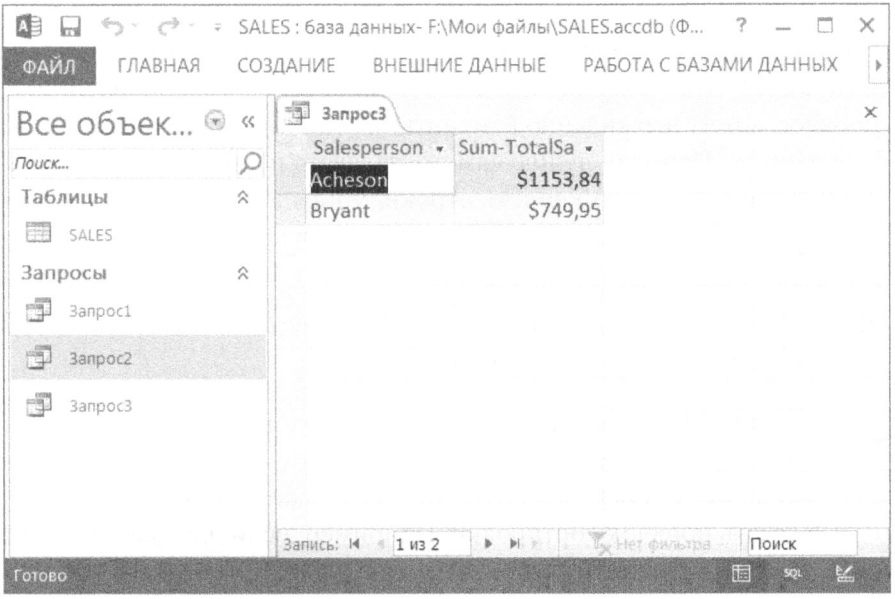


Рис. 10.4. Общие объемы продаж по всем продавцам, за исключением Беннетта

Предложение ORDER BY

Чтобы отобразить таблицу, полученную при выполнении запроса, в алфавитном порядке (по возрастанию или убыванию), используйте предложение ORDER BY. В то время как предложение GROUP BY собирает строки в группы и сортирует последние по алфавиту, предложение ORDER BY сортирует отдельные строки. Предложение ORDER BY должно быть последним в запросе. Если запрос содержит еще и предложение GROUP BY, то сначала оно собирает строки результата в группы, а затем предложение ORDER BY сортирует строки в каждой группе. Если же предложение GROUP BY не задано, то вся таблица рассматривается как одна группа, и тогда предложение ORDER BY сортирует все ее строки, упорядочивая значения в столбцах, указанных в этом предложении.

Это можно проиллюстрировать с помощью таблицы SALES. Она содержит столбцы InvoiceNo (номер счета-фактуры), SaleDate (дата продажи), SalesPerson (продавец), TotalSale (всего продано). Выполнив следующий простой запрос, можно вывести на экран все данные таблицы SALES, но они будут расположены в произвольном порядке.

```
SELECT * FROM SALES ;
```

В одних реализациях используется порядок, в котором строки вставлялись в таблицу, а в других — порядок, заданный при последнем обновлении. Кроме того, порядок может неожиданно измениться, если кто-то физически реорганизует базу данных. Поэтому лучше самому задавать порядок следования строк. Например, если вам нужно увидеть строки, упорядоченные по дате продажи (столбец SaleDate), выполните следующую инструкцию.

```
SELECT * FROM SALES ORDER BY SaleDate ;
```

При выполнении этого запроса все строки таблицы SALES будут выстроены в порядке, заданном значениями столбца SaleDate.

СОВЕТ

Порядок расположения тех строк, в столбце SaleDate которых хранятся одинаковые значения, зависит от конкретной реализации. Впрочем, и для строк с одинаковыми значениями SaleDate можно указать порядок сортировки. Например, если для каждого значения SaleDate вы хотите выводить строки таблицы SALES в порядке возрастания номера счета-фактуры (столбец InvoiceNo), то используйте следующий запрос.

```
SELECT * FROM SALES ORDER BY SaleDate, InvoiceNo ;
```

В этом примере строки таблицы SALES вначале упорядочиваются по значениям SaleDate, а затем строки с одинаковым значением SaleDate

упорядочиваются по значениям InvoiceNo. Однако не путайте этот пример с таким запросом.

```
SELECT * FROM SALES ORDER BY InvoiceNo, SaleDate ;
```

При его выполнении данные о продажах из таблицы SALES упорядочиваются по столбцу InvoiceNo. Затем для каждого значения InvoiceNo строки таблицы SALES располагаются в порядке, задаваемом столбцом SaleDate. Скорее всего, нужный результат вы не получите, потому что маловероятно, чтобы для одного номера счета-фактуры существовало несколько разных дат продажи.

Следующий запрос служит еще одним примером того, как SQL может извлекать данные.

```
SELECT * FROM SALES ORDER BY SalesPerson, SaleDate ;
```

В этом примере упорядочение выполняется вначале по столбцу SalesPerson, а затем — по столбцу SaleDate. Просмотрев данные, расположенные в таком порядке, вы, возможно, захотите его изменить.

```
SELECT * FROM SALES ORDER BY SaleDate, SalesPerson ;
```

Теперь упорядочение выполняется вначале по столбцу SaleDate, а затем — по столбцу SalesPerson.

Во всех приведенных выше примерах упорядочение выполнялось в порядке возрастания (ASC), который принят по умолчанию. Последняя инструкция SELECT вначале выводит строки таблицы SALES, относящиеся к самым первым продажам, а строки с одной и той же датой будут упорядочены по фамилии продавца (Беннетт будет стоять перед Брайантом). Если вы предпочитаете порядок по убыванию (DESC), то можете указать его для одного или нескольких столбцов, задаваемых предложением ORDER BY.

```
SELECT * FROM SALES
 ORDER BY SaleDate DESC, Salesperson ASC ;
```

В этом примере для данных о продажах указывается порядок убывания дат, в результате чего записи о недавних продажах будут показаны первыми, но для продавцов будет использован порядок по возрастанию, т.е. их фамилии будут расположены в обычном алфавитном порядке.

Использование инструкции FETCH для ограничения выборки

Любое изменение стандарта ISO/IEC SQL расширяло возможности языка. Казалось бы, нужно только радоваться. Но иногда, внося изменение, трудно

предугадать все возможные последствия. Так случилось с новой инструкцией ограничения выборки FETCH в стандарте SQL:2008.

Идея инструкции FETCH состоит в следующем. Несмотря на то что инструкция SELECT может возвращать неопределенное количество строк, вас, возможно, интересуют только первые три или первые десять. Реализуя эту идею, стандарт SQL:2008 добавил синтаксис, показанный в следующем примере.

```
SELECT Salesperson, AVG(TotalSale)
  FROM SALES
 GROUP BY Salesperson
 ORDER BY AVG(TotalSale) DESC
 FETCH FIRST 3 ROWS ONLY;
```

Инструкция, казалось бы, выглядит нормально. Из списка продавцов, торгующих дорогостоящими товарами, вы хотите увидеть первую тройку. Но здесь есть небольшая загвоздка. А что если на третье место "претендуют" сразу три продавца, у которых совпало значение среднего объема продаж, по которому и упорядочивается список? Ведь только один из них может занять "призовое место". Кто именно? Не ясно.

Подобная неоднозначность недопустима в работе с базой данных, поэтому такая ситуация была исправлена в стандарте SQL:2011. Новый синтаксис включает новое предложение — модификатор WITH TIES (с учетом равенства).

```
SELECT Salesperson, AVG(TotalSale)
  FROM SALES
 GROUP BY Salesperson
 ORDER BY AVG(TotalSale) DESC
 FETCH FIRST 3 ROWS WITH TIES;
```

Теперь результат вполне определен: если полученные значения совпадают, то в результат включаются все они. Если же вы уберете из этой инструкции модификатор WITH TIES, то результат снова станет неопределенным.

Стандарт SQL:2011 добавил еще два усовершенствования в инструкцию FETCH. Во-первых, теперь разрешено задавать не только точное количество выводимых строк, но и их число, *выраженное в процентах*. Рассмотрим следующий пример.

```
SELECT Salesperson, AVG(TotalSale)
  FROM SALES
 GROUP BY Salesperson
 ORDER BY AVG(TotalSale) DESC
 FETCH FIRST 10 PERCENT ROWS ONLY;
```

Очевидно, что и при "процентном" задании числа выводимых строк весьма вероятна проблема равных строк, поэтому здесь тоже можно использовать модификатор WITH TIES. Решение о включении этого модификатора должно

приниматься в зависимости от того, что именно вы хотите получить в данной конкретной ситуации.

Во-вторых, теперь можно начинать вывод строк не с самого начала списка, а *со смещением*. Предположим, вас интересуют *не* первые три или *не* первые десять записей (заданные числом или в процентах), а вторые три или вторые десять. Возможно, вы хотите сразу перейти к более глубокой части результирующего набора данных. Стандарт SQL:2011 учитывает возможность такой ситуации. Вот пример кода с новыми элементами синтаксиса.

```
SELECT Salesperson, AVG(TotalSale)
  FROM SALES
 GROUP BY Salesperson
 ORDER BY AVG(TotalSale) DESC
       OFFSET 3 ROWS
 FETCH NEXT 3 ROWS ONLY;
```

Ключевое слово `OFFSET` позволяет указать, сколько первых строк в таблице результатов нужно опустить. Ключевое слово `NEXT` определяет количество выводимых строк, следующих непосредственно за смещением. При выполнении этой инструкции будут выведены строки с данными о продавцах, которые имеют четвертый, пятый и шестой (по убыванию) результаты вычисления среднего объема продаж. Как видите, без применения синтаксиса `WITH TIES` проблема неопределенности по-прежнему остается актуальной. Если окажется, что третий, четвертый и пятый продавцы имеют равные значения, будет неясно, кого из них включить в первую группу, а каких двух продавцов включить во вторую.

ВНИМАНИЕ!

Возможно, имеет смысл избегать использования инструкции `FETCH` для ограничения числа выводимых строк. Уж слишком велика вероятность получения дезориентирующих результатов.

Использование оконных функций для создания результирующего множества

В стандарте SQL:1999 впервые были представлены окна и оконные функции. С помощью окна пользователь может произвольным образом разделить набор данных на части (секции), в каждой секции произвольно упорядочить строки и определить коллекцию строк (оконный блок, или фрейм окна), которая будет связана с заданной строкой.

Фрейм окна для строки *R* представляет собой некоторое подмножество секции, содержащей строку *R*. Например, фрейм окна может включать все строки

от начала секции и до строки *R* (в том числе и ее саму) на основе заданного способа упорядочения строк в секции.

Оконная функция вычисляет значение для строки *R* на основе строк, входящих в оконный фрейм строки *R*.

Например, у вас есть таблица SALES, в которой содержатся столбцы CustID, InvoiceNo и TotalSale. Предположим, ваш начальник отдела продаж интересуется, какие общие объемы продаж были зафиксированы для каждого покупателя в пределах заданного диапазона номеров счетов-фактур. Получить нужную информацию можно с помощью следующего SQL-кода.

```
SELECT CustID, InvoiceNo,
   SUM (TotalSale) OVER
   ( PARTITION BY CustID
     ORDER BY InvoiceNo
     ROWS BETWEEN
     UNBOUNDED PRECEDING
     AND CURRENT ROW )
  FROM SALES;
```

Предложение OVER определяет, как разбить строки запроса на секции до обработки (в данном случае функцией SUM). Секция здесь привязывается к каждому покупателю. В каждую секцию попадает список номеров счетов, и в рамках заданного диапазона строк, т.е. для каждого покупателя, будут суммироваться все значения TotalSale.

К исходным оконным функциям стандарт SQL:2011 добавил ряд усовершенствований и ввел новые ключевые слова.

Разделение окна на фрагменты с помощью функции NTILE

Оконная функция NTILE делит упорядоченную секцию окна на некоторое количество (*n*) фрагментов (групп), нумеруя их от 1 до *n*. Если количество строк в секции *m* не делится нацело на *n*, то после того как функция NTILE равномерно заполнит фрагменты строками, остаток от деления *m*/*n*, именуемый *r*, будет равномерно распределен между первыми *r* фрагментами, делая их больше остальных.

Предположим, вы хотите разделить своих служащих на пять категорий (групп) по убыванию величины оклада. Для этого можно использовать следующий код.

```
SELECT FirstName, LastName, NTILE (5)
  OVER (ORDER BY Salary DESC)
    AS BUCKET
  FROM Employee;
```

Если, к примеру, у вас работает одиннадцать сотрудников, то в каждую группу попадают два сотрудника, за исключением первой группы, которой "повезет" заполучить на одного сотрудника больше. Итак, в первую группу войдут три сотрудника с самыми высокими окладами, а в пятую — только два, причем с самыми низкими окладами.

Навигация в пределах окна

В стандарт SQL:2011 было добавлено пять оконных функций (`LAG`, `LEAD`, `NTH_VALUE`, `FIRST_VALUE` и `LAST_VALUE`), которые вычисляют выражение в строке *R2*, расположенной в пределах фрейма окна, относящегося к текущей строке *R1*. Эти функции позволяют извлекать информацию из строк, которые заданы во фрейме окна, определенного для текущей строки.

Функция LAG — взгляд назад

Функция `LAG` позволяет получить информацию из текущей строки в окне, а также информацию из другой строки, которая заданным образом предшествует текущей.

Предположим, у вас есть таблица, в которой регистрируются общие объемы продаж, полученные за каждый день текущего года. Вас интересует, насколько сегодняшний объем продаж отличается от вчерашнего. Ваш интерес может удовлетворить функция `LAG`.

```
SELECT TotalSale AS TodaySale,
   LAG (TotalSale) OVER
       (ORDER BY SaleDate) AS PrevDaySale
  FROM DailyTotals;
```

Для каждой строки в таблице `DailyTotals` этот запрос вернет строку, которая будет содержать объем продаж, зафиксированный на данную дату, и объем продаж, зафиксированный на предыдущую дату. По умолчанию смещение между датами равно 1, поэтому здесь для каждой строки берется "вчерашний" результат. Чтобы сравнить объемы продаж текущего дня с результатами работы недельной давности, можно воспользоваться следующим кодом.

```
SELECT TotalSale AS TodaySale,
   LAG (TotalSale, 7) OVER
       (ORDER BY SaleDate) AS PrevDaySale
  FROM DailyTotals;
```

Первые семь строк в оконном фрейме не будут иметь предшественника недельной давности. В такой ситуации для значения `PrevDaySale` по умолчанию возвращается результат `NULL`. Если он вас не устраивает, т.е. вместо `NULL` вы

предпочитаете использовать другое значение (например, 0), укажите его в инструкции явным образом.

```
SELECT TotalSale AS TodaySale,
   LAG (TotalSale, 7, 0) OVER
       (ORDER BY SaleDate) AS PrevDaySale
  FROM DailyTotals;
```

По умолчанию при выполнении функции LAG учитываются строки с любым значением заданного выражения (в нашем случае это поле TotalSale), которое может быть пустым (NULL). Если вы хотите опустить такие NULL-строки и учитывать только те, которые содержат действительные (непустые) значения, используйте модификатор IGNORE NULLS, как показано в следующем примере.

```
SELECT TotalSale AS TodaySale,
   LAG (TotalSale, 7, 0) IGNORE NULLS
  OVER (ORDER BY SaleDate) AS PrevDaySale
  FROM DailyTotals;
```

Функция LEAD — взгляд вперед

Оконная функция LEAD по своему действию подобна функции LAG, но она "смотрит" не назад, как LAG, а вперед, т.е. интересуется строкой, следующей за текущей в оконном фрейме. Рассмотрим пример.

```
SELECT TotalSale AS TodaySale,
  LEAD (TotalSale, 7, 0) IGNORE NULLS
  OVER (ORDER BY SaleDate) AS NextDaySale
  FROM DailyTotals;
```

Функция NTH_VALUE — поиск заданной строки

Функция NTH_VALUE по своему действию подобна функциям LAG и LEAD, но вместо вычисления выражения в строке, предшествующей или следующей за текущей строкой, она вычисляет выражение в строке, отстоящей на величину заданного смещения от первой или последней строки в оконном фрейме.

Рассмотрим пример.

```
SELECT TotalSale AS ChosenSale,
       NTH_VALUE (TotalSale, 2)
            FROM FIRST
          IGNORE NULLS
            OVER ( ORDER BY SaleDate
                   ROWS BETWEEN 10 PRECEDING AND 10 FOLLOWING )
              AS EarlierSale
  FROM DailyTotals;
```

В этом примере выражение EarlierSale вычисляется следующим образом.

- Формируется оконный фрейм, связанный с текущей строкой. Он включает десять предыдущих и десять следующих строк.
- Значение TotalSale вычисляется в каждой строке оконного фрейма.
- Поскольку задано предложение IGNORE NULLS, любые строки, в которых поле TotalSale содержит пустое значение, опускаются.
- Начиная с первого значения, оставшегося после исключения строк, которые содержат пустое значение в поле TotalSale, переходим на две строки вперед (поскольку задано предложение FROM FIRST).

Переменная EarlierSale будет равна значению TotalSale из заданной строки.

Если не хотите опускать строки, содержащие значение NULL в поле TotalSale, используйте вместо предложения IGNORE NULLS предложение RESPECT NULLS. Если в функции NTH_VALUE вместо предложения FROM FIRST использовать предложение FROM LAST, то вместо счета в прямом направлении, начиная с первой записи в оконном фрейме, она будет отсчитывать строки в обратном направлении (назад), начиная с последней записи в оконном фрейме. Число, задающее количество отсчитываемых строк, все равно должно быть положительным, даже если вы считаете в обратном направлении.

Функции FIRST_VALUE и LAST_VALUE — специальные случаи

Функции FIRST_VALUE и LAST_VALUE представляют собой специальные случаи использования функции NTH_VALUE. Функция FIRST_VALUE является эквивалентом функции NTH_VALUE, в которой задано предложение FROM FIRST, а смещение установлено равным нулю. Функция LAST_VALUE — эквивалент функции NTH_VALUE с предложением FROM LAST и нулевым смещением. С каждой из этих двух специальных функций можно использовать модификатор IGNORE NULLS либо RESPECT NULLS.

Вложение оконных функций

Иногда для получения нужного результата самым простым путем достаточно вложить одну функцию в другую. Стандарт SQL:2011 предоставил возможность вложения оконных функций. В качестве примера рассмотрим случай, когда потенциальный инвестор пытается понять, подходящее ли сейчас время для покупки пакета акций. Чтобы разобраться в этом вопросе, он хочет сравнить текущий курс акций с ценой, по которой они были проданы при совершении последних ста сделок. Ему нужно узнать, в скольких случаях из предыдущих ста сделок интересующая его акция была продана дешевле текущей цены. Для получения ответа инвестор создает такой запрос.

```
SELECT SaleTime,
     SUM ( CASE WHEN SalePrice <
           VALUE OF (SalePrice AT CURRENT ROW)
           THEN 1 ELSE 0 )
     OVER ( ORDER BY SaleTime
            ROWS BETWEEN 100 PRECEDING AND CURRENT ROW )
  FROM StockSales;
```

Здесь окно охватывает сто строк, предшествующих текущей строке, которые соответствуют ста продажам, совершенным непосредственно перед текущим моментом. Каждый раз, когда при оценке очередной строки оказывается, что значение `SalePrice` меньше текущей цены, сумма увеличивается на 1. Конечный результат даст нам число, которое будет означать количество продаж из предыдущих ста, совершенных по цене, меньшей, чем текущая стоимость акции.

Выполнение расчетов по группам записей

Иногда при использовании ключа сортировки, выбранного для упорядочения строк секции, имеют место дубликаты. В таких случаях целесообразно объединить все строки с одинаковыми значениями ключа сортировки в группу, а затем выполнить нужные действия в группе. Для этого можно использовать ключевое слово `GROUPS`.

Рассмотрим пример.

```
SELECT CustomerID, SaleDate,
   SUM (InvoiceTotal) OVER
   ( PARTITION BY CustomerID
     ORDER BY SaleDate
     GROUPS BETWEEN 2 PRECEDING AND 2 FOLLOWING )
  FROM Customers;
```

Оконный фрейм в этом примере может состоять из пяти групп: две группы, предшествующие группе, содержащей текущую запись, группа, содержащая текущую запись, и две группы, следующие за группой, содержащей текущую запись. Строки в каждой группе имеют одно и то же значение даты (`SaleDate`), причем значение `SaleDate`, связанное с одной группой, отличается от значений `SaleDate` в других группах.

Распознавание шаблона записи

После появления стандарта SQL:2016 стало возможным распознавать шаблоны в таблице благодаря функции `MATCH_RECOGNIZE`. Как следует из ее названия, эта функция распознает совпадения между данными в столбце таблицы и шаблоном.

Например, работая со строками таблицы, вы можете поинтересоваться, имеется ли какой-то шаблон или тренд в значениях столбца. Если существует шаблон, который считается существенным, то было бы хорошо иметь возможность распознать его при появлении. Для этого как раз и предназначена функция MATCH_RECOGNIZE. Она весьма гибкая, но эта гибкость влечет за собой повышенную сложность.

Функция MATCH_RECOGNIZE допустима в предложении FROM инструкции SELECT. Она обрабатывает строки из входной таблицы и параллельно создает выходную таблицу, которая содержит строки, созданные при распознавании целевого шаблона.

При этом используется следующий синтаксис.

```
SELECT <столбцы> FROM <входная таблица>
MATCH_RECOGNIZE (
   [PARTITION BY <столбцы>]
   [ORDER BY <столбцы>]
   MEASURES MATCH_NUMBER () AS <число>,
      <атрибут> AS <начало шаблона>,
      LAST <атрибут> AS <смена тренда>,
      LAST <атрибут> AS <конец шаблона>,
      [<другие функции>]
   ONE ROW PER MATCH/ALL ROWS PER MATCH
   AFTER MATCH SKIP <куда перейти после найденного совпадения>
   PATTERN <искомый шаблон записи>
   [SUBSET <объединение шаблонов записей>]
   DEFINE <логическое условие>
) ;
```

Как видите, я не шутил, когда говорил о том, что функция MATCH_RECOGNIZE довольно сложна. Однако, учитывая, насколько важно иметь возможность находить шаблоны в данных, попытка научиться использовать их может стоить потраченного на нее времени. В техническом отчете ISO/IEC, занимающем восемьдесят страниц, содержатся объяснения и примеры использования, которые делают функцию MATCH_RECOGNIZE более понятной. Этот отчет доступен по следующему адресу:

```
http://standards.iso.org/ittf/PubliclyAvailableStandards/
c065143_ISO_IEC_TR_19075-5_2016.zip
```

Глава 11

Использование реляционных операторов

В ЭТОЙ ГЛАВЕ...

- Объединение таблиц, имеющих сходную структуру
- Объединение таблиц, имеющих разную структуру
- Извлечение нужных данных из нескольких 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 скрыт смысл реляционности. Вообще говоря, эти базы данных называют *реляционными* потому, что они состоят из множества *связанных* между собой таблиц (от англ. "related" — "связанный", "relational" — "родственный").

Так как данные, хранящиеся в реляционной базе, распределены по множеству таблиц, запросы обычно обращаются к нескольким таблицам. В SQL предусмотрены операции, которые объединяют данные из разных исходных таблиц в одну. Это операторы `UNION`, `INTERSECTION` и `EXCEPT`, а также семейство операторов `JOIN`, каждый из которых объединяет данные своим особым способом.

Оператор UNION

Оператор UNION — это SQL-реализация операции объединения из реляционной алгебры. Он позволяет извлечь информацию из нескольких таблиц с одинаковой структурой. Термин *одинаковая структура* подразумевает следующее:

- у всех таблиц одинаковое количество столбцов;
- у всех соответствующих столбцов идентичный тип данных и одинаковая длина.

При соблюдении этих критериев таблицы считаются совместимыми для объединения. В результате объединения двух таблиц возвращаются все строки, содержащиеся в каждой из них, но без дубликатов.

Предположим, вы создаете бейсбольную базу данных, которая состоит из двух таблиц, совместимых для объединения: AMERICAN (Американская лига) и NATIONAL (Национальная лига). Каждая из них содержит по три столбца, и типы всех соответствующих столбцов совпадают. В сущности, даже имена таких столбцов одинаковые, хотя для объединения это условие не является обязательным.

В таблице NATIONAL перечислены имена и фамилии питчеров Национальной лиги, а также количество игр, в которых они все время играли 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 данном примере для иллюстрации действия оператора UNION я вставил в каждую из таблиц всего по нескольку 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 точно так же, как и оператор `UNION` без ключевого слова `DISTINCT`, — в обоих случаях строки-дубликаты удаляются из результирующего набора.

ВНИМАНИЕ!

Символ "звездочка" (`*`) в приведенных выше инструкциях `SELECT` означает выбор *всех* столбцов в таблице. Это сокращенное обозначение в большинстве случаев работает отлично, но если реляционные операции со звездочкой использовать во встроенном или модульном коде SQL, то могут возникнуть проблемы. Что случится, если в одну таблицу вы добавите один или несколько новых столбцов, а в другую — нет? Или если вы добавите в две таблицы разные столбцы? Тогда эти таблицы больше не будут совместимыми для объединения, и ваша программа при следующем запуске сработает уже неправильно. И даже если в обе таблицы будут добавлены одни и те же столбцы (т.е. таблицы останутся совместимыми для объединения), то программа, вероятнее всего, не будет готова работать с этими дополнительными данными. Поэтому лучше явно перечислять нужные столбцы, а не полагаться на звездочку. В то же время при вводе с консоли разового SQL-запроса звездочка работает прекрасно. Если запрос на объединение не даст нужного результата, вы сможете тут же отобразить структуру таблиц, чтобы проверить их на предмет совместимости для объединения.

Оператор UNION ALL

Как упоминалось выше, оператор UNION обычно устраняет строки-дубликаты из результата. В большинстве случаев такое поведение устраивает пользователя. Но иногда одинаковые строки нужно оставить. В таких случаях используйте оператор UNION ALL.

Вернемся к нашему примеру с бейсбольными командами и предположим, что Боб Тарли по кличке "Пуля" был продан в середине сезона из команды "Нью-Йорк Янкиз", входящей в Американскую лигу, в "Бруклин Доджерс" из Национальной лиги. А теперь допустим, что в каждой команде за сезон у этого питчера было по восемь игр, во время которых он бессменно подавал мяч. Обычный оператор UNION, продемонстрированный в приведенном выше примере, отбросит одну из двух строк с данными об этом игроке. В результате окажется, что за сезон он полностью провел на подаче мяча только восемь игр, хотя на самом деле таких игр было шестнадцать. Корректные данные можно получить с помощью следующего запроса.

```
SELECT * FROM NATIONAL
UNION ALL
SELECT * FROM AMERICAN ;
```


СОВЕТ

Иногда оператор UNION можно применить к двум таблицам, которые не являются полностью совместимыми для объединения. Если в таблицу, которую вы хотите получить в результате, должны войти не все столбцы, а лишь те, которые есть в обеих исходных таблицах (и при этом они являются совместимыми), то можно использовать оператор UNION CORRESPONDING. В этом случае запросы будет работать только с заданными столбцами, и только они войдут в результирующую таблицу.

Оператор UNION CORRESPONDING

Бейсбольные статистики собирают совершенно разную информацию по питчерам и аутфилдерам (игрокам во внешней части поля). Но в обоих случаях фиксируются следующие данные: имя, фамилия игрока, количество стандартных аутов, ошибки и доля принятых мячей. Естественно, по аутфилдерам не собирают данные о выигрышах/проигрышах, остановленных мячах и прочие сведения, относящиеся только к подаче мяча. Но все равно, чтобы получить общую информацию об эффективности игры в защите, можно применить оператор UNION, который извлечет данные из двух таблиц: OUTFIELDER (аутфилдер) и PITCHER (питчер).

```
SELECT *
  FROM OUTFIELDER
UNION CORRESPONDING
   (FirstName, LastName, Putouts, Errors, FieldPct)
SELECT *
  FROM PITCHER ;
```

В результате получится таблица, в которой для каждого питчера или аутфилдера указаны имя и фамилия, количество аутов, число ошибок и доля принятых мячей. Здесь, как и при использовании простого оператора `UNION`, повторяющиеся строки удаляются. Таким образом, если игрок некоторое время играл во внешней части поля, а также (в других играх) был питчером, то при выполнении оператора `UNION CORRESPONDING` часть статистики этого игрока будет потеряна. Чтобы этого не случилось, используйте вариант оператора `UNION ALL CORRESPONDING`.


СОВЕТ

В списке, указанном сразу за ключевым словом `CORRESPONDING`, должны перечисляться имена только тех столбцов, которые существуют в обеих объединяемых таблицах. При отсутствии этого списка подразумевается использование полного списка имен столбцов, одновременно существующих в обеих таблицах. Следует учесть, что, если в одну или несколько таблиц будут добавлены новые столбцы, то этот неявно заданный список может измениться. Так что имена столбцов лучше не опускать, а указывать явно.

Оператор INTERSECT

В результате выполнения оператора `UNION` создается таблица, которая будет содержать все строки, существующие в *каждой* из исходных таблиц. Если же нужны только строки, одновременно существующие во *всех* исходных таблицах, то используют оператор `INTERSECT`. Он является SQL-реализацией операции пересечения из реляционной алгебры. Работу этого оператора мы покажем на вымышленном примере, представив, что игрок Боб Тарли был обменян в середине сезона в команду "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, мы увидим только те строки, которые находятся одновременно во всех исходных таблицах.

```
SELECT *
  FROM NATIONAL
INTERSECT
SELECT *
  FROM AMERICAN;

FirstName     LastName     CompleteGames
---------     --------     -------------
Bob           Turley                   8
```

После рассмотрения полученной таблицы становится ясно, что Боб Тарли был единственным питчером, у которого в обеих лигах одинаковое число игр с бессменной подачей мяча. Следует заметить, что, как и в случае с оператором UNION, оператор INTERSECT DISTINCT создает тот же результат, что и оператор INTERSECT без дополнительного ключевого слова. В рассмотренном выше 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 указано ключевое слово ALL, то в результирующую таблицу попадут строки-дубликаты. А при использовании ключевого слова CORRESPONDING совместимость для объединения не является обязательным требованием для исходных таблиц, хотя у соответствующих столбцов должны быть одинаковые тип данных и длина.

Вот пример использования оператора INTERSECT ALL.

```
SELECT *
  FROM NATIONAL
INTERSECT ALL
SELECT *
  FROM AMERICAN;
```

```
FirstName    LastName    CompleteGames
---------    --------    -------------
Bob          Turley                8
Bob          Turley                8
```

Рассмотрим еще один пример. Предположим, что в муниципалитете хранятся данные о мобильных телефонах, используемых полицейскими, пожарными, уборщиками улиц и другими работниками городского хозяйства. Данные обо всех действующих мобильных телефонах заносятся в таблицу PHONES. А данные о телефонах, которые уже не обслуживаются, заносятся в другую таблицу, OUT, имеющую такую же структуру, как и таблица PHONES. Информация ни по одному из телефонов не может одновременно содержаться в двух таблицах. Выполнив оператор INTERSECT, можно проверить, не произошло ли такое ненужное дублирование данных.

```
SELECT *
  FROM PHONES
INTERSECT CORRESPONDING (PhoneID)
SELECT *
  FROM OUT ;
```

ЗАПОМНИ!

Если после выполнения оператора INTERSECT CORRESPONDING в результирующей таблице окажутся какие-либо строки, то это означает наличие проблемы. Необходимо проверить все значения столбца PhoneID, которые попали в эту таблицу, ведь мобильный телефон, соответствующий этому идентификатору, либо обслуживается, либо нет. Одновременно и того, и другого не может быть. Обнаружив противоречивые данные, можно выполнить в одной из двух таблиц инструкцию DELETE и тем самым восстановить целостность базы данных

Оператор EXCEPT

Оператор UNION работает с двумя исходными таблицами и возвращает все строки, которые существуют в каждой из них. Другой оператор, INTERSECT, возвращает все строки, которые имеются одновременно и в первой, и во второй таблицах. А оператор EXCEPT (или EXCEPT DISTINCT), наоборот, возвращает все строки, которые имеются в первой таблице, но *отсутствуют* во второй.

Вернемся к примеру с базой данных муниципальных мобильных телефонов. Предположим, партия телефонов была возвращена поставщику для ремонта и как следствие перестала обслуживаться (т.е. их номера были занесены в таблицу OUT). Через некоторое время эти телефоны вернулись из ремонта и

вновь были поставлены на обслуживание. В таблицу PHONES данные об исправленных телефонах занесли, но из таблицы OUT их почему-то забыли удалить. С помощью оператора EXCEPT можно вывести все номера телефонов, находящиеся в столбце PhoneID таблицы OUT, за исключением тех, которые принадлежат уже исправленным телефонам.

```
SELECT *
  FROM OUT
EXCEPT CORRESPONDING (PhoneID)
SELECT *
  FROM PHONES;
```

При выполнении этого запроса возвращаются все строки из таблицы OUT, в которых значения PhoneID не находятся в таблице PHONES.

Табличные объединения

Операторы UNION, INTERSECT и EXCEPT представляют ценность только в таких многотабличных базах данных, которые содержат совместимые для объединения таблицы. Однако во многих случаях приходится извлекать данные из нескольких исходных таблиц, имеющих между собой мало (или даже ничего) общего. Для этого предусмотрены мощные реляционные операторы, объединяющие данные из множества таблиц в одну.

Стандарт SQL поддерживает множество типов объединений. Какое из них лучше всего подходит для конкретной ситуации, зависит от того результата, который требуется получить. В следующих разделах мы подробно рассмотрим каждый из типов объединений.

Простое объединение

Любой многотабличный запрос является разновидностью объединения. Исходные таблицы объединяются в том смысле, что таблица-результат будет включать информацию из всех исходных таблиц. Самую простую операцию объединения двух таблиц реализует инструкция SELECT без предложения WHERE. В таком запросе каждая строка из первой таблицы объединяется с каждой строкой из второй. В результате получается таблица, которая представляет собой декартово произведение двух исходных таблиц. Количество строк в результирующей таблице равно числу строк в первой, умноженному на число строк во второй исходной таблице.

Предположим, что вы работаете менеджером по персоналу, и в ваши обязанности входит поддержка данных о сотрудниках вашей компании. Такая информация, как домашний адрес и номер телефона, не является особо

засекреченной. Однако доступ к таким данным, как зарплата (и некоторым другим), должен предоставляться только тем, у кого есть соответствующее разрешение. Поэтому, чтобы защитить конфиденциальную информацию, вы держите ее в отдельной таблице, защищенной паролем. Рассмотрим таблицы EMPLOYEE и COMPENSATION.

```
EMPLOYEE                              COMPENSATION
--------------------------------      -------------------
EmpID (идентификатор сотрудника)      Employ (сотрудник)
FName (имя)                           Salary (зарплата)
LName (фамилия)                       Bonus (премиальные)
City (город)
Phone (телефон)
```

Заполним таблицы любыми взятыми для примера данными.

```
EmpID   FName   LName   City     Phone
-----   -----   -----   ------   -----
    1   Whitey  Ford    Orange   555-1001
    2   Don     Larson  Newark   555-3221
    3   Sal     Maglie  Nutley   555-6905
    4   Bob     Turley  Passaic  555-8908

Employ   Salary   Bonus
------   ------   -----
     1    33000   10000
     2    18000    2000
     3    24000    5000
     4    22000    7000
```

Создадим виртуальную результирующую таблицу с помощью следующего запроса.

```
SELECT *
  FROM EMPLOYEE, COMPENSATION ;
```

Вот как выглядит результат.

```
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 — декартовом произведении таблиц EMPLOYEE и COMPENSATION — много избыточных данных. Хуже того, в большинстве объединенных строк (каждая строка из таблицы EMPLOYEE объединена с каждой строкой из таблицы COMPENSATION) вообще нет смысла. Единственными содержательными строками в этой таблице являются те, в которых число из столбца EmpID, взятого из таблицы EMPLOYEE, равно числу из столбца Employ, взятого из таблицы COMPENSATION. В этих строках имя, фамилия и адрес сотрудника объединены с его же выплатами.

Если вы пытаетесь извлечь из множества таблиц базы данных полезную информацию, то учтите, что декартово произведение, созданное с помощью простого объединения, почти никогда не даст вам полезный результат, хотя это почти всегда первый шаг в нужном направлении. Отфильтровать ненужные строки можно с помощью ограничений, указываемых в предложении WHERE.

Объединение по равенству

Объединение по равенству — это простое объединение с предложением WHERE, условие которого означает, что значение из одного столбца первой таблицы должно быть равно значению из соответствующего столбца второй.

Если применить такое объединение к таблицам из предыдущего примера, то можно получить более содержательный результат.

```
SELECT *
  FROM EMPLOYEE, COMPENSATION
 WHERE EMPLOYEE.EmpID = COMPENSATION.Employ ;
```

Вот как выглядит итоговая таблица.

```
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, поскольку столбец EmpID дублирует значения столбца 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 получим 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
```

Данная таблица отображает именно ту информацию, которую мы хотим знать, без каких-либо лишних данных. Впрочем, писать сам запрос было несколько утомительно. Чтобы избежать двусмысленности, в именах столбцов приходилось явно указывать имена таблиц. Единственная польза от этого — тренировка пальцев.

Можно сократить объем вводимого кода, используя псевдонимы. *Псевдоним* — это альтернативное, 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` — псевдоним таблицы `EMPLOYEE`, а `C` — псевдоним таблицы `COMPENSATION`. Действие псевдонима ограничено инструкцией, в которой он определен. Объявив псевдоним в предложении `FROM`, его необходимо использовать в пределах данной инструкции, причем в одной инструкции нельзя одновременно использовать и длинную форму имени таблицы, и ее псевдоним.

СОВЕТ

Смешивание полных имен с псевдонимами привело бы к путанице. Рассмотрим следующий пример.

```
SELECT T1.C, T2.C
  FROM T1 T2, T2 T1
 WHERE T1.C > T2.C ;
```

В данном примере для таблицы `T1` используется псевдоним `T2`, а для таблицы `T2` — `T1`. Конечно, такой выбор псевдонимов неразумен, однако формально он не противоречит никаким правилам. Если допустить возможность совместного использования полных имен и псевдонимов, то в данном примере невозможно определить, о какой таблице идет речь.

Предыдущий пример с псевдонимами эквивалентен следующей инструкции SELECT без них.

```
SELECT T2.C, T1.C
  FROM T1 , T2
 WHERE T2.C > T1.C ;
```

Стандарт SQL позволяет объединять больше двух таблиц. Их максимальное количество зависит от конкретной реализации. Синтаксис, используемый при таких объединениях, аналогичен тому, который используется 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 ;
```

В этой инструкции выполняется объединение трех таблиц по равенству. Эта инструкция извлекает данные, хранящиеся в соответствующих столбцах каждой из таблиц. В результате мы получаем таблицу, в которой будут имена и фамилии продавцов, число совершенных каждым из них продаж и размер премиальных. Теперь начальник отдела продаж быстро поймет, заслужил ли продавец свое вознаграждение.

Если данные о работе продавцов с начала года до текущей даты хранить в отдельной таблице YTD_SALES, то производительность и надежность базы данных будут выше, чем при хранении этих данных в таблице EMPLOYEE. Данные в таблице EMPLOYEE относительно статичные: имя и фамилия человека, его адрес и номер телефона изменяются не слишком часто. И наоборот, данные о продажах за год изменяются довольно часто. Так как в таблице YTD_SALES столбцов меньше, чем в таблице EMPLOYEE, то и обновляется она быстрее. К тому же, если при обновлении объемов продаж не трогать таблицу EMPLOYEE, то уменьшается риск случайного изменения хранящихся в ней данных.

Перекрестное объединение

Оператор CROSS JOIN предназначен для выполнения простого объединения без предложения WHERE. Поэтому инструкцию

```
SELECT *
  FROM EMPLOYEE, COMPENSATION ;
```

можно переписать так:

```
SELECT *
  FROM EMPLOYEE CROSS JOIN COMPENSATION ;
```

Результатом такого объединения является декартово произведение двух исходных таблиц. Операция `CROSS JOIN` редко дает нужный результат сразу, но ее использование может стать первым шагом в цепочке операций, которая в конце концов приведет вас к поставленной цели.

Естественное объединение

Частным случаем объединения по равенству является *естественное объединение*. В предложении `WHERE` из объединения по равенству сравниваются значения заданных столбцов первой и второй исходных таблиц. Сравниваемые столбцы должны иметь одинаковые тип, длину и имя. При естественном объединении на равенство проверяются значения *всех* столбцов из первой таблицы со значениями соответствующих столбцов из второй (под соответствием понимается совпадение типов, длин и имен столбцов из разных таблиц).

Предположим, в таблице `COMPENSATION` из предыдущего примера также имеются столбцы `Salary` и `Bonus`, но столбец `Employ` заменен столбцом `EmpID`. В этом случае можно выполнить естественное объединение таблиц `COMPENSATION` и `EMPLOYEE`. Традиционный синтаксис такого объединения выглядит так.

```
SELECT E.*, C.Salary, C.Bonus
  FROM EMPLOYEE E, COMPENSATION C
 WHERE E.EmpID = C.EmpID ;
```

Этот запрос является частным случаем естественного объединения. Инструкция `SELECT` вернет объединенные строки, для которых `E.EmpID = C.EmpID`. Рассмотрим следующий запрос.

```
SELECT E.*, C.Salary, C.Bonus
  FROM EMPLOYEE E NATURAL JOIN COMPENSATION C ;
```

Данный запрос объединит строки, для которых `E.EmpID = C.EmpID`, `E.Salary = C.Salary` и `E.Bonus = C.Bonus`. Результирующая таблица будет содержать только те строки, в которых значения *всех* соответствующих столбцов совпадают. В этом примере результаты обоих запросов окажутся одинаковыми, поскольку таблица `EMPLOYEE` не содержит ни столбца `Salary`, ни столбца `Bonus`.

Условное объединение

Условное объединение подобно объединению по равенству, но проверяемое условие не обязательно является условием равенства (хотя и оно не исключается). Проверяемым условием может быть любой правильно составленный

предикат. Если условие выполняется, то соответствующая строка становится частью результирующей таблицы. Синтаксис условного объединения немного отличается от того, который мы встречали до сих пор. Это отличие состоит в том, что условие содержится в предложении ON, а не WHERE.

Предположим, бейсбольному статистику нужно знать, какие питчеры из Национальной лиги провели полностью на подаче столько же игр, сколько это сделал хотя бы один питчер Американской лиги. В общем случае ответ на такой вопрос даст объединение по равенству, но можно применить и условное объединение.

```
SELECT *
  FROM NATIONAL JOIN AMERICAN
    ON NATIONAL.CompleteGames = AMERICAN.CompleteGames ;
```

Объединение по именам столбцов

Объединение по именам столбцов напоминает естественное объединение, только оно более гибкое. При естественном объединении проверяется равенство значений из *всех* одноименных столбцов в исходных таблицах. В объединении по именам столбцов можно выбирать, какие именно одноименные столбцы должны проверяться. При желании можно выбрать все столбцы с одинаковыми именами, фактически превращая объединение по именам столбцов в естественное, но можно выбрать и меньшее их количество. Таким образом, вы получаете больший контроль над тем, какие строки из перекрестного произведения попадут в результирующую таблицу.

Скажем, вы изготовитель шахматных наборов и заносите данные о готовых белых и черных фигурах в специальные таблицы, которые называются WHITE и BLACK соответственно. В каждой таблице имеются поля Piece (фигура), Quant (количество) и Wood (дерево). В этих таблицах хранятся такие данные.

```
WHITE                          BLACK
-----                          -----
Piece    Quant    Wood         Piece    Quant    Wood
-----    -----    ----         -----    -----    ----
Король     502     Дуб         Король     502    Эбен
Ферзь      398     Дуб         Ферзь      397    Эбен
Ладья     1020     Дуб         Ладья     1020    Эбен
Слон       985     Дуб         Слон       985    Эбен
Конь       950     Дуб         Конь       950    Эбен
Пешка      431     Дуб         Пешка      453    Эбен
```

Для каждой разновидности фигур — короля, ферзя, ладьи, слона, коня, пешки, — изготавливаемых из дуба и эбена (черного дерева), количество белых и черных фигур должно быть равным. Если это равенство нарушено, значит,

некоторые фигуры были либо потеряны, либо украдены, и, следовательно, нужно улучшить условия хранения товара.

При естественном объединении проверяется равенство значений во всех одноименных столбцах. Применив естественное объединение, мы получили бы пустую таблицу, потому что в таблице WHITE нет таких строк, в которых значение в столбце Wood совпало бы с каким-нибудь значением из столбца Wood таблицы BLACK. Пустая таблица не позволит определить, пропало что-нибудь или нет. Здесь подойдет объединение по именам столбцов, в котором столбец Wood исключается из рассмотрения.

```
SELECT *
  FROM WHITE JOIN BLACK
 USING (Piece, Quant) ;
```

В результате объединения получим таблицу только с теми строками, в которых количество белых и черных фигур на складе совпадает.

```
Piece    Quant    Wood      Piece    Quant    Wood
-----    -----    ----      -----    -----    ----
Король     502     Дуб      Король     502    Эбен
Ладья     1020     Дуб      Ладья     1020    Эбен
Слон       985     Дуб      Слон       985    Эбен
Конь       950     Дуб      Конь       950    Эбен
```

Внимательный читатель должен заметить, что из списка пропали ферзь и пешка, — признак того, что каких-то из этих фигур не хватает.

Внутреннее объединение

Вы, наверное, считаете, что объединения — довольно эзотерические операции, и для их применения нужен недюжинный интеллект. Возможно, вы еще слышали о загадочном *внутреннем объединении*, которое является квинтэссенцией реляционных операций.

Это шутка! Во внутренних объединениях вовсе нет ничего мистического. В действительности внутренними являются все объединения, о которых говорилось ранее. Объединение по именам столбцов из последнего примера можно представить в виде внутреннего объединения (INNER JOIN), если воспользоваться следующим синтаксисом.

```
SELECT *
  FROM WHITE INNER JOIN BLACK
 USING (Piece, Quant) ;
```

Результат при этом получится тот же самый.

Внутреннее объединение было так названо, чтобы его можно было отличить от внешнего. При выполнении внутреннего объединения из таблицы-

результата выбрасываются все строки, для которых (в соответствующих столбцах) нет совпадающих элементов в обеих исходных таблицах; при внешнем объединении строки с несовпадающими элементами остаются. Вот и все различия, и никакой метафизики.

Внешнее объединение

При объединении двух таблиц в первой из них (назовем ее левой) могут существовать строки, которые не имеют соответствующих строк (т.е. строк с совпадающими элементами в заданных столбцах) во второй (правой) таблице. И наоборот, в правой таблице могут находиться строки, которые не имеют соответствующих строк в левой. При выполнении внутреннего объединения этих таблиц все несоответствующие строки из результата удаляются. В то же время при внешнем объединении такие строки остаются. Внешнее объединение бывает трех видов: левое, правое и полное.

Левое внешнее объединение

В запросе, включающем в себя объединение, *левой* таблицей считается та, которая в инструкции запроса предшествует ключевому слову `JOIN`, а *правой* — та, которая следует за ним. При левом внешнем объединении несоответствующие строки, существующие в левой таблице, сохраняются в результирующей таблице, а несоответствующие в правой — удаляются из нее.

Чтобы понять работу внешних объединений, представьте себе корпоративную базу данных, в которой хранятся записи о сотрудниках компании, ее отделах и филиалах. Примеры данных этой компании приведены в табл. 11.1–11.3.

Таблица 11.1. Таблица `LOCATION` (филиал)

LOCATION_ID (идентификатор филиала)	*CITY* (город)
1	Барнаул
3	Тюмень
5	Челябинск

Таблица 11.2. Таблица `DEPT` (отдел)

DEPT_ID (идентификатор отдела)		*NAME* (название)
21	1	Продажи
24	1	Администрация

Окончание табл. 11.2

DEPT_ID (идентификатор отдела)		*NAME* (название)
27	5	Сервис
29	5	Склад

Таблица 11.3. Таблица EMPLOYEE (сотрудник)

(идентификатор сотрудника)		(фамилия)
61	24	Калашников
63	27	Макаров

Теперь предположим, что вам нужно получить данные о том, в каком отделе и филиале работает каждый сотрудник компании. Эту задачу можно выполнить с помощью объединения по равенству.

```
SELECT *
  FROM LOCATION L, DEPT D, EMPLOYEE E
 WHERE L.LocationID = D.LocationID
   AND D.DeptID = E.DeptID ;
```

Результат выполнения запроса будет таким.

```
1  Барнаул     24  1  Администрация   61  24  Калашников
5  Челябинск   27  5  Сервис         63  27  Макаров
```

Полученная в результате таблица содержит все данные обо всех сотрудниках (с указанием отдела и филиала). Поскольку каждый сотрудник компании работает в каком-либо филиале и в одном из отделов, то для этого примера как раз и подходит объединение по равенству.

А теперь предположим, что вам нужны данные о филиалах, а также связанные с ними данные об отделах и сотрудниках. Это уже совершенно другая задача, потому что в филиале может и не существовать отделов. Для получения таких данных мы используем 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 собираются из трех таблиц. Сначала объединяются таблицы LOCATION и DEPT, а затем получившаяся таблица объединяется с таблицей EMPLOYEE. Даже если строки из таблицы, расположенной слева

от оператора LEFT OUTER JOIN, и не имеют соответствующих строк в таблице, расположенной справа, они все равно входят в результат. Таким образом, при первом объединении в результат войдут все филиалы, даже без отделов. А при втором объединении — все отделы, даже без персонала. И вот какой получится результат.

```
1  Барнаул     24    1     Администрация  61    24    Калашников
5  Челябинск   27    5     Сервис         63    27    Макаров
3  Тюмень      NULL  NULL  NULL           NULL  NULL  NULL
5  Челябинск   29    5     Склад          NULL  NULL  NULL
1  Барнаул     21    1     Продажи        NULL  NULL  NULL
```

Как видите, здесь первые две строки такие же, как и в результирующей таблице предыдущего примера. А третья строка (3 Тюмень) в столбцах, относящихся к отделам и сотрудникам, содержит пустые значения, потому что в Тюмени нет никаких отделов и никто из сотрудников там постоянно не работает. В четвертой (5 Челябинск) и пятой (1 Барнаул) строках находятся данные о складе и об отделе продаж, но в столбцах этих строк, относящихся к сотрудникам, находятся пустые значения, так как в этих двух отделах постоянного персонала нет. Это внешнее объединение сообщает всю ту же самую информацию, что и объединение по равенству, но дополнительно предоставляет следующие данные:

- обо всех филиалах компании (с отделами или без таковых);
- обо всех отделах компании (с персоналом или без него).

Нельзя быть уверенным в том, что строки в последнем примере будут выведены в нужном вам порядке, так как он зависит от конкретной реализации. Чтобы вывести строки в требуемом порядке, вставьте в инструкцию SELECT предложение ORDER BY.

```
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

Поскольку левого внутреннего объединения не существует, левое внешнее объединение называют короче — *левое объединение* (LEFT JOIN).

Правое внешнее объединение

Готов поклясться, вы уже знаете, как ведет себя правое внешнее объединение. И вы правы! *Правое внешнее объединение* сохраняет несоответствующие

строки, взятые из правой таблицы, но удаляет несоответствующие строки, взятые из левой. Правое внешнее объединение можно использовать с теми же таблицами нашего примера и получить тот же результат. Но для этого необходимо изменить 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 запроса первое объединение создает таблицу, в которой находятся все отделы, с персоналом или без него. А второе объединение создает таблицу со всеми филиалами независимо от наличия в них отделов.


СОВЕТ

Поскольку правого внутреннего объединения не существует, его можно называть просто *правым объединением* (RIGHT JOIN).

Полное внешнее объединение

Полное внешнее объединение вобрало в себя свойства левого и правого внешних объединений. После его выполнения в результирующей таблице остаются несоответствующие строки как из левой, так и из правой таблиц. Рассмотрим теперь общий вариант корпоративной базы данных, которая использовалась в предыдущих примерах. В этой базе могут находиться:

- филиалы без отделов;
- отделы без привязки к филиалам;
- отделы без сотрудников;
- сотрудники без привязки к отделам.

Чтобы отобразить все филиалы, отделы и сотрудников независимо от того, имеют ли они соответствующие строки в других таблицах, используйте полное внешнее объединение (FULL OUTER JOIN).

```
SELECT *
  FROM LOCATION L FULL OUTER JOIN DEPT D
    ON (L.LocationID = D.LocationID)
  FULL OUTER JOIN EMPLOYEE E
    ON (D.DeptID = E.DeptID) ;
```


СОВЕТ

Поскольку полного внутреннего объединения не существует, полное внешнее объединение называют просто *полным объединением* (FULL JOIN).

Объединение со слиянием

В отличие от других видов объединений, *объединение со слиянием* (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, содержащую 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 пустые значения. Аналогично столбцы, взятые из правой исходной таблицы, содержат все строки этой исходной таблицы, в которых все столбцы, взятые из левой исходной таблицы, содержат пустые значения. Таким образом, таблица, получившаяся в результате объединения со слиянием, содержит все столбцы из обеих исходных таблиц, причем число ее строк равно суммарному количеству строк обеих исходных таблиц.

В большинстве случаев объединение со слиянием можно использовать лишь как промежуточный результат: в нем слишком большое количество пустых значений. Однако вы сможете извлечь из этого объединения полезную информацию, если будете использовать его с выражением COALESCE (см. главу 9).

Предположим, вы работаете в компании, которая проектирует и производит ракеты, предназначенные для экспериментальных запусков. Работа ведется одновременно над несколькими проектами. Под вашим руководством работают инженеры-проектировщики, которые являются специалистами в различных областях. Как менеджера вас интересует, какие инженеры (и с какой квалификацией) работали над какими проектами. В настоящий момент эти данные разбросаны по трем таблицам: EMPLOYEE (сотрудник), PROJECTS (проекты) и SKILLS (квалификация).

В таблице EMPLOYEE хранятся данные о сотрудниках, и ее первичным ключом является столбец EMPLOYEE.EmpID. Каждый проект, над которым работал сотрудник, занимает одну строку в другой таблице — PROJECTS. Внешний ключ этой таблицы, PROJECTS.EmpID, ссылается на таблицу EMPLOYEE. В таблице SKILLS для каждого сотрудника перечислены виды деятельности, в которых он имеет квалификацию. Внешний ключ этой таблицы, SKILLS.EmpID, тоже ссылается на таблицу EMPLOYEE.

В таблице EMPLOYEE для каждого сотрудника отведена 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. 11.4–11.6.

Таблица 11.4. Таблица EMPLOYEE

EmpID (идентификатор сотрудника)	Name (фамилия)
1	Федченко
2	Фрид
3	Томенко

Таблица 11.5. Таблица PROJECTS

ProjectName (название проекта)	EmpID
Конструкция ракеты X-63	1
Конструкция ракеты X-64	1
Система наведения X-63	2
Система наведения X-64	2
Телеметрия X-63	3
Телеметрия X-64	3

Таблица 11.6. Таблица SKILLS

Skill (квалификация)	EmpID
Механическое проектирование	1
Расчеты аэродинамической нагрузки	1
Проектирование аналоговых устройств	2
Проектирование гироскопов	2
Проектирование цифровых устройств	3
Проектирование РЛС	3

Как видно из этих таблиц, инженер Федченко работал над проектами конструкций 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 операцию объединения по равенству.

```
SELECT *
  FROM EMPLOYEE E, PROJECTS P, SKILLS S
 WHERE E.EmpID = P.EmpID
   AND E.EmpID = S.EmpID ;
```

Эту же операцию можно представить в виде внутреннего объединения, используя следующий синтаксис.

```
SELECT *
  FROM EMPLOYEE E INNER JOIN PROJECTS P
    ON (E.EmpID = P.EmpID)
 INNER JOIN SKILLS S
    ON (E.EmpID = S.EmpID) ;
```

Оба варианта дают одинаковый результат, показанный в табл. 11.7.

Таблица 11.7. Результаты выполнения внутреннего объединения

E.EmpID	Name	P.EmpID	ProjectName	S.EmpID	Skill
1	Федченко	1	Конструкция ракеты X-63	1	Механическое проектирование
1	Федченко	1	Конструкция ракеты X-63	1	Расчеты аэродинамической нагрузки
1	Федченко	1	Конструкция ракеты X-64	1	Механическое проектирование
1	Федченко	1	Конструкция ракеты X-64	1	Расчеты аэродинамической нагрузки
2	Фрид	2	Система наведения X-63	2	Проектирование аналоговых устройств
2	Фрид	2	Система наведения X-63	2	Проектирование гироскопов
2	Фрид	2	Система наведения X-64	2	Проектирование аналоговых устройств
2	Фрид	2	Система наведения X-64	2	Проектирование гироскопов

Окончание табл. 11.7

E.EmpID	Name	P.EmpID	ProjectName	S.EmpID	Skill
3	Томенко	3	Телеметрия X-63	3	Проектирование цифровых устройств
3	Томенко	3	Телеметрия X-63	3	Проектирование РЛС
3	Томенко	3	Телеметрия X-64	3	Проектирование цифровых устройств
3	Томенко	3	Телеметрия X-64	3	Проектирование РЛС

Такое расположение данных вряд ли можно назвать удовлетворительным. Идентификатор сотрудника повторяется три раза в каждой строке. Для каждого сотрудника его проекты и виды квалификации указываются дважды. Вывод: внутреннее объединение не очень подходит для решения подобных задач. Более удачный результат можно получить, используя объединение со слиянием.

```
SELECT *
  FROM EMPLOYEE E UNION JOIN PROJECTS P
 UNION JOIN SKILLS S ;
```

Обратите внимание на то, что в объединении со слиянием отсутствует предложение ON. Дело в том, что здесь данные не фильтруются, поэтому в предложении ON нет необходимости. Результат, полученный при выполнении этой инструкции, показан в табл. 11.8.

Таблица 11.8. Результат выполнения оператора UNION JOIN

E.EmpID	Name	P.EmpID	ProjectName	S.EmpID	Skill
1	Федченко	NULL	NULL	NULL	NULL
NULL	NULL	1	Конструкция ракеты X-63	NULL	NULL
NULL	NULL	1	Конструкция ракеты X-64	NULL	NULL
NULL	NULL	NULL	NULL	1	Механическое проектирование
NULL	NULL	NULL	NULL	1	Расчет аэродинамической нагрузки
2	Фрид	NULL	NULL	NULL	NULL

Окончание табл. 11.8

E.EmpID	Name	P.EmpID	ProjectName	S.EmpID	Skill
NULL	NULL	2	Система наведения X-63	NULL	NULL
NULL	NULL	2	Система наведения X-64	NULL	NULL
NULL	NULL	NULL	NULL	2	Проектирование аналоговых устройств
NULL	NULL	NULL	NULL	2	Проектирование гироскопов
3	Томенко	NULL	NULL	NULL	NULL
NULL	NULL	3	Телеметрия X-63	NULL	NULL
NULL	NULL	3	Телеметрия X-64	NULL	NULL
NULL	NULL	NULL	NULL	3	Проектирование цифровых устройств
NULL	NULL	NULL	NULL	3	Проектирование РЛС

Каждая таблица была расширена справа или слева пустыми (NULL) значениями, после чего было выполнено объединение в единую таблицу всех строк, получившихся в результате этого расширения. Порядок строк произвольный и зависит от используемой реализации. Теперь можно представить полученные данные в более приемлемом виде.

Для идентификатора сотрудника в таблице отведены три столбца, но в каждой строке определенным является только один из них. Вид выводимой таблицы можно улучшить, если использовать для этих столбцов выражение COALESCE. Как уже упоминалось в главе 9, это выражение выбирает из переданного ему списка первое непустое значение. В данном случае COALESCE выберет из списка столбцов единственное непустое 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 объединены 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. 11.9.

Таблица 11.9. Результат применения оператора UNION JOIN совместно с выражением COALESCE

ID	Name	ProjectName	Skill
1	Федченко	Конструкция ракеты X-63	NULL
1	Федченко	Конструкция ракеты X-64	NULL
1	Федченко	NULL	Механическое проектирование
1	Федченко	NULL	Расчет аэродинамической нагрузки
2	Фрид	Система наведения X-63	NULL
2	Фрид	Система наведения X-64	NULL
2	Фрид	NULL	Проектирование аналоговых устройств
2	Фрид	NULL	Проектирование гироскопов
3	Томенко	Телеметрия X-63	NULL
3	Томенко	Телеметрия X-64	NULL
3	Томенко	NULL	Проектирование цифровых устройств
3	Томенко	NULL	Проектирование РЛС

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, информация какого типа содержится в конкретной строке. Если столбец ProjectName содержит непустое значение, значит, в строке указан проект, над которым работал сотрудник. А если непустым является столбец Skill, то в строке указана квалификация сотрудника.

Можно немного улучшить результат, если в инструкцию 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, P.*
            FROM PROJECTS) P
UNION JOIN (SELECT "Skill" AS Type, S.*
            FROM SKILLS) S
ORDER BY ID, Type ;
```

В первом предложении `UNION JOIN` таблица `PROJECTS` (см. предыдущий пример) заменена вложенной инструкцией `SELECT`, которая добавляет к столбцам, взятым из этой таблицы, еще один столбец, `P.Type`, с постоянным значением `"Project"` (Проект). Аналогично во втором предложении `UNION JOIN` таблица `SKILLS` заменена другой вложенной инструкцией `SELECT`, которая добавляет к столбцам, взятым из этой таблицы, еще один столбец, `S.Type`, с постоянным значением `"Skill"` (Специализация). В каждой строке значением столбца `P.Type` является или `NULL`, или `"Project"`, а значением столбца `S.Type` — или `NULL`, или `"Skill"`.

Список столбцов во внешней инструкции `SELECT` содержит выражение `COALESCE (P.Type, S.Type) AS Type`, которое из двух столбцов `Type` "сделает" один, причем с таким же именем `Type`. Затем этот новый столбец `Type` указывается в предложении `ORDER BY`, благодаря чему среди строк с одинаковыми значениями `ID` первыми будут стоять строки с названиями проектов, а за ними — с квалификационными навыками. Результат показан в табл. 11.10.

Таблица 11.10. Улучшенный результат применения оператора `UNION JOIN` вместе с выражениями `COALESCE`

ID	Name	Type	ProjectName	Skill
1	Федченко	Проект	Конструкция ракеты X-63	NULL
1	Федченко	Проект	Конструкция ракеты X-64	NULL
1	Федченко	Специализация	NULL	Механическое проектирование
1	Федченко	Специализация	NULL	Расчет аэродинамической нагрузки
2	Фрид	Проект	Система наведения X-63	NULL
2	Фрид	Проект	Система наведения X-64	NULL
2	Фрид	Специализация	NULL	Проектирование аналоговых устройств

Окончание табл. 11.10

ID	Name	Type	ProjectName	Skill
2	Фрид	Специализация	NULL	Проектирование гироскопов
3	Томенко	Проект	Телеметрия X-63	NULL
3	Томенко	Проект	Телеметрия X-64	NULL
3	Томенко	Специализация	NULL	Проектирование цифровых устройств
3	Томенко	Специализация	NULL	Проектирование РЛС

Полученная в результате таблица представляет собой отчет — причем довольно удобный для чтения — об опыте участия в проектах и о специализации всех сотрудников, перечисленных в таблице EMPLOYEE.

Если учесть многообразие существующих операций объединения (JOIN), то связывание данных из разных таблиц не должно создавать проблем, какой бы ни была структура этих таблиц. Поверьте, если в вашей базе хранятся какие-то исходные данные, в SQL найдется достаточно средств, чтобы извлечь их оттуда и представить в удобном для чтения виде.

Предложения ON и WHERE

Назначение предложений ON и WHERE в объединениях разных видов может оказаться недостаточно понятным. Попробуем прояснить эту ситуацию.

- Предложение ON является частью внутренних, левых, правых и полных объединений. В перекрестных объединениях и объединениях со слиянием предложения ON нет, поскольку никакой фильтрации данных не происходит.
- Во внутреннем объединении предложения ON и WHERE логически эквивалентны; с их помощью можно указать одно и то же условие.
- Предложение ON во внешних (левых, правых и полных) объединениях отличается от предложения WHERE. Последнее всего-навсего фильтрует строки, возвращаемые предложением FROM. Строки, отбракованные фильтром, не попадают в результат. А предложение ON, используемое во внешнем объединении, вначале фильтрует строки перекрестного произведения, а затем включает в результат отбракованные строки, расширенные пустыми значениями.

Глава 12

Вложенные запросы

В ЭТОЙ ГЛАВЕ...

- Извлечение данных из нескольких таблиц с помощью одной инструкции SQL
- Сравнение значения из одной таблицы с набором значений из другой
- Использование инструкции SELECT для сравнения значения из одной таблицы со значением из другой
- Сравнение значения из одной таблицы со всеми соответствующими значениями из другой
- Создание коррелированных подзапросов
- Использование подзапросов для определения строк, подлежащих обновлению, удалению или вставке

Одним из лучших способов защиты целостности данных является исключение аномалий модификации (см. главу 5) за счет нормализации базы данных. *Нормализация* подразумевает разбиение одной таблицы на несколько других по тематическому признаку. Например, нецелесообразно, чтобы в одной таблице содержались данные и о товарах, и о покупателях, несмотря на то что эти данные взаимосвязаны.

Правильно нормализованная база данных содержит множество таблиц. Типичный запрос к такой базе данных обращен как минимум к двум таблицам. В подобных случаях используется оператор объединения JOIN или другие реляционные операторы (UNION, INTERSECT или EXCEPT). Реляционные операторы извлекают данные из нескольких таблиц и сводят их в один результирующий набор. Каждый из перечисленных операторов делает это по-своему.

Еще один способ извлечения данных из нескольких таблиц заключается в использовании вложенных запросов. В SQL *вложенным* называется такой запрос, в котором внешняя инструкция содержит подзапрос. Этот подзапрос сам может быть инструкцией, включающей другой вложенный подзапрос (более низкого уровня). Теоретически число уровней вложенности подзапросов не ограничено, но на практике зависит от конкретной реализации.

Подзапросы обязательно должны быть инструкциями SELECT, но в качестве самой внешней инструкции может выступать INSERT, UPDATE или DELETE.

Поскольку внешний запрос и его подзапрос могут обращаться к разным таблицам, вложенные запросы представляют собой еще один способ извлечения информации из множества таблиц.

Предположим, вы хотите с помощью запроса к базе данных своей компании найти всех руководителей отделов старше пятидесяти лет. Используя операторы объединения, о которых говорилось в главе 11, можно составить 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 ;
```

Здесь D — это псевдоним для таблицы DEPT (отдел), E — для таблицы EMPLOYEE (сотрудник). В таблице EMPLOYEE первичным ключом является столбец идентификатора ID, а в таблице DEPT — столбец идентификатора руководителя отдела ManagerID. Простое объединение (перечисление таблиц в предложении FROM) формирует пары строк из связанных таблиц, а предложение WHERE отсеивает все строки, которые не удовлетворяют заданному критерию. Обратите внимание на то, что в список параметров инструкции SELECT включены столбцы DeptNo (номер отдела) и Name (название отдела) из таблицы DEPT, а также столбцы Age (возраст) и Name (фамилия) из таблицы EMPLOYEE.

Теперь предположим, что вас интересуют строки, отобранные по тому же условию, но только со столбцами из таблицы DEPT. Другими словами, нас интересуют отделы, начальники которых старше пятидесяти лет, и при этом нам совершенно безразличны сами личности (их имена и точный возраст). В этом случае вместо объединения можно использовать подзапрос.

```
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 мы видим два новых элемента: ключевое слово EXISTS и выражение SELECT * в предложении WHERE внешней инструкции SELECT. Внутренняя инструкция SELECT является подзапросом, а ключевое слово EXISTS — одним из предикатов, используемых вместе с подзапросами.

Назначение подзапросов

Подзапросы располагаются в предложении WHERE внешней инструкции. Их роль состоит в задании для этого предложения условия отбора. Разные виды подзапросов дают разные результаты. Одни подзапросы создают список значений, которые затем передаются в качестве входных данных для внешней инструкции. Другие получают единственное значение, которое затем проверяется внешней инструкцией с помощью оператора сравнения. Существуют также подзапросы, возвращающие логические значения.

Вложенные запросы, возвращающие наборы строк

Предположим, вы работаете в фирме, которая занимается сборкой компьютерных систем. В вашей компании на основе приобретаемых комплектующих собирают системы (изделия), которые затем продают другим компаниям и правительственным организациям. Производственную информацию вы храните в реляционной базе данных. Она состоит из множества таблиц, но сейчас нас интересуют только три: PRODUCT, COMP_USED и COMPONENT. В таблице PRODUCT содержится список всех стандартных изделий, выпускаемых вашей фирмой (табл. 12.1), а в таблице COMPONENT — список всех сборочных компонентов, необходимых для выпуска ваших изделий (табл. 12.2). Таблица COMP_USED содержит данные о том, какие компоненты входят в состав каждого изделия (табл. 12.3).

Таблица 12.1. Таблица PRODUCT

Столбец	Тип	Ограничение
Model (модель)	Char (6)	PRIMARY KEY
ProdName (название изделия)	Char (35)	
ProdDesc (описание изделия)	Char (31)	
ListPrice (цена)	Numeric (9, 2)	

Таблица 12.2. Таблица COMPONENT

Столбец	Тип	Ограничение
CompID (идентификатор компонента)	CHAR (6)	PRIMARY KEY
CompType (тип компонента)	CHAR (10)	
CompDesc (описание компонента)	CHAR (31)	

Таблица 12.3. Таблица COMP_USED

Столбец	Тип	Ограничение
Model (модель)	CHAR (6)	FOREIGN KEY (для PRODUCT)
CompID (идентификатор компонента)	CHAR (6)	FOREIGN KEY (для COMPONENT)

Любой компонент может использоваться в нескольких изделиях, а каждое изделие может состоять из множества компонентов (отношение "многие ко многим"). Такая ситуация способна привести к нарушению целостности данных. Чтобы этого не случилось, была создана таблица COMP_USED, связывающая таблицы COMPONENT и PRODUCT. Любой компонент может быть упомянут во многих строках таблицы COMP_USED, но в каждой строке этой таблицы указан только один компонент (отношение "один ко многим"). Аналогично и изделие (модель) может быть упомянуто во многих строках таблицы COMP_USED, но в каждой строке этой таблицы указывается только одно изделие (еще одно отношение "один ко многим"). С помощью связующей таблицы COMP_USED сложное отношение "многие ко многим" разбивается на два относительно простых отношения "один ко многим". Такое упрощение отношений может служить одним из примеров нормализации.

Подзапросы, вводимые предикатом IN

Одна из разновидностей вложенных запросов работает по принципу сравнения одиночного значения с набором значений, возвращаемым инструкцией SELECT. Для этого используется предикат IN, имеющий следующий синтаксис.

```
SELECT список_столбцов
  FROM таблица
 WHERE выражение IN (подзапрос) ;
```

Если значение выражения, заданного в предложении WHERE, есть в списке, полученном при выполнении подзапроса, заданного в скобках после ключевого

слова IN, то предложение WHERE возвращает логическое значение True. И тогда из каждой обработанной подзапросом строки заданной таблицы в результирующую таблицу добавляются значения столбцов, перечисленных в списке. Таблица, указанная во внешнем запросе, может совпадать с таблицей, которая используется в подзапросе, или же это могут быть совершенно разные таблицы.

Чтобы показать, как работает подобный запрос, воспользуемся базой данных вымышленной компании "Мегасервис". Предположим, в компьютерной отрасли возник дефицит мониторов. Под вопросом оказывается выпуск тех изделий, в состав которых должны входить мониторы. Вы, конечно же, хотите знать, что это за изделия. Тогда введите следующий запрос.

```
SELECT Model
  FROM COMP_USED
 WHERE CompID IN
      (SELECT CompID
         FROM COMPONENT
        WHERE CompType = 'Монитор') ;
```

Вначале SQL-код выполняет запрос самого нижнего уровня, т.е. обрабатывает таблицу COMPONENT, возвращая значения CompID из тех строк, в которых значением столбца CompType является 'Монитор'. В результате формируется список идентификаторов всех мониторов. Затем внешний запрос сравнивает с элементами этого списка значение CompID, взятое из каждой строки таблицы COMP_USED. Если сравнение было успешным, то значение столбца Model из той же строки добавляется в результирующую таблицу, создаваемую внешней инструкцией SELECT. В результате формируется список всех моделей изделий, в состав которых входит монитор. Следующий пример показывает, что получится, если выполнить этот запрос.

```
Model
-----
CX3000
CX3010
CX3020
MB3030
MX3020
MX3030
```

Теперь вам ясно, каких товаров в скором времени не будет на складе. Рекламу этих товаров следует на время свернуть.

Этот вид вложенного запроса предполагает, что в подзапросе задается единственный столбец, и тип данных этого столбца совпадает с типом данных аргумента, указанного перед ключевым словом IN.

СОВЕТ

В программировании важно избегать усложнений. Инструкции, включающие вложенные запросы SELECT, могут быть очень сложными и малопонятными. Чтобы получить работоспособный код, рекомендуется сначала выполнить внутреннюю инструкцию SELECT как отдельный запрос и сравнить полученные результаты с требуемыми. Когда вы убедитесь, что внутренняя инструкция SELECT работает правильно, можете включить ее во внешнюю часть инструкции. Такой подход даст вам больше шансов на то, что весь код будет работать правильно.

Подзапросы, вводимые предикатом NOT IN

Если с помощью предиката IN можно было проверить *наличие* значения в списке, то благодаря предикату NOT IN можно добиться обратного. Запрос с предикатом IN, приведенный в предыдущем разделе, помог руководству фирмы узнать, какие изделия будут недоступны для продажи. Хоть это и ценная информация, на ней много не заработаешь. А вот что действительно нужно знать руководству, так это то, какие товары можно будет активно продавать. Руководство фирмы хочет продвигать именно те товары, в состав которых мониторы *не* входят. Эту информацию можно получить с помощью подзапроса, перед которым стоит предикат NOT IN.

```
SELECT Model
  FROM COMP_USED
 WHERE CompID NOT IN
      (SELECT CompID
         FROM COMPONENT
        WHERE CompType = 'Монитор')) ;
```

В результате получаем следующее.

```
Model
-----
PX3040
PB3050
PX3040
PB3050
```

ЗАПОМНИ!

Стоит обратить внимание на тот факт, что результирующий набор данных содержит дубликаты. Причина повторений следующая. Изделие, собираемое из *нескольких* компонентов, среди которых нет монитора, указывается в *нескольких* строках таблицы COMP_USED, т.е. для каждого компонента будет отведена своя строка. И для каждой такой строки запрос создает отдельную строку в результирующей таблице.

В этом примере получившаяся виртуальная таблица невелика, и нам нетрудно оценить результат. Но на практике такая таблица может состоять из сотен и тысяч строк. Чтобы не путаться, повторяющиеся строки лучше удалить. Сделать это легко: достаточно вставить в запрос ключевое слово DISTINCT. Тогда в результирующую таблицу будут добавляться только строки, которые в нее еще не попали.

```
SELECT DISTINCT Model
  FROM COMP_USED
 WHERE CompID NOT IN
      (SELECT CompID
         FROM COMPONENT
        WHERE CompType = 'Монитор')) ;
```

Как и ожидалось, результат получился следующий.

```
Model
-----
PX3040
PB3050
```

Вложенные запросы, возвращающие одно значение

Нередко перед подзапросом полезно поставить один из шести операторов сравнения (=, <>, <, <=, >, >=). Это можно сделать в случае, когда вычисление выражения, стоящего перед оператором, дает единственное значение, а при выполнении подзапроса, указанного после этого оператора, тоже получается одно значение. Исключением является случай, когда непосредственно после оператора сравнения стоит квантор всеобщности или существования (ALL, ANY или SOME).

Чтобы проиллюстрировать ситуацию, когда вложенный подзапрос возвращает единственное значение, вернемся к базе данных компании "Мегасервис". В ней определена таблица CUSTOMER, содержащая информацию о фирмах, которые покупают изделия "Мегасервис". Кроме того, в базе данных есть еще одна таблица, CONTACT, с личными данными представителей каждой компании-клиента. Структура этих таблиц приведена в табл. 12.4 и 12.5.

Таблица 12.4. Таблица CUSTOMER

Столбец	Тип	Ограничение
CustID (идентификатор покупателя)	INTEGER	PRIMARY KEY
Company (компания)	CHAR (40)	UNIQUE

Окончание табл. 12.4

Столбец	Тип	Ограничение
`CustAddress` (адрес)	`CHAR (30)`	
`CustCity` (город)	`CHAR (20)`	
`CustState` (штат)	`CHAR (2)`	
`CustZip` (почтовый индекс)	`CHAR (10)`	
`CustPhone` (телефон)	`CHAR (12)`	
`ModLevel` (уровень)	`INTEGER`	

Таблица 12.5. Таблица `CONTACT`

Столбец	Тип	Ограничение
`CustID` (идентификатор представителя)	`INTEGER`	`FOREIGN KEY`
`ContFName` (имя)	`CHAR (10)`	
`ContLName` (фамилия)	`CHAR (16)`	
`ContPhone` (телефон)	`CHAR (12)`	
`ContInfo` (дополнительная информация)	`CHAR (50)`	

Предположим, вам нужно просмотреть контактную информацию о компании "Олимпик Лтд.", но вы не помните, какой у этой компании идентификатор в столбце `CustID`. В этом случае можно использовать вложенный запрос такого вида.

```
SELECT *
  FROM CONTACT
 WHERE CustID =
       (SELECT CustID
          FROM CUSTOMER
         WHERE Company = 'Олимпик Лтд.') ;
```

Результат выполнения этого вложенного запроса будет примерно таким.

```
CustID   ContFName   ContLName   ContPhone   ContInfo
------   ---------   ---------   ---------   ---------
   118   Джерри      Эттуотер    576-3456    Специалист по 3D-печати
```

Теперь вы сможете позвонить Джерри Эттуотеру в компанию "Олимпик Лтд." и сообщить ему об акционной распродаже 3D-принтеров.

Если подзапрос используется в сравнении с участием оператора равенства (=), то в результирующем списке, возвращаемом этим подзапросом SELECT, должен находиться *только один столбец* (CustID в данном примере). Более того, подзапрос должен вернуть *только одну строку*, чтобы в сравнении (со стороны подзапроса) участвовало только одно значение.

В этом примере предполагается, что в таблице CUSTOMER содержится только одна строка, в которой столбец Company содержит значение 'Олимпик Лтд.'. В инструкции CREATE TABLE, с помощью которой была создана таблица CUSTOMER, для столбца Company было установлено ограничение UNIQUE, и это гарантирует, что подзапрос в предыдущем примере вернет только одно значение (или вообще ни одного). Однако подзапросы, подобные приведенному в примере, обычно работают со столбцами, для которых это ограничение не установлено. В таких случаях, во избежание повторения значений в столбцах, приходится полагаться на достаточно хороший уровень знания содержимого базы данных.

Если окажется, что в столбце Company таблицы CUSTOMER содержится более одного значения 'Олимпик Лтд.', то при выполнении подзапроса возникнет ошибка.

С другой стороны, если в таблице CUSTOMER не зарегистрирована компания "Олимпик Лтд.", то подзапрос вернет значение NULL, и результатом сравнения также будет пустое значение. В этом случае итоговая виртуальная таблица будет пустой. Дело в том, что предложение WHERE возвращает только те строки, для которых в результате сравнения было получено значение True, а строки с результатами сравнения False и NULL будут отфильтрованы. Такое может произойти, если, например, по чьей-то ошибке в столбце Company окажется неправильное название, например 'Олимпик Лтд'.

Несмотря на то что в таких выражениях оператор равенства (=) является самым распространенным, в них можно использовать и пять остальных операторов сравнения. Для каждой строки из таблицы, заданной в предложении FROM внешней инструкции, единственное значение, возвращаемое подзапросом, сравнивается с выражением, заданным в предложении WHERE внешней инструкции. Если результатом сравнения является значение True, то текущая строка добавляется в результирующую таблицу.

Если в состав подзапроса будет включена итоговая функция, то он гарантированно вернет единственное значение. (Итоговые функции всегда возвращают единственное значение; см. главу 3.) Естественно, такой подзапрос окажется полезным только тогда, когда нужно получить именно агрегированное значение.

Предположим, что вы — торговый представитель компании "Мегасервис" и, чтобы оплатить неожиданно свалившиеся на вас счета, должны заработать довольно большие комиссионные. У вас не остается иного выхода, кроме как перестать тратить время на мелочи 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 внешняя инструкция работают с одной и той же таблицей. Подзапрос возвращает единственное значение — максимальное значение цены 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, используемый в сравнении с участием оператора, отличного от "равно".

```
SELECT Model, ProdName, ListPrice
  FROM PRODUCT
 WHERE ListPrice <
      (SELECT AVG(ListPrice)
         FROM PRODUCT) ;
```

Подзапрос возвращает единственное значение — среднее значение цен, хранящихся в столбце `ListPrice` таблицы `PRODUCT`. А внешний запрос возвращает строки из той же таблицы, в которых значение столбца `ListPrice` меньше этого среднего значения.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

Первоначально стандарт языка SQL разрешал использовать в сравнении только один подзапрос, который должен был находиться в правой части сравнения. В соответствии со стандартом SQL:1999 подзапросом может быть любой из двух операндов сравнения и даже оба сразу. Более поздние версии стандарта SQL поддерживают такую возможность.

Использование подзапросов вместе с предикатами ALL, SOME и ANY

Есть еще один способ обеспечить выдачу подзапросом единственного значения — поставить перед этим подзапросом квантор всеобщности или существования. В сочетании с оператором сравнения квантор всеобщности `ALL` и

кванторы существования SOME и ANY обрабатывают список, возвращаемый подзапросом, таким образом, что он сводится к единственному значению.

Воздействие этих кванторов на операцию сравнения проиллюстрируем примером, в котором используется база данных из главы 11. 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
Bob        Turley                8

SELECT * FROM AMERICAN

FirstName  LastName  CompleteGames
---------  --------  -------------
Whitey     Ford                 12
Don        Larson               10
Bob        Turley                8
Allie      Reynolds             14
```

Чисто теоретически питчеры с самым большим количеством игр, бессменно проведенных на подаче, должны быть в Американской лиге, поскольку в этой лиге разрешены назначенные хиттеры. Один из способов проверить эту теорию — создать запрос, который возвращает данные обо всех питчерах Американской лиги, бессменно сыгравших на подаче больше игр, чем все питчеры Национальной лиги. Для этого можно сформулировать 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 нужно возвращать только те значения CompleteGames из таблицы AMERICAN, которые больше любого

значения, возвращаемого подзапросом (т.е. больше самого большого значения, возвращаемого подзапросом). В данном случае таким наибольшим значением является 13. В таблице AMERICAN единственной строкой, в которой находится значение, большее 13, является запись Алли Рейнольдса с его четырнадцатью играми, бессменно сыгранными на подаче.

Но что если ваше первоначальное допущение ошибочно? Что если лидером Главной лиги по количеству бессменных игр был все-таки питчер Национальной лиги, несмотря на то что в этой лиге нет правила назначенного хиттера? Если это так, то следующий запрос вернул бы предупреждение об отсутствии строк, удовлетворяющих условиям запроса.

```
SELECT *
  FROM AMERICAN
 WHERE CompleteGames > ALL
      (SELECT CompleteGames FROM NATIONAL) ;
```

Это означало бы, что в Американской лиге нет такого питчера, который бессменно пробыл бы на подаче в течение большего количества игр, чем питчер-рекордсмен Национальной лиги.

Вложенные запросы как средство проверки 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, поэтому ни одна из них не возвращается. Перед подзапросом можно ставить предикаты EXISTS (существует) и NOT EXISTS (не существует). Эти предикаты позволяют узнать, имеются ли в таблице, указанной в подзапросе (в предложении FROM), какие-либо строки, отвечающие условиям его предложения WHERE.


ЗАПОМНИ!

Подзапросы, перед которыми стоит предикат EXISTS или NOT EXISTS, принципиально отличаются от всех тех, о которых говорилось ранее. Во всех предыдущих случаях SQL-код вначале выполняет подзапрос, а затем использует его результат во внешней инструкции. А подзапросы с предикатами EXISTS и NOT EXISTS — это коррелированные (связанные) подзапросы, и выполняются они по-другому.

Коррелированный подзапрос вначале находит таблицу и строку, заданные во внешней инструкции, а затем выполняет подзапрос в той строке его таблицы, которая согласуется с текущей строкой таблицы внешней инструкции.

Подзапрос возвращает одну или несколько строк либо вообще не возвращает ни одной. Если он возвращает хотя бы одну строку, то предикат EXISTS возвращает истинное значение, и внешняя инструкция выполняет свое действие.

В аналогичной ситуации предикат NOT EXISTS оказывается ложным, и внешняя инструкция свое действие *не* выполняет. После обработки строки в таблице внешней инструкции та же операция выполняется со следующей строкой. Это действие повторяется до тех пор, пока не будут обработаны все строки таблицы, указанной во внешней инструкции.

Предикат EXISTS

Допустим, вы работаете продавцом в компании "Мегасервис" и хотите позвонить представителям всех калифорнийских организаций, покупающих ее продукцию. Попробуйте использовать следующий запрос.

```
SELECT *
  FROM CONTACT
 WHERE EXISTS
      (SELECT *
         FROM CUSTOMER
        WHERE CustState = 'CA'
          AND CONTACT.CustID = CUSTOMER.CustID) ;
```

Обратите внимание на ссылку CONTACT.CustID. Она указывает на столбец из таблицы, заданной во внешнем запросе. Значение этого столбца сравнивается с содержимым другого столбца, CUSTOMER.CustID, находящегося в таблице внутреннего запроса. Для каждой строки внешнего запроса вы проверяете внутренний запрос, в котором используется значение столбца CustID из текущей строки таблицы CONTACT, указанной в предложении WHERE внешнего запроса.

При этом происходит следующее.

1. Таблицы CONTACT и CUSTOMER связывает столбец CustID.
2. SQL-код просматривает первую запись таблицы CONTACT, находит в таблице CUSTOMER строку, имеющую то же значение CustID, и проверяет в этой строке значение кода штата, т.е. значение поля CustState.
3. Если CUSTOMER.CustState = 'CA', то в выводимую таблицу добавляется текущая строка таблицы CONTACT.
4. Точно так же обрабатываются и все последующие записи таблицы CONTACT.
5. Поскольку внешний запрос имеет вид SELECT * FROM CONTACT, то возвращаются *все* поля таблицы, содержащей контактные данные представителей калифорнийских организаций, включая фамилии и номера телефонов.

Предикат NOT EXISTS

В предыдущем примере продавец из компании "Мегасервис" хотел узнать имена и номера телефонов представителей всех калифорнийских организаций, покупающих продукцию его компании. Предположим, что другой продавец

работает со всеми остальными штатами, кроме Калифорнии. Данные о 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 результирующую таблицу добавляются только те строки из таблицы CONTACT, для которых подзапрос не возвращает соответствующую строку.

Другие коррелированные подзапросы

Как уже упоминалось в предыдущем разделе, подзапросы, вводимые предикатом IN или оператором сравнения, не обязательно должны быть коррелированными, хотя такой вариант вполне возможен.

Коррелированные подзапросы, вводимые предикатом IN

Ранее уже рассказывалось, каким образом некоррелированный подзапрос можно использовать вместе с предикатом IN. Чтобы понять, как использовать коррелированный подзапрос с тем же предикатом, задайте тот же самый вопрос, который мы ставили при рассмотрении предиката EXISTS: "Какие фамилии и номера телефонов у представителей всех организаций — покупателей продукции «Мегасервис» в Калифорнии?" Ответ можно получить с помощью коррелированного подзапроса, вводимого ключевым словом IN.

```
SELECT *
  FROM CONTACT
 WHERE 'CA' IN
       (SELECT CustState
          FROM CUSTOMER
         WHERE CONTACT.CustID = CUSTOMER.CustID) ;
```

Здесь инструкция оценивает каждую запись таблицы CONTACT. Если значение столбца CustID в текущей записи совпадает со значением столбца CustID таблицы CUSTOMER, то значение CUSTOMER.CustState сравнивается со значением 'CA'. Результатом выполнения подзапроса является список, в котором содержится не более одного элемента. Если этот единственный элемент равен 'CA', то это означает выполнение условия, заданного в предложении WHERE внешнего запроса, и поэтому вся текущая строка из таблицы CONTACT добавляется в результирующую таблицу.

Коррелированные подзапросы, вводимые операторами сравнения

Как будет показано в следующем примере, перед коррелированным подзапросом может стоять также любой из шести операторов сравнения.

Компания "Мегасервис" выплачивает каждому своему продавцу премию, которая зависит от общей суммы продаж за месяц. Чем выше эта сумма, тем выше процент премии. Список процентных ставок для расчета премий хранится в таблице BONUSRATE со столбцами MinAmount (нижняя граница), MaxAmount (верхняя граница) и BonusPct (процентная ставка).

```
MinAmount     MaxAmount     BonusPct
---------     ---------     --------
     0.00      24999.99     0.
 25000.00      49999.99     0.1
 50000.00      99999.99     0.2
100000.00     249999.99     0.3
250000.00     499999.99     0.4
500000.00     749999.99     0.5
750000.00     999999.99     0.6
```

Например, если у продавца ежемесячная сумма продаж попадает в диапазон от 100 тысяч до 249 999,99 доллара, то он получает премию в размере 0,3% от этой суммы.

Продажи записываются в главную таблицу сделок TRANSMASTER.

```
TRANSMASTER
-----------
Столбец         Тип       Ограничение   Описание
------          ----      -----------   --------
TransID         INTEGER   PRIMARY KEY   Идентификатор сделки
CustID          INTEGER   FOREIGN KEY   Идентификатор покупателя
EmpID           INTEGER   FOREIGN KEY   Идентификатор сотрудника
TransDate       DATE                    Дата сделки
NetAmount       NUMERIC                 Облагаемая налогом сумма
Freight         NUMERIC                 Стоимость перевозки
Tax             NUMERIC                 Налог
InvoiceTotal    NUMERIC                 Итоговая сумма счета-фактуры
```

Премии начисляются на основе суммы значений из столбца NetAmount 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 BonusPct
  FROM BONUSRATE
 WHERE MinAmount <=
       (SELECT SUM (NetAmount)
          FROM TRANSMASTER
```

```
      WHERE EmpID = 133)
  AND MaxAmount >=
     (SELECT SUM (NetAmount)
        FROM TRANSMASTER
       WHERE EmpID = 133) ;
```

Этот запрос интересен тем, что в нем содержатся два подзапроса и для объединения двух логических выражений используется логический оператор AND. В подзапросах задействована итоговая функция SUM, которая возвращает единственное значение — общую сумму продаж за месяц для сотрудника с идентификатором 133. Затем это значение сравнивается со значениями в столбцах MinAmount и MaxAmount из таблицы BONUSRATE, и по результату двух сравнений мы находим процентную ставку для расчета премии для этого сотрудника.

Если идентификатор продавца, хранящийся в столбце EmpID, вам не известен, но известна его фамилия, то такой же ответ можно получить, используя более сложный запрос.

```
SELECT BonusPct
  FROM BONUSRATE
 WHERE MinAmount <=
      (SELECT SUM (NetAmount)
         FROM TRANSMASTER
        WHERE EmpID =
             (SELECT EmpID
                FROM EMPLOYEE
               WHERE EmplName = 'Иванов'))
   AND MaxAmount >=
      (SELECT SUM (NetAmount)
         FROM TRANSMASTER
        WHERE EmpID =
             (SELECT EmpID
                FROM EMPLOYEE
               WHERE EmplName = 'Иванов'));
```

В этом примере, чтобы узнать процентную ставку для расчета премии для сотрудника по фамилии Иванов, используется подзапрос, вложенный в другой подзапрос, который, в свою очередь, вложен во внешний запрос. Эта структура работает только тогда, когда вам наверняка известно, что в компании работает единственный сотрудник с такой фамилией. А что делать, если таких сотрудников-однофамильцев несколько? Тогда в предложение WHERE подзапроса самого нижнего уровня нужно добавлять условия, которые обеспечат выбор единственной строки из 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 10, этому предложению обычно предшествует предложение GROUP BY. Предложение HAVING действует как фильтр, ограничивая группы, созданные предложением GROUP BY. Группы, которые не удовлетворяют условию предложения HAVING, в результат не попадут. В этом случае предложение HAVING проверяет условие для каждой группы, созданной предложением GROUP BY.


СОВЕТ

В отсутствие предложения GROUP BY предложение HAVING проверяет условие для всего набора строк, переданного предложением WHERE. В таком случае весь набор считается одной группой. Если же в запросе нет ни предложения WHERE, ни предложения GROUP BY, то условие предложения HAVING проверяется уже для всей таблицы. Рассмотрим пример.

```
SELECT TM1.EmpID
  FROM TRANSMASTER TM1
 GROUP BY TM1.Department
HAVING MAX (TM1.NetAmount) >= ALL
      (SELECT 2 * AVG (TM2.NetAmount)
         FROM TRANSMASTER TM2
        WHERE TM1.EmpID <> TM2.EmpID) ;
```

В этом запросе для одной и той же таблицы используются два псевдонима. Такой подход позволяет получить идентификаторы (EmpID) всех продавцов, у которых размер максимальной сделки как минимум в два раза превысил средний уровень продаж всех остальных продавцов. Работа данного запроса описана ниже.

1. Строки таблицы TRANSMASTER группируются внешним запросом по значениям столбца Department. Это делается с помощью предложений SELECT, FROM и GROUP BY.
2. Образовавшиеся группы фильтруются предложением HAVING. В нем для каждой из групп вычисляется (с помощью функции MAX) максимальное значение из столбца NetAmount для строк этой группы.
3. Внутренний запрос вычисляет удвоенное среднее по значениям NetAmount для всех тех строк таблицы TRANSMASTER, в которых значение столбца EmpID не равно значению этого столбца в текущей группе внешнего запроса.


ЗАПОМНИ!

В последней строке запроса приходится указывать два разных значения идентификатора сотрудника (EmpID). Именно поэтому здесь используются разные псевдонимы для таблицы TRANSMASTER, созданные в предложениях FROM внешнего и внутреннего запросов.

4. Эти псевдонимы используются в операции сравнения, чтобы обратиться как к значению столбца EmpID из текущей строки внутреннего подзапроса (TM2.EmpID), так и к значению того же столбца из текущей группы внешнего запроса (TM1.EmpID).

Инструкции UPDATE, DELETE и INSERT

Предложения WHERE могут входить не только в инструкции SELECT, но и в инструкции UPDATE, DELETE и INSERT. А в этих инструкциях (как и в инструкции SELECT) предложения WHERE также могут содержать подзапросы.

Предположим, например, что компания "Мегасервис" только что заключила соглашение о партнерстве с компанией "Олимпик Лтд.", согласно которому "Мегасервис" задним числом предоставляет последней десятипроцентную скидку на все покупки прошлого месяца. Информацию об этой скидке можно ввести в базу данных, используя инструкцию UPDATE.

```
UPDATE TRANSMASTER
   SET NetAmount = NetAmount * 0.9
 WHERE SaleDate > (CurrentDate - 30) DAY AND CustID =
       (SELECT CustID
          FROM CUSTOMER
         WHERE Company = 'Олимпик Лтд.') ;
```

В инструкции UPDATE можно использовать и коррелированный подзапрос. Предположим, в таблице CUSTOMER существует столбец LastMonthsMax (максимум за последние месяцы), и руководство компании "Мегасервис" хочет предоставить своим клиентам скидку для сделок, размер которых превышает показатель LastMonthsMax.

```
UPDATE TRANSMASTER TM
   SET NetAmount = NetAmount * 0.9
 WHERE NetAmount >
       (SELECT LastMonthsMax
          FROM CUSTOMER C
         WHERE C.CustID = TM.CustID) ;
```

Обратите внимание на то, что этот подзапрос является коррелированным. Дело в том, что предложение WHERE, расположенное в последней строке инструкции, обращается одновременно 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 инструкции UPDATE может обращаться и к обновляемой таблице. Предположим, руководство решило дать десятипроцентную скидку клиентам, купившим товары на сумму более 10 тыс. долларов.

```
UPDATE TRANSMASTER TM1
   SET NetAmount = NetAmount * 0.9
 WHERE 10000 < (SELECT SUM(NetAmount)
                  FROM TRANSMASTER TM2
                 WHERE TM1.CustID = TM2.CustID);
```

Во внутреннем подзапросе для всех строк таблицы TRANSMASTER, относящихся к одному и тому же покупателю, вычисляется (с помощью функции SUM) сумма значений столбца NetAmount. Что это означает? Предположим, в таблице TRANSMASTER к покупателю со значением CustID, равным 37, относятся четыре строки, в которых столбец NetAmount содержит такие значения: 3000, 5000, 2000 и 1000. Тогда для значения CustID, равного 37, сумма значений NetAmount равна 11000.

Следует отметить, что порядок, в котором инструкция UPDATE обрабатывает строки, определяется конкретной реализацией и обычно является непредсказуемым. Этот порядок может зависеть от того, каким образом строки хранятся на диске. Предположим, в имеющейся реализации строки таблицы TRANSMASTER (для значения столбца CustID, равного 37) обрабатываются в следующем порядке: первой — строка со значением NetAmount, равным 3000, затем — строка со значением NetAmount, равным 5000, и т.д. После обновления первых трех строк со значением CustID, равным 37, у них в столбце NetAmount будут такие значения: 2700 (90% от 3000), 4500 (90% от 5000) и 1800 (90% от 2000). А затем, когда в таблице TRANSMASTER выполняется обработка последней строки, в которой значение CustID равно 37, а значение NetAmount равно 1000, значение функции SUM, возвращенное подзапросом, должно быть равно 10000. Это значение получается как сумма новых значений NetAmount из первых трех строк со значением CustID, равным 37, а также старого значения из последней строки, имеющей то же значение CustID. В результате может показаться, что последняя строка для значения CustID, равного 37, не должна обновляться, поскольку сравнение с этим значением функции SUM не будет истинным (10000 не меньше 10000). Но при обращении подзапроса к обновляемой таблице инструкция UPDATE работает уже по-другому.


ЗАПОМНИ!

В инструкции UPDATE при выполнении всех подзапросов используются старые значения обновляемой таблицы. В предыдущей инструкции UPDATE для всех строк с полем CustID, равным 37, подзапрос возвращает одно и то же число 11000, т.е. первоначальное значение функции SUM.

В инструкциях SELECT и UPDATE подзапрос, заданный в предложении WHERE, работает одинаково. То же справедливо и для инструкций DELETE и INSERT. Так, чтобы удалить записи обо всех сделках компании "Олимпик Лтд.", можно использовать следующую инструкцию.

```
DELETE FROM TRANSMASTER
 WHERE CustID =
      (SELECT CustID
         FROM CUSTOMER
        WHERE Company = 'Олимпик Лтд.') ;
```

Как и в случае с инструкцией UPDATE, подзапросы в инструкции DELETE могут быть коррелированными и могут обращаться к изменяемой таблице (из которой в данном случае удаляются строки). Здесь действуют правила, подобные тем, которые используются для подзапросов в инструкции UPDATE. Предположим, вы хотите удалить из таблицы CUSTOMER все строки пользователей, для которых общая сумма значений NetAmount больше 10 тыс. долларов.

```
DELETE FROM TRANSMASTER TM1
 WHERE 10000 < (SELECT SUM(NetAmount)
                  FROM TRANSMASTER TM2
                 WHERE TM1.CustID = TM2.CustID) ;
```

Этот запрос удаляет из таблицы TRANSMASTER все строки, относящиеся к пользователям, сумма покупок которых превышает 10 тыс. долларов. Все обращения к таблице TRANSMASTER, имеющиеся в подзапросе, подразумевают содержимое этой таблицы, которое было перед любыми удалениями, уже выполненными текущей инструкцией. Поэтому даже при удалении из таблицы TRANSMASTER последней строки, в которой значение столбца CustID равно 37, подзапрос все равно выполняется для этой таблицы так, как будто бы не было до этого никаких удалений (и возвращает суммарное значение 11000).


ЗАПОМНИ!

При обновлении, удалении или вставке записей базы данных существует риск, что данные в изменяемой таблице не будут соответствовать данным в других таблицах. Такое несоответствие называется *аномалией изменения* (см. главу 5). Если из таблицы TRANSMASTER удаляются записи, а от нее зависит другая таблица, TRANSDETAIL (подробности сделок), то необходимо удалить соответствующие записи и из второй таблицы. Эта операция называется *каскадным удалением*, поскольку удаление родительской записи должно вызывать каскад удалений связанных с ней дочерних записей, иначе неудаленные дочерние записи становятся "висячими". В данном случае таковыми могли бы оказаться строки описания счетов-фактур, которые были бы обречены на забвение, ведь записей о самих счетах-фактурах больше не существует.

Если ваша реализация SQL не поддерживает каскадное удаление, то вам придется организовать его самостоятельно. В таком случае, прежде чем

удалять записи из родительской таблицы, удалите соответствующие записи из дочерней. И тогда у вас не останется "висячих" записей в дочерней таблице.

Регистрация изменений с помощью конвейерных DML-операций

В предыдущем разделе было показано, как инструкции UPDATE, DELETE или INSERT могут включать вложенную инструкцию SELECT (в предложении WHERE). Стандарт SQL:2011 расширяет возможности использования вложенных инструкций, и теперь любая инструкция манипуляции данными (т.е. инструкция UPDATE, INSERT, DELETE или MERGE) может быть вложена в инструкцию SELECT. Эта новая возможность называется *конвейерной DML-операцией*.

Давайте взглянем на операцию изменения данных в таблице как на переход от *старой* таблицы (до применения операции DELETE, INSERT или UPDATE) к *новой* (после применения операции изменения). Во время выполнения операции по изменению данных создаются вспомогательные (или промежуточные) таблицы, именуемые *дельта-таблицами*. Так, при выполнении операции DELETE создается старая дельта-таблица (OLD TABLE), которая содержит удаляемые строки. При выполнении операции INSERT создается новая дельта-таблица (NEW TABLE), которая содержит вставляемые строки. Следуя этой логике, при выполнении операции UPDATE создается как старая, так и новая дельта-таблица, причем старая должна содержать заменяемые строки, а новая — заменяющие их строки.

С помощью конвейерных DML-инструкций вы сможете получать информацию из дельта-таблиц. Предположим, вы хотите удалить из своей линейки товаров все изделия с номерами (значениями ProductID), лежащими в диапазоне 1000–1399. При этом вы хотите получить точные данные о том, какие именно изделия будут удалены. Для этого можно выполнить следующий код.

```
SELECT Oldtable.ProductID
  FROM OLD TABLE (DELETE FROM Product
                    WHERE ProductID BETWEEN 1000 AND 1399)
    AS Oldtable ;
```

В этом примере ключевые слова OLD TABLE означают, что результат инструкции SELECT берется из старой дельта-таблицы. Результат представляет собой список значений ProductID тех изделий, которые удаляются.

Аналогичным образом можно получить список строк из новой дельта-таблицы, если использовать ключевые слова NEW TABLE для отображения значений ProductID в строках, вставляемых инструкцией INSERT или обновляемых инструкцией UPDATE. Поскольку инструкция UPDATE создает как старую, так и новую дельта-таблицы, с помощью конвейерных DML-операций можно извлечь содержимое обеих дельта-таблиц или любой из них.

Глава 13

Рекурсивные запросы

В ЭТОЙ ГЛАВЕ...

- Концепция рекурсивной обработки
- Определение рекурсивных запросов
- Применение рекурсивных запросов

Стандарт SQL-92 и его более ранние версии часто критиковали за отсутствие *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 возможности SQL существенно возросли. Если используемая вами реализация SQL включает в себя поддержку рекурсии, то вы сможете эффективно решать широкий класс задач. Но поскольку рекурсивные средства не входят в ядро SQL, многие СУБД могут их и не иметь.

Что такое рекурсия

Рекурсия — довольно старая концепция, реализованная в таких языках программирования, как Logo, LISP и C++. В них можно определить *функцию* (набор, состоящий из одной или нескольких команд), которая выполняет заданную операцию. Основная программа обращается к функции с помощью команды, которая называется *вызовом функции*. Если в качестве одной из команд, составляющих функцию, задать вызов самой функции, то это и будет простейшая форма рекурсии.

Для иллюстрации достоинств и недостатков рекурсии приведем простую программу, в которой одна из функций использует рекурсивные вызовы. Эта

программа, написанная на языке C++, рисует на экране спираль, начиная с единичного сегмента, направленного вверх. В ее состав входят три функции.

- Функция `line(n)` чертит отрезок прямой длиной *n* единиц.
- Функция `left_turn(d)` поворачивает "чертежный инструмент" на *d* градусов против часовой стрелки.
- Функция `spiral(segment)` определяется следующим образом.

```
void spiral(int segment)
{
   line(segment)
   left_turn(90)
   spiral(segment + 1)
} ;
```

Если из основной программы вызвать функцию `spiral(1)`, то будут выполнены следующие действия:

- `spiral(1)` чертит отрезок единичной длины, направленный вверх;
- `spiral(1)` выполняет поворот на 90° против часовой стрелки;
- `spiral(1)` вызывает функцию `spiral(2)`;
- `spiral(2)` чертит отрезок, равный по длине двум единичным и направленный влево;
- `spiral(2)` выполняет поворот на 90° против часовой стрелки;
- `spiral(2)` вызывает функцию `spiral(3)`;
- и т.д.

Постепенно благодаря программе появляется спиральная ломаная линия, показанная на рис. 13.1.

Хьюстон, у нас проблема

Здесь ситуация не такая серьезная, как с космическим кораблем "Аполлон-13", когда на пути к Луне разрушился его главный кислородный бак. Но и мы испытываем трудности — наша программа продолжает вызывать саму себя и чертит все более и более длинные отрезки. Программа будет делать так до тех пор, пока компьютер, пытающийся ее выполнить, не исчерпает свои ресурсы и (в лучшем случае) не выведет на экран сообщение об ошибке. А если вам не повезет, то компьютер просто "зависнет".

Сбой недопустим

Такой сценарий развития событий иллюстрирует одну из опасностей, связанных с использованием рекурсии. Функция, которая содержит обращение

к себе самой, вызывает на выполнение свой новый экземпляр, а тот, в свою очередь, вызывает еще один, и так до бесконечности. Обычно такое развитие событий нас не устраивает.

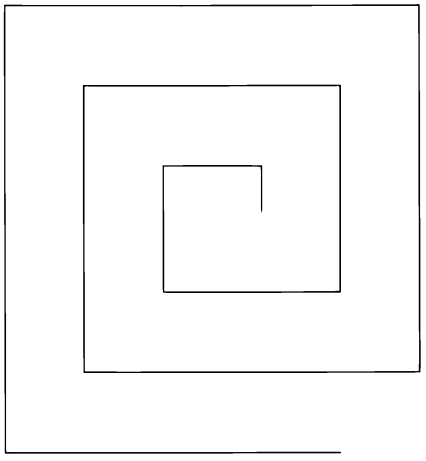


Рис. 13.1. Результат вызова функции `spiral(1)`

Чтобы решить проблему, программисты помещают в рекурсивную функцию *условие завершения*, т.е. устанавливают предел того, насколько глубоко должна зайти рекурсия. В результате программа выполняет нужные действия до заданного предела, а затем корректно завершается. Условие завершения можно поместить и в нашу программу рисования спирали, что позволит сберечь ресурсы компьютера и избежать головокружения у программистов.

```
void spiral2(int segment)
{
   if (segment <= 10)
   {
      line(segment)
      left_turn(90)
      spiral2(segment + 1)
   }
} ;
```

После вызова `spiral2(1)` функция выполняет свои команды, а затем рекурсивно вызывает саму себя (`spiral2(segment + 1)`) до тех пор, пока значение `segment` не превысит 10. Как только значение `segment` станет равным 11, выражение `if (segment <= 10)` вернет значение `False`, и код, находящийся во внутренних скобках, будет пропущен. Управление снова будет передано предыдущему вызову функции `spiral2()`, а оттуда постепенно вернется к первому вызову, после чего программа завершит свою работу. Вся последовательность получившихся вызовов функций и возвратов показана на рис. 13.2.

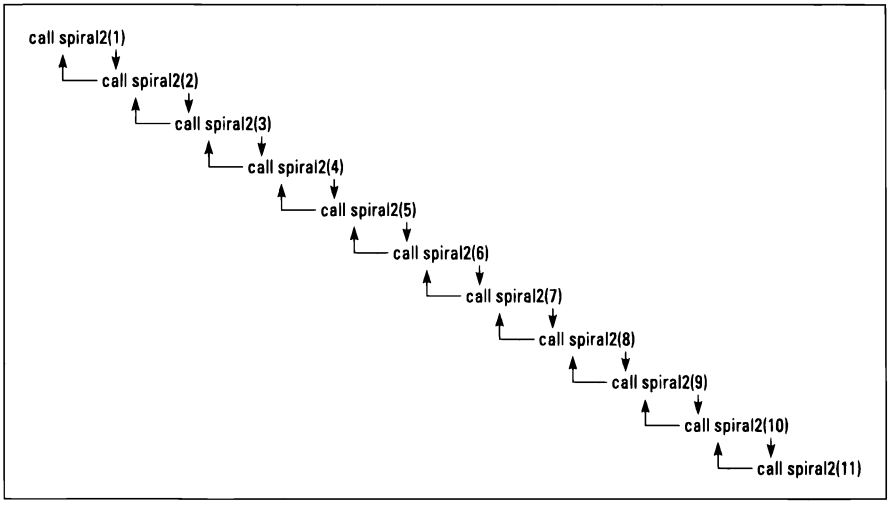


Рис. 13.2. "Спуск" и "подъем" по рекурсивным вызовам

Каждый раз, когда функция вызывает саму себя, она еще на один уровень "погружается вниз" относительно основной программы — места начала операции. Чтобы основная программа продолжила работу, последняя итерация должна вернуть управление предпоследней — той, которая ее вызвала. Предпоследняя итерация обязана поступить точно так же, и этот процесс продолжается до тех пор, пока управление не будет передано основной программе, в которой был сделан первый вызов рекурсивной функции.

СОВЕТ

Рекурсия — это мощный инструмент для повторного выполнения кода, когда наперед неизвестно точно, сколько раз код должен быть выполнен. Она идеально подходит для поиска в древовидных структурах, например в файловых системах и многоуровневых распределенных сетях.

Что такое рекурсивный запрос

Рекурсивным называется запрос, который функционально зависит от себя самого. Самым простым вариантом такой функциональной зависимости является случай, когда в инструкции запроса `Q1` находится вызов этого же запроса. Возможен и более сложный случай, когда запрос `Q1` зависит от запроса `Q2`, который, в свою очередь, зависит от `Q1`. Здесь по-прежнему имеют место функциональная зависимость и рекурсия, сколько бы запросов ни находилось между первым и вторым вызовом одного и того же запроса.

Где можно применить рекурсивный запрос

Во многих трудных ситуациях рекурсивные запросы помогают сэкономить и время, и нервы. Предположим, например, что у вас есть документ, который дает право бесплатного перелета любым авиарейсом воображаемой компании Vannevar Airlines. Неплохо, правда? И тут же встает вопрос: "Куда можно бесплатно полететь?" Все авиарейсы Vannevar Airlines перечислены в таблице FLIGHT, и для каждого из них указан его номер, пункт отправления и пункт назначения (табл. 13.1).

Таблица 13.1. Авиарейсы компании Vannevar Airlines

FlightNo (номер рейса)	Source (пункт отправления)	Destination (пункт назначения)
3141	Портленд	Округ Ориндж
2173	Портленд	Шарлотт
623	Портленд	Дейтона-Бич
5440	Округ Ориндж	Монтгомери
221	Шарлотт	Мемфис
32	Мемфис	Шампейн
981	Монтгомери	Мемфис

Маршруты этих авиарейсов, нанесенные на карте Соединенных Штатов, показаны на рис. 13.3.

Чтобы начать реализацию своего плана проведения отпуска, создайте с помощью SQL в базе данных таблицу FLIGHT.

```
CREATE TABLE FLIGHT (
   FlightNo      INTEGER      NOT NULL,
   Source        CHAR (30),
   Destination   CHAR (30)
) ;
```

Как только таблица будет создана, ее можно заполнить данными из табл. 13.1.

Предположим, вы хотите лететь из Портленда к своему другу в Монтгомери. Естественно, вы зададите себе вопросы: "В какие города я попаду самолетами Vannevar Airlines, если начать с Портленда?" и "Куда я смогу долететь самолетами этой авиакомпании, если садиться на самолет в Монтгомери?"

В одни города можно долететь без пересадок, а в другие нельзя. По пути в некоторые города придется делать не менее одной такой пересадки. Конечно, можно найти все города, куда самолеты Vannevar Airlines могут вас доставить из любого выбранного вами города. Но если вы будете искать города, выполняя один запрос за другим, то учтите, что вы пошли по самому трудному пути.

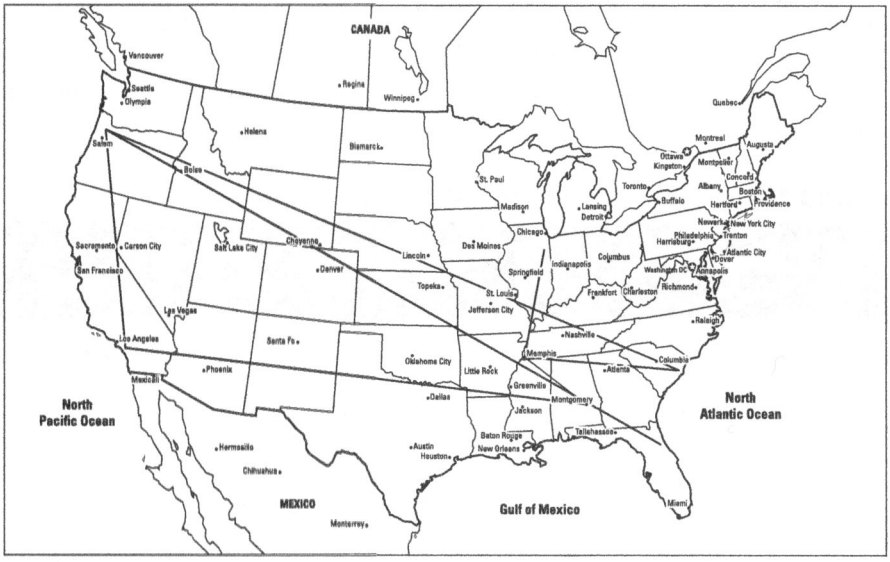


Рис. 13.3. Карта маршрутов компании Vannevar Airlines

Решение "в лоб"

Найти то, что вы хотите узнать (при условии, что у вас есть терпение и время), можно с помощью последовательности запросов, в первом из которых пунктом отправления является Портленд:

```
SELECT Destination FROM FLIGHT WHERE Source = 'Портленд';
```

Этот первый запрос возвращает три значения: `Округ Ориндж`, `Шарлотт` и `Дейтона-Бич`. Первый из них, если хотите, можно сделать пунктом отправления уже во втором запросе:

```
SELECT Destination FROM FLIGHT WHERE Source = 'Округ Ориндж';
```

В результате выполнения второго запроса получаем значение `Монтгомери`. В третьем запросе можем снова использовать результаты первого запроса, взяв на этот раз в качестве начального пункта второй город (`Шарлотт`):

```
SELECT Destination FROM FLIGHT WHERE Source = 'Шарлотт';
```

Этот запрос возвращает значение Мемфис. Результаты первого запроса можно использовать и в четвертом запросе, взяв в качестве начального пункта последний из этих результатов (Дейтона-Бич):

```
SELECT Destination FROM FLIGHT WHERE Source = 'Дейтона-Бич';
```

Прошу прощения, четвертый запрос возвращает неопределенное значение, поскольку у Vannevar Airlines нет авиарейсов из Дейтона-Бич. Но в качестве начального пункта можно также использовать город Монтгомери, который возвращен вторым запросом. Итак, в пятом запросе используем результат выполнения второго:

```
SELECT Destination FROM FLIGHT WHERE Source = 'Монтгомери';
```

Этот запрос возвращает значение Мемфис, но мы уже узнали, что в этот город можно попасть через Шарлотт. Тем не менее Мемфис можно использовать в качестве начального пункта в следующем запросе:

```
SELECT Destination FROM FLIGHT WHERE Source = 'Мемфис';
```

Этот запрос возвращает значение Шампейн. Им также можно пополнить список городов, куда вы сможете попасть (пусть даже с двумя пересадками). А так как вас интересуют авиарейсы с пересадками, то в запросе в качестве начального пункта можно использовать и этот город:

```
SELECT Destination FROM FLIGHT WHERE Source = 'Шампейн';
```

Обидно! Запрос возвращает неопределенное значение. Оказывается, у Vannevar Airlines нет авиарейсов и из Шампейн. (Уже семь запросов — вам еще не надоело?)

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, вы застрянете в Шампейн. В таком случае вы сможете поступить в Университет штата Иллинойс и прослушать там пару лекций по базам данных.

Конечно, когда-нибудь этот метод даст исчерпывающий ответ на вопрос "В какие города можно попасть из Портленда?" Но вводить один запрос за другим, при этом составляя каждый из них (кроме первого) на основе результатов предыдущего, — сложная работа, требующая много времени и, скажем прямо, нудная.

Экономия времени с помощью рекурсивного запроса

Получить нужную информацию будет проще, если создать всего один 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 = 'Портленд';
```

В таблицу REACHABLEFROM заносятся данные о доступности одних городов из других. В начале первого прохода, выполняемого в процессе рекурсии, в таблице FLIGHT будет семь строк, а в таблице REACHABLEFROM — ни одной. Оператор UNION берет семь строк из таблицы FLIGHT и копирует их в таблицу REACHABLEFROM. Тогда в таблице REACHABLEFROM появятся данные, показанные в табл. 13.2.

ВНИМАНИЕ!

Как упоминалось ранее, рекурсия не является частью базового языка SQL, поэтому некоторые СУБД могут ее не поддерживать.

Таблица 13.2. Таблица REACHABLEFROM после одного прохода рекурсии

Source	Destination
Портленд	Округ Ориндж
Портленд	Шарлотт
Портленд	Дейтона-Бич
Округ Ориндж	Монтгомери
Шарлотт	Мемфис
Мемфис	Шампейн
Монтгомери	Мемфис

Интересное начнется уже при втором проходе. Предложение `WHERE in.Destination = out.Source` означает, что просматриваются только те строки, в которых поле `Destination` таблицы `REACHABLEFROM` равно полю `Source` таблицы `FLIGHT`. Для каждой такой строки берутся значения поля `Source` из таблицы `REACHABLEFROM` и поля `Destination` из таблицы `FLIGHT`, которые затем в качестве новой строки добавляются в таблицу `REACHABLEFROM`. Результат этого прохода показан в табл. 13.3.

Таблица 13.3. Таблица `REACHABLEFROM` после двух проходов рекурсии

Source	Destination
Портленд	Округ Ориндж
Портленд	Шарлотт
Портленд	Дейтона-Бич
Округ Ориндж	Монтгомери
Шарлотт	Мемфис
Мемфис	Шампейн
Монтгомери	Мемфис
Портленд	Монтгомери
Портленд	Мемфис
Округ Ориндж	Мемфис
Шарлотт	Шампейн

Эти результаты выглядят более полезными. Теперь в таблице `REACHABLEFROM` поле `Destination` содержит все города, в которые можно попасть из любого города, указанного в поле `Source` той же таблицы, делая при этом не более одной пересадки. На следующем прохода рекурсия обработает маршруты с двумя пересадками, и так будет продолжаться до тех пор, пока не будут найдены все города, куда только можно попасть.

После завершения рекурсии третья, последняя, инструкция `SELECT` (которая не участвует в рекурсии) извлекает из таблицы `REACHABLEFROM` только те города, в которые можно попасть из Портленда. В этом примере видно, что можно попасть во все остальные шесть городов, причем с небольшим числом пересадок.

ЗАПОМНИ!

Если вы внимательно изучите код рекурсивного запроса, то увидите, что он не выглядит проще, чем семь отдельных запросов, которые он заменил. Однако у этого запроса есть два преимущества:

- после его запуска постороннее вмешательство больше не требуется;
- он быстро работает.

Представьте себе реальную авиакомпанию, на карте маршрутов которой находится намного больше городов. Чем больше возможных пунктов назначения, тем ощутимее польза от рекурсивного метода.

Что же делает запрос рекурсивным? То, что мы определяем таблицу `REACHABLEFROM` на основе ее самой. Рекурсивной частью определения является вторая инструкция `SELECT`, которая следует сразу за оператором `UNION`. Таблица `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 таблице `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 начального пункта. Третья, и последняя, инструкция `SELECT` возвращает только те города, в которые можно попасть из Портленда. Так что желаем приятного путешествия!

Где еще можно использовать рекурсивные запросы

Любая задача, которую можно представить в виде древовидной структуры, поддается решению с помощью рекурсивного запроса. Классическим примером того, как такие запросы применяются в промышленности, является обработка материалов (процесс превращения сырья в конечный продукт). Предположим, ваша компания разрабатывает новый гибридный автомобиль. Такую машину собирают из узлов (двигателя, батарей и т.п.), которые, в свою очередь, состоят из меньших деталей (коленчатого вала, электродов и пр.), а те — из еще меньших компонентов.

Отслеживать в реляционной базе данные обо всех этих компонентах очень трудно — если, конечно, в ней не используется рекурсия. Рекурсия позволяет, начав с целой машины, добраться до наименьшей детали. Хотите найти данные о крепежном винте, который держит клемму отрицательного электрода

вспомогательной батареи? Это возможно, причем без особых затрат времени. Справляться с такими задачами в SQL позволяет конструкция `WITH RECURSIVE`.


СОВЕТ

Кроме того, рекурсия вполне естественна при анализе "что, если". Например, что произойдет, если руководство авиакомпании Vannevar Airlines решит отменить полеты из Портленда в Шарлотт? Как это повлияет на полеты в те города, куда сейчас можно добраться из Портленда? Рекурсивный запрос незамедлительно даст ответы на эти вопросы.

4 Управление операциями

В ЭТОЙ ЧАСТИ...

- Управление доступом к данным
- Защита данных
- Применение процедурных языков

Глава 14

Безопасность базы данных

В ЭТОЙ ГЛАВЕ...

- Управление доступом к таблицам базы данных
- Распределение полномочий
- Предоставление прав доступа
- Аннулирование полномочий
- Предотвращение попыток несанкционированного доступа
- Предоставление прав на установку полномочий

Администратор должен обладать специальными знаниями о работе баз данных, поэтому в предыдущих главах рассказывалось о средствах SQL, с помощью которых создаются таблицы и обрабатываются данные. В главе 3 вы узнали о средствах SQL, которые предназначены для защиты базы данных от потенциальных угроз и несанкционированного использования, а сейчас мы рассмотрим эту тему более подробно.

Администратор базы данных может управлять доступом к ней со стороны пользователей, предоставляя или запрещая им доступ к отдельным компонентам системы. Администратор может даже выдавать право предоставления полномочий или лишать такого права. Правильное использование средств безопасности, имеющихся в SQL, обеспечивает надежную защиту важных данных. В то же время неразумное использование этих средств лишь создает проблемы рядовым пользователям, которые просто пытаются выполнять свою работу.

В базах данных часто содержится жизненно важная информация, к которой нельзя допускать первого встречного. Поэтому в SQL существуют разные уровни доступа — от полного доступа до полного его отсутствия, — и между ними находится несколько уровней частичного доступа. Решив, какие именно операции уполномочен выполнять каждый конкретный пользователь, администратор базы данных может предоставить ему все нужные для работы права и при этом ограничить его доступ к частям базы данных, содержащим конфиденциальную информацию.

Язык управления данными

Инструкции SQL, используемые для создания баз данных, образуют подгруппу, которая называется *языком определения данных* (Data Definition Language — DDL). А для добавления, изменения или удаления информации можно задействовать другие инструкции SQL, собирательно называемые *языком обработки данных* (Data Manipulation Language — DML). В SQL существуют также инструкции, которые не попадают ни в одну из этих категорий. Иногда программисты называют их *языком управления данными* (Data Control Language — DCL). Инструкции DCL в основном предназначены для защиты базы данных от несанкционированного доступа, от нежелательных последствий одновременной работы нескольких пользователей, а также от возможных аварий в электрических сетях и неисправностей оборудования. В этой главе мы поговорим о защите от несанкционированного доступа.

Уровни доступа пользователей

SQL обеспечивает контролируемый доступ к девяти функциям управления базами данных.

- **Создание, просмотр, обновление и удаление.** Эти функции соответствуют инструкциям `INSERT`, `SELECT`, `UPDATE` и `DELETE` (см. главу 6).
- **Обеспечение ссылочной целостности.** С помощью ключевого слова `REFERENCES` (см. главы 3 и 5) можно определить ограничения ссылочной целостности для таблицы, зависящей от другой таблицы в базе данных.
- **Использование символьных наборов.** Ключевое слово `USAGE` относится к доменам, наборам символов, схемам сортировки и трансляциям (см. главу 5).

- **Определение новых типов данных.** С помощью ключевого слова `UNDER` можно использовать типы данных, определяемые пользователем.
- **Реакция на событие.** С помощью ключевого слова `TRIGGER` обеспечивается выполнение заданной инструкции SQL или целого блока инструкций при каждом возникновении некоторого предопределенного события.
- **Выполнение.** С помощью ключевого слова `EXECUTE` обеспечивается выполнение заданной процедуры.

Администратор базы данных

В большинстве крупных баз данных с большим количеством пользователей высшей властью обладает администратор базы данных (database administrator — DBA), которому предоставлены права и полномочия на любые действия с базой данных. Одновременно администратор несет и огромную ответственность. Он может легко испортить базу данных и пустить на ветер тысячи часов работы пользователей. Все администраторы должны тщательно обдумывать последствия, которые может возыметь каждое их действие.

Администратор не только обладает полным доступом к базе данных — под его контролем находятся и все права пользователей. Некоторые избранные пользователи должны получать доступ к большему количеству данных и, возможно, к большему количеству таблиц, чем остальные пользователи.

Самый простой способ стать администратором — это самостоятельно установить СУБД. В руководстве по установке указывается учетная запись, или регистрационное имя, 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 действием, которое вы совершите после ввода учетной записи и пароля (иначе говоря, *регистрации*), должна стать замена полученного вами пароля собственным, секретным.


ЗАПОМНИ!

Если пароль не будет изменен, то любой, кто прочитает руководство пользователя СУБД, сможет зарегистрироваться с полным набором полномочий, и вряд ли вам это понравится. Но если изменение внесено, то зарегистрироваться в качестве администратора смогут только те, кто знает ваш новый пароль. Желательно, чтобы он был известен малому кругу особо доверенных людей. Вдруг на вас завтра свалится метеорит? Всякое может случиться! Вашим коллегам

нужно иметь возможность работать и в ваше отсутствие. Каждый, кто знает регистрационное имя администратора базы данных и пароль, становится администратором (естественно, после использования этих реквизитов для доступа к системе).


СОВЕТ

Если у вас есть полномочия администратора, то регистрироваться в системе в качестве такового следует только тогда, когда нужно выполнить какое-то специальное задание, для которого требуются эти полномочия. Как только вы выполните это задание, тут же выходите из системы. Для выполнения повседневной 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.

Владельцы объектов базы данных

Кроме администраторов, существует еще один класс привилегированных пользователей: *владельцы объектов базы данных*. Такими объектами, например, являются таблицы и представления. Любой пользователь, создающий какой-либо объект базы данных, может назначить его владельца. Владелец таблицы обладает всеми возможными полномочиями, связанными с этой таблицей, включая управление доступом к ней. Представление создается на основе таблиц, причем владелец представления не обязательно должен быть владельцем этих таблиц. Однако в этом случае владелец представления получает полномочия только на само представление, а не на таблицы, на основе которых оно было создано. Отсюда следует вывод, что нельзя обойти защиту таблицы, принадлежащей другому пользователю, создав на ее основе какое-либо представление.

Открытый доступ

В сетевой терминологии термином "открытый доступ" обозначают всех пользователей, которые не имеют специальных полномочий администратора или владельцев объектов и которым эти привилегированные пользователи специально не предоставили особых прав доступа. Если привилегированный пользователь определяет некоторые права доступа как `PUBLIC`, то их получают все пользователи системы.

В большинстве 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 показан на рис. 14.1.

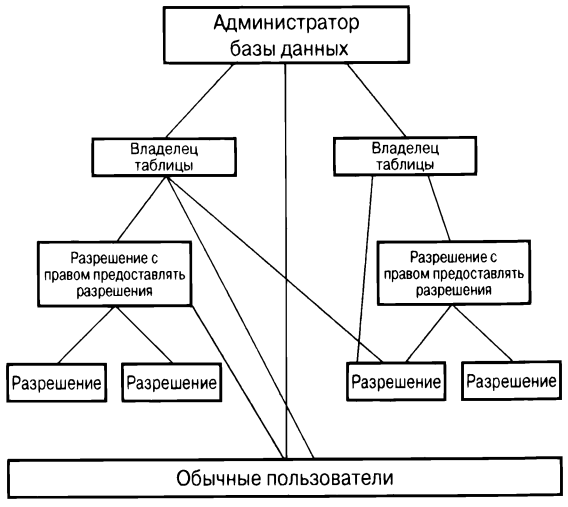


Рис. 14.1. Иерархическая структура прав доступа

Предоставление полномочий пользователям

В силу своего положения администратор базы данных имеет все полномочия на все ее объекты. В конце концов, владелец объекта имеет на него все права, а база данных сама является объектом. Ни у кого из пользователей не будет полномочий на объект, если только их ему специально не предоставит тот, у кого эти полномочия (а также право их передавать) уже есть. Предоставить полномочия можно с помощью инструкции GRANT, которая имеет следующий синтаксис.

```
GRANT список_полномочий
   ON объект
   TO список_пользователей
   [WITH HIERARCHY OPTION]
   [WITH GRANT OPTION]
   [GRANTED BY предоставитель] ;
```

В этой инструкции элемент *список_полномочий* определяется следующим образом.

```
полномочия[, полномочия] ...
```

или

```
ALL PRIVILEGES
```

В свою очередь, *полномочия* определяются следующим образом.

```
SELECT
| DELETE
| INSERT [(имя_столбца[, имя_столбца]...)]
| UPDATE [(имя_столбца[, имя_столбца]...)]
| REFERENCES [(имя_столбца[, имя_столбца]...)]
| USAGE
| UNDER
| TRIGGER
| EXECUTE
```

А *объект* в инструкции GRANT определяется так.

```
[TABLE] <имя таблицы>
| DOMAIN <имя домена>
| COLLATION <имя схемы сортировки>
| CHARACTER SET <имя символьного набора>
| TRANSLATION <имя трансляции>
| TYPE <имя допустимого типа данных, определенного пользователем>
| SEQUENCE <имя генератора последовательностей>
| <специальная метка процедуры>
```

И наконец, *список_пользователей* в инструкции GRANT определяется следующим образом.

```
регистрационное_имя[, регистрационное_имя]...
| PUBLIC
```

Параметр *предоставитель* может иметь значение CURRENT_USER или CURRENT_ROLE.

СОВЕТ

Приведенный синтаксис трактует представление как таблицу. Полномочия SELECT, DELETE, INSERT, UPDATE и REFERENCES относятся только к таблицам и представлениям. Полномочие USAGE имеет отношение к доменам, наборам символов, схемам сортировки и трансляциям. Полномочие UNDER относится только к типам, а EXECUTE — к процедурам. В последующих разделах приведены различные примеры использования инструкции GRANT.

Роли

Одним из типов идентификатора подтверждения полномочий является *имя пользователя*. Оно удостоверяет пользователя или программу, имеющую полномочия на выполнение с базой данных одной или нескольких операций. Если в крупной организации с большим количеством пользователей предоставлять полномочия отдельно каждому сотруднику, то такая операция может занять очень много времени. В стандарте SQL эта проблема решается за счет введения понятия *роли*.

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` (ответственный за безопасность). Естественно, эти полномочия должны отличаться от тех, которые предоставляются пользователям, имеющим роль `SalesClerk` (продавец).

Как правило, не все средства, определенные в последней версии стандарта SQL, доступны во всех СУБД. Прежде чем пытаться использовать роли, ознакомьтесь с документацией к конкретной СУБД.

Для создания роли можно использовать следующий синтаксис:

```
CREATE ROLE SalesClerk ;
```

После того как роль будет создана, можно назначить ее тому или иному пользователю с помощью инструкции `GRANT`:

```
GRANT SalesClerk to Becky ;
```

Полномочия ролям предоставляются точно так же, как и пользователям, правда, за одним 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` новые записи.

Просмотр данных

А вот пример предоставления полномочий на просмотр таблицы товаров всем пользователям.

```
GRANT SELECT
   ON PRODUCT
   TO PUBLIC ;
```

Эти полномочия позволяют любому пользователю системы просматривать содержимое таблицы `PRODUCT`.

ВНИМАНИЕ!

Эта инструкция может быть очень опасной. В столбцах таблицы PRODUCT — таких, например, как CostOfGoods (закупочная стоимость товаров) — может храниться информация, не предназначенная для всеобщего обозрения. Чтобы предоставить доступ к большей части информации, скрывая при этом конфиденциальные данные, создайте на основе таблицы представление, в котором не будет столбцов с информацией, а затем предоставьте полномочия SELECT не на саму таблицу, а на ее представление. Пример такой процедуры приведен ниже.

```
CREATE VIEW MERCHANDISE AS
     SELECT Model, ProdName, ProdDesc, ListPrice
       FROM PRODUCT ;
GRANT SELECT
   ON MERCHANDISE
   TO PUBLIC ;
```

Пользуясь представлением MERCHANDISE, рядовой пользователь не сможет увидеть CostOfGoods или любой другой столбец таблицы PRODUCT, за исключением тех четырех, которые перечислены в инструкции CREATE VIEW. Это столбцы Model (модель), ProdName (название товара), ProdDesc (описание товара) и ListPrice (цена в прайсе).

Модификация табличных данных

В любой организации табличные данные со временем меняются. Поэтому некоторым сотрудникам необходимо предоставить возможность обновлять данные, а всем остальным, наоборот, запретить это делать. Ниже приведен пример предоставления полномочий на обновление данных.

```
GRANT UPDATE (BonusPct)
   ON BONUSRATE
   TO SalesMgr ;
```

Исходя из рыночной конъюнктуры, менеджер по продажам может регулировать значения процентной ставки, на основе которых рассчитываются премии продавцов (столбец BonusPct). Однако менеджер не может изменять значения в столбцах MinAmount и MaxAmount, которые определяют диапазон, соответствующий 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 устаревших строк из таблицы

Клиенты прекращают свою деятельность или перестают покупать ваши изделия по какой-то другой причине. Сотрудники увольняются, уходят на пенсию или отправляются в мир иной. Товары устаревают. Жизнь продолжается, и данные со временем теряют актуальность. Устаревшие записи необходимо удалять. При этом следует тщательно контролировать, кто и какие записи может удалять. С этой задачей справляется инструкция GRANT.

```
GRANT DELETE
   ON EMPLOYEE
   TO PersonnelMgr ;
```

Менеджер по персоналу может удалять записи из таблицы EMPLOYEE (сотрудник). Этим также может заниматься администратор или владелец данной таблицы. Кроме них, записи о сотрудниках больше никто удалять не сможет (если только кто-то не получит такую возможность благодаря другой инструкции GRANT).

Использование ссылок на связанные таблицы

Если в одной таблице в качестве внешнего ключа используется первичный ключ другой таблицы, то пользователи первой таблицы имеют доступ к данным второй. Эта ситуация создает потенциально опасную лазейку, через которую неуполномоченные пользователи могут извлечь секретную информацию. При этом пользователю для получения информации о содержимом таблицы не нужны никакие специальные права доступа к ней. Если у этого пользователя есть право доступа к первой таблице, которая ссылается на вторую, то этих прав часто бывает достаточно, чтобы получить доступ и ко второй таблице.

Предположим, например, что в таблице LAYOFF_LIST содержатся имена и фамилии сотрудников, подлежащих увольнению в следующем месяце. 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), и теперь он может создать новую таблицу SNOOP, в которой EmpID является внешним ключом. Этот внешний ключ и даст возможность потихоньку подглядывать в таблицу LAYOFF_LIST. (О том, как создать внешний ключ с помощью предложения REFERENCES, см в главе 5.) Все эти приемы должны быть известны каждому системному администратору. Вот как выглядит код создания таблицы SNOOP.

```
CREATE TABLE SNOOP
   (EmpID INTEGER REFERENCES LAYOFF_LIST) ;
```

Теперь все, что нужно сделать, — это попытаться с помощью инструкции INSERT вставить в таблицу SNOOP строки, соответствующие идентификатору каждого сотрудника. Вставки, принимаемые этой таблицей, как раз и относятся к сотрудникам, внесенным в список увольняемых, в то время как все отвергаемые вставки — к сотрудникам, отсутствующим в этом списке.


СОВЕТ

Не все потеряно! Вы не подвержены риску обнародования конфиденциальных данных. Последние версии стандарта SQL закрывают эту потенциальную брешь, требуя, 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 ;
```

Безусловно, вам нужно удостовериться, что эти новые возможности реализованы в вашей СУБД.

Использование доменов

На безопасность также влияют домены, наборы символов, схемы сортировки и трансляции. В частности, создавая домены, внимательно следите за тем, чтобы из-за них не пострадала система безопасности.

Можно определить домен, который охватывает некоторый набор столбцов. Тем самым вы предписываете, что все эти столбцы будут иметь один и тот же тип, а также одни и те же ограничения. Теперь столбцы, создаваемые инструкцией CREATE DOMAIN, смогут унаследовать тип и ограничения заданного домена. Естественно, при необходимости для отдельных столбцов эти характеристики можно переопределить, и все же домены — это удобное средство, которое позволяет с помощью одного объявления задавать несколько характеристик сразу для целого набора столбцов.

Домены пригодятся в случае существования множества таблиц, содержащих столбцы со сходными характеристиками. Например, база данных вашей фирмы может состоять из нескольких таблиц. Представим, что в каждой из них содержится столбец Price (цена) с типом данных DECIMAL(10, 2), значения в котором должны быть неотрицательными и не превышать 10 000. В этом случае, прежде чем создавать таблицы с такими столбцами, имеет смысл создать домен, определяющий характеристики этих столбцов. Создание домена PriceTypeDomain (домен типа цены) продемонстрировано 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) с типом данных `CHAR(5)`, где первый символ должен быть `X`, `C` или `H`, а последний — или `9`, или `0`. Для таких столбцов можно создать домен `ProductCodeDomain` (домен кода товара), что и показано 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.

```
CREATE TABLE PRODUCT (
   ProductCode    ProductCodeDomain,
   ProductName    CHAR (30),
   Price          PriceTypeDomain
) ;
```


ВНИМАНИЕ!

Как упоминалось ранее в контексте других стандартных средств SQL, не все версии СУБД поддерживают все возможности языка. Инструкция `CREATE DOMAIN` относится именно к таким, неуниверсальным, средствам. СУБД iAnywhere от Sybase поддерживает ее, как и PostgreSQL, но Oracle и SQL Server — нет.

Вместо определения типа данных для поля `ProductCode` или `Price` в определении таблицы `PRODUCT` указывается соответствующий домен. Таким образом, эти столбцы получают нужные типы данных, и, кроме того, для них устанавливаются ограничения, определенные в инструкциях `CREATE DOMAIN`.

При использовании доменов возникают вопросы, связанные с безопасностью. Если кто-то вдруг захочет использовать созданные вами домены, может ли это 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` инструкции `CREATE DOMAIN`. Если значение этой верхней границы является закрытой информацией, то необходимо запретить использовать домен `PriceType` неуполномоченным пользователям. Для защиты таблиц в таких 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 следующим образом:

```
GRANT USAGE ON DOMAIN PriceType TO SalesMgr ;
```

Применение инструкции DROP к доменам может вызвать проблемы, связанные с безопасностью. Если вы попытаетесь с помощью инструкции DROP удалить домен, а в каких-то таблицах есть столбцы, определенные с его помощью, то такие таблицы станут источником неприятностей. Прежде чем применить инструкцию DROP к самому домену, ее вначале следует применить к каждой из использующих его таблиц. Возможно, вы обнаружите, что к доменам эта инструкция вообще не применима. Действие инструкции DROP зависит от конкретной реализации. СУБД iAnywhere может сильно отличаться в этом смысле от PostgreSQL. Однако в любом случае придется ограничить круг лиц, которым разрешено применять инструкцию DROP к доменам. То же относится к наборам символов, схемам сортировки и трансляциям.

Запуск инструкций SQL

В некоторых случаях выполнение одной инструкции SQL может запустить другую или даже целый их блок. В SQL поддерживаются триггеры. *Триггер* — это механизм, который задает событие для запуска, время активизации и одно или несколько запускаемых действий.

- **Событие** инициирует запуск (выражаясь простым языком, дает команду "пуск").
- **Время активизации** триггера определяет, в какой момент должно произойти действие: непосредственно перед событием или после него.
- **Запускаемое действие** — это одна или несколько инструкций SQL.

  При запуске нескольких инструкций SQL все они должны содержаться внутри структуры BEGIN ATOMIC. . . END. Запускаемое действие может быть выражено инструкцией INSERT, UPDATE или DELETE.

Например, можно использовать триггер для выполнения инструкции, контролирующей допустимость нового значения, перед обновлением данных с помощью инструкции UPDATE. Если новое значение окажется некорректным, то обновление будет отменено.

Для создания триггера пользователь или роль должны иметь полномочие TRIGGER. Ниже приведен пример создания 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 таблице журнала CUSTLOG будет сделана запись об удалении.

Предоставление уровневых полномочий

В главе 2 были описаны структурированные типы данных как одна из разновидностей пользовательских типов данных. Архитектура структурированных типов данных по большей части определяется концепциями объектно-ориентированного программирования. В частности, принцип *иерархии* состоит в том, что у типа могут быть *подтипы*, которые наследуют некоторые из атрибутов того типа, от которого они происходят (от *супертипа*). Кроме унаследованных атрибутов, у них могут быть также исключительно их собственные атрибуты. В такой иерархии может быть несколько уровней, а тип, лежащий на нижнем уровне, называется *листовым* (конечным) типом.

Типизированная таблица — это таблица, каждая строка которой является экземпляром соответствующего структурированного типа данных. У типизированной таблицы есть по одному столбцу для каждого атрибута структурированного типа данных. Название и тип данных столбца совпадают с названием и типом данных атрибута.

Предположим, например, что вы хозяин арт-галереи и продаете произведения искусства. Кроме оригиналов, вы продаете репродукции, подписанные копии с ограниченным тиражом, а также обычные копии и постеры. Для своих экспонатов вы можете создать такой структурированный тип данных.

```
CREATE TYPE artwork (
   artist         CHARACTER VARYING (30),
   title          CHARACTER VARYING (50),
   description    CHARACTER VARYING (256),
   medium         CHARACTER VARYING (20),
   creationDate   DATE )
   NOT FINAL
```

ВНИМАНИЕ!

Снова отмечу, что инструкция CREATE TYPE доступна не во всех СУБД. Она поддерживается в PostgreSQL, Oracle и SQL Server.

В своей системе управления галереей вы хотите установить различия между оригиналами и репродукциями. Более того, вы считаете нужным установить различия между разными видами репродукций. На рис. 14.2 показан возможный вариант использования иерархии для реализации необходимых разграничений. Тип `artwork` (экспонат галереи) может иметь подтипы, которые, в свою очередь, могут иметь собственные подтипы.

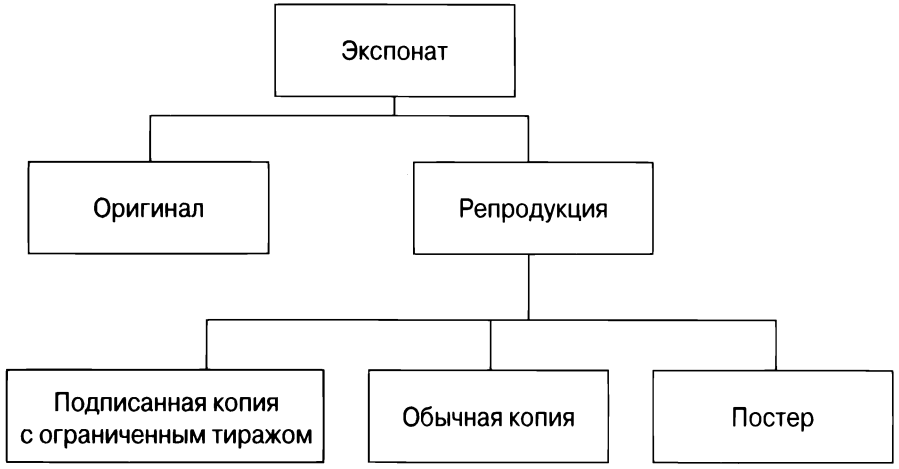


Рис. 14.2. Иерархия типов для экспонатов галереи

Между типами в иерархии типов и таблицами в иерархии типизированных таблиц есть соответствие "один к одному". Стандартные таблицы, рассмотренные в главах 4 и 5, нельзя поместить в иерархию типизированных таблиц.

Вместо первичного ключа у типизированной таблицы есть *автореферентный* столбец, гарантирующий уникальность не только максимальной супертаблицы иерархии, но и всех ее подтаблиц. Автореферентный столбец определяется предложением `REF IS` в инструкции создания максимальной супертаблицы. Когда ссылку создает сама система, уникальность значений гарантируется.

Право на предоставление полномочий

Администратор базы данных может предоставить любому пользователю любые полномочия. Владелец объекта также может предоставить любому пользователю любые полномочия, связанные с этим объектом. Но те, кто получил свои полномочия таким способом, не могут предоставить их третьим лицам. Это ограничение позволяет администратору или владельцу объекта в достаточной степени сохранять контроль над ситуацией. Только пользователи,

уполномоченные администратором или владельцем объекта предоставлять права доступа третьим лицам, могут выполнять такие операции.

С точки зрения безопасности представляется разумным ограничить возможность раздачи прав доступа. Тем не менее пользователям зачастую нужны права на предоставление полномочий. Конвейер не должен останавливаться только из-за того, что кто-то заболел, находится в отпуске или ушел на обед.


ЗАПОМНИ!

Можно предоставить *некоторым* пользователям право передавать их права доступа надежным коллегам. Для передачи пользователю такого права в инструкции `GRANT` используется предложение `WITH GRANT OPTION`. Вот пример.

```
GRANT UPDATE (BonusPct)
   ON BONUSRATE
   TO SalesMgr
 WITH GRANT OPTION ;
```

Теперь менеджер по продажам может предоставлять права на обновление данных с помощью следующей инструкции.

```
GRANT UPDATE (BonusPct)
   ON BONUSRATE
   TO AsstSalesMgr ;
```

После выполнения этой инструкции заместитель менеджера по продажам сможет вносить изменения в данные столбца `BonusPct` таблицы `BONUSRATE`.


ВНИМАНИЕ!

Безусловно, приходится искать компромисс между безопасностью и удобством. Владелец таблицы `BONUSRATE`, предоставляя менеджеру по продажам полномочия `UPDATE` с помощью атрибута `WITH GRANT OPTION`, делится с ним значительной частью своей власти. Ему остается надеяться, что менеджер по продажам серьезно отнесется к возложенной на него ответственности и будет осторожен с передачей полномочий другим лицам.

Отзыв полномочий

Наряду с предоставлением прав доступа должна быть и возможность их отзыва. 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 таких сотрудников придется немедленно отозвать.

В SQL удаление полномочий на доступ к данным выполняется с помощью инструкции REVOKE. Ее синтаксис аналогичен синтаксису инструкции GRANT, но только с противоположным результатом.

```
REVOKE [GRANT OPTION FOR] список_полномочий
    ON объект
  FROM список_пользователей [RESTRICT | CASCADE] ;
```

С помощью этой инструкции можно отзывать полномочия, составляющие *список_полномочий*, не затрагивая при этом все остальные. Главное различие между инструкциями REVOKE и GRANT состоит в том, что в первой из них можно использовать одно из двух необязательных ключевых слов: RESTRICT (ограничить) или CASCADE (каскадное удаление).

Пусть для предоставления полномочий вы использовали инструкцию GRANT вместе с предложением WITH GRANT OPTION. Тогда использование ключевого слова CASCADE в инструкции REVOKE приводит к отзыву указанных полномочий как у того пользователя, которому вы их предоставили, так и у всех пользователей, кому эти полномочия он уже успел предоставить сам.

С другой стороны, инструкция 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 инструкции REVOKE с ключевым словом RESTRICT, то вместо выполнения этой инструкции будет выдано сообщение об ошибке. Это четкое указание на необходимость найти тех людей, которым данный человек предоставил полномочия. Затем можно подтвердить или отменить полномочия этих лиц.

Инструкцию REVOKE с необязательным предложением GRANT OPTION FOR можно использовать для отзыва у пользователя возможности предоставлять указанные полномочия другим лицам, оставляя при этом полномочия для самого пользователя. Если инструкция REVOKE содержит и предложение GRANT OPTION FOR, и ключевое слово CASCADE, то отзываются все полномочия, предоставленные пользователем, а также полномочия этого пользователя на предоставление полномочий другим. А если в инструкции есть и предложение GRANT OPTION FOR, и ключевое слово RESTRICT, то события развиваются по одному из двух сценариев.

- Если пользователь еще не предоставил никому другому те полномочия, которые вы у него отзываете, то инструкция REVOKE успешно выполняется, аннулируя разрешение этому пользователю предоставлять полномочия кому-либо другому.
- Если пользователь уже успел предоставить кому-нибудь хотя бы одно из отзываемых у него полномочий, то его полномочия не отзываются, а возвращается код ошибки.

ВНИМАНИЕ!

Возможность предоставления полномочий с помощью предложения WITH GRANT OPTION в сочетании с выборочным отзывом полномочий делает систему безопасности намного сложнее, чем кажется на первый взгляд. Например, любой пользователь может получить одни и те же полномочия от множества "дарителей". И если один из них затем отзовет предоставленные им полномочия, то у пользователя они все равно останутся, поскольку продолжают действовать полномочия, предоставленные другим "дарителем". Если благодаря предложению WITH GRANT OPTION полномочия передаются от одного пользователя к другому, то такая ситуация порождает *цепочку зависимостей*. В этой цепочке полномочия одного пользователя зависят от таких же полномочий другого. Если вы администратор или владелец объекта, никогда не забывайте о том, что полномочия, предоставленные с помощью предложения WITH GRANT OPTION, могут "всплыть" в самых неожиданных местах. Отзыв полномочий у нежелательных пользователей при одновременном сохранении их у законных "арендаторов" может создавать достаточно трудные ситуации. В действительности предложения GRANT OPTION и CASCADE могут таить в себе многочисленные ловушки. Поэтому при использовании этих предложений обращайтесь к документации на вашу СУБД.

Совместное использование инструкций GRANT и REVOKE

Предоставление множеству пользователей многочисленных полномочий на выбранные столбцы таблицы сопряжено с вводом большого объема кода. Рассмотрим следующий пример. Вице-президент по продажам хочет, чтобы все, кто занимается продажами, могли просматривать содержимое таблицы CUSTOMER (клиент). Но обновлять, удалять или вставлять строки позволено только менеджерам по продажам. И *никто* не имеет права обновлять поле идентификатора клиента CustID. Соответствующие полномочия можно предоставить менеджерам Tyson, Keith и David с помощью следующих инструкций GRANT.

```
GRANT SELECT, INSERT, DELETE
   ON CUSTOMER
   TO Tyson, Keith, David ;
GRANT UPDATE
   ON CUSTOMER (Company, CustAddress, CustCity,
      CustState, CustZip, CustPhone, ModLevel)
```

```
   TO Tyson, Keith, David ;
GRANT SELECT
   ON CUSTOMER
   TO Jen, Val, Mel, Neil, Rob, Sam, Walker, Ford,
      Brandon, Cliff, Joss, MichelleT, Allison, Andrew,
      Scott, MichelleB, Jaime, Lynleigh, Matthew, Amanda;
```

Теперь попытаемся упростить этот код. Все пользователи обладают полномочиями на просмотр (SELECT) таблицы CUSTOMER. Менеджеры по продажам имеют на эту таблицу полномочия вставки (INSERT) и удаления (DELETE), а также могут обновлять любой ее столбец, кроме CustID. Поэтому того же результата можно достичь более простым способом.

```
GRANT SELECT
   ON CUSTOMER
   TO SalesReps ;
GRANT INSERT, DELETE, UPDATE
   ON CUSTOMER
   TO Managers ;
REVOKE UPDATE
    ON CUSTOMER (CustID)
  FROM Managers ;
```

Это тот же уровень защиты, что и в предыдущем примере, и здесь также нужно использовать три инструкции. Никто не сможет изменить данные в столбце CustID. Полномочия INSERT, DELETE и UPDATE имеют только менеджеры. Как видите, три последние инструкции значительно короче, поскольку здесь не нужно указывать имена всех сотрудников отдела продаж, имена всех менеджеров и перечислять все столбцы таблицы.

Глава 15

Защита данных

В ЭТОЙ ГЛАВЕ...

- Как избежать повреждения базы данных
- Проблемы, вызванные одновременно выполняемыми операциями
- Решение проблем конкуренции с помощью SQL
- Задание уровня защиты с помощью инструкции `SET TRANSACTION`
- Защита данных, не препятствующая нормальной работе с ними
- Предотвращение вредоносных атак

Каждый слышал о законе Мерфи: "Если какая-нибудь неприятность *может* случиться, то она *случается*". Когда наши дела долгое время идут хорошо, мы потешаемся над этим псевдозаконом. Порой нам даже кажется, что мы из тех немногих счастливчиков, над кем не властен закон Мерфи. Даже если неприятности все-таки происходят, мы обычно легко с ними справляемся.

Однако чем сложнее система, тем выше в ней риск возникновения проблем (как сказал бы математик, "вероятность возникновения проблем пропорциональна квадрату сложности системы"). По этой причине масштабные программные проекты почти никогда не сдаются в срок и часто содержат множество ошибок. Реальное многопользовательское приложение, работающее с базой данных, — это большая и сложная система, в которой существует высокая вероятность сбоев. И хотя были разработаны специальные методы проектирования приложений, позволяющие свести последствия сбоев к минимуму,

полностью исключить их невозможно. Это хорошая новость для специалистов, занимающихся сопровождением баз данных. Автоматика, скорее всего, никогда их не заменит. В этой главе мы обсудим наиболее распространенные проблемы, возникающие при работе с базами данных, и средства SQL, которые позволяют их решить.

Угрозы целостности данных

Киберпространство (в том числе ваша собственная сеть) может быть прекрасным местом для посещения, но для находящейся в нем информации это отнюдь не райский уголок. Данные могут быть повреждены или испорчены самыми разными способами. В главе 5 рассказывалось о проблемах, возникающих в результате ввода неправильных данных, ошибок оператора и злонамеренного повреждения данных. Необдуманно составленные инструкции SQL и некорректно спроектированные приложения также могут навредить данным, и не нужно обладать большой фантазией, чтобы представить, как это может произойти. Впрочем, данным угрожают и такие очевидные опасности, как нестабильность платформы и аппаратные сбои. Ниже мы обсудим эти две угрозы, а также неприятности, связанные с одновременным доступом к данным.

Нестабильность платформы

Нестабильность платформы относится к той категории угроз, которых даже и быть не должно, но, увы, они существуют. Чаще всего это случается при запуске одного или нескольких новых и не полностью проверенных компонентов системы. Проблемы могут "притаиться" в новом выпуске СУБД, в новой версии операционной системы и даже в новом оборудовании. Проявляются они обычно тогда, когда вы запускаете очень важное задание. В результате система зависает, а данные портятся. И вам больше ничего не остается, кроме как ругать последними словами свой компьютер и тех, кто его собирал, а также надеяться на работоспособность последней резервной копии.

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. Экспериментируйте с новым программным обеспечением на машине, полностью изолированной от рабочей сети.

Аппаратный сбой

Даже проверенное, высоконадежное оборудование иногда отказывает, отправляя данные на тот свет. Все материальное со временем изнашивается, даже современные компьютеры. Если такой сбой происходит тогда, когда база данных открыта, то данные можно потерять, даже не осознав этого. Рано или поздно подобное случается со всеми. Уж если закон Мерфи и проявляет себя, то в самое неподходящее время.


СОВЕТ

Одним из способов защиты данных от сбоев оборудования является *избыточность*. Создавайте дополнительные копии всего подряд. Для обеспечения максимальной безопасности можно продублировать аппаратные средства вместе с их настройками (если ваша организация может себе это позволить) с таким расчетом, чтобы при необходимости можно было быстро установить и запустить резервные копии базы данных и приложений на резервном оборудовании. Если же ограниченность в средствах не позволяет обеспечить полное дублирование, как можно чаще создавайте резервные копии своей базы данных и приложений — причем настолько часто, чтобы после неожиданного сбоя вам не пришлось заново вводить слишком большой объем данных. Многие СУБД предлагают возможности репликации. И это прекрасно, но они вас не спасут, если вы не сконфигурируете свою систему так, чтобы они функционировали должным образом.

Еще один способ избежать наихудших последствий аппаратных сбоев — применять транзакции, и это основная тема данной главы. *Транзакция* — это неделимая единица работы. Транзакция или выполняется целиком, или не выполняется вовсе. Если подход "все или ничего" кажется вам слишком радикальным, то учтите, что самые большие проблемы возникают в результате частичного выполнения операций с базой данных.

Конкурентный доступ

Предположим, что программы и оборудование, с которыми вы работаете, проверены, данные введены правильно, приложения свободны от ошибок, а оборудование абсолютно надежно. Выходит, что данным ничего не грозит? К сожалению, это не так. Если несколько пользователей одновременно пытаются открыть одну и ту же таблицу базы данных, возникает ситуация *конкурентного доступа* (т.е. их компьютеры соревнуются за право первоочередного доступа к данным). Поэтому многопользовательские СУБД должны обладать эффективными средствами разрешения конфликтов, связанных с одновременным доступом к данным.

Проблемы взаимодействия транзакций

Проблемы, связанные с одновременным доступом, возникают даже в относительно простых приложениях. Рассмотрим в качестве примера 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). При этом выполняются следующие условия.

- В таблице ORDER_MASTER поле OrderNumber (номер заказа) является первичным ключом, а поле CustomerNumber (номер клиента) — внешним ключом, который ссылается на таблицу CUSTOMER.
- В таблице LINE_ITEM поле LineNumber (номер строки) является первичным ключом, а поле ItemNumber (идентификатор товара) — внешним ключом, ссылающимся на таблицу INVENTORY, одним из полей которой является Quantity (количество).
- В таблице INVENTORY первичным ключом является поле ItemNumber. Кроме того, одним из полей этой таблицы является 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 выполнялся полностью, или не выполнялся вовсе. Частичные выполнения заказов не допускаются. (Не переживайте: это же воображаемая ситуация.) Вы пишете приложение, 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 возможна, то приложение оформляет заказ, соответственно уменьшая в таблице INVENTORY значение столбца QuantityOnHand и удаляя из таблиц ORDER_MASTER и LINE_ITEM записи, относящиеся к этому заказу. Пока все хорошо. Вы настраиваете приложение так, чтобы при одновременном доступе нескольких пользователей оно обрабатывало заказы одним из двух способов.

- Первый способ состоит в обработке в таблице INVENTORY записей, соответствующих каждой записи таблицы LINE_ITEM. Если значение поля QuantityOnHand является достаточно большим, то приложение его уменьшит. Но если это значение меньше требуемого, происходит откат транзакции. При этом восстанавливаются все изменения, уже внесенные в таблицу INVENTORY в процессе обработки предыдущих строк таблицы LINE_ITEM данного заказа.
- Второй способ сводится к проверке всех записей таблицы INVENTORY, соответствующих какой-либо записи заказа, находящей-

ся в таблице LINE_ITEM. Если значения во *всех* этих записях таблицы INVENTORY достаточно большие, то выполняется их уменьшение.

Если заказ выполним, то большей эффективностью обладает первый способ, если же нет — второй. Таким образом, если бо́льшая часть заказов выполнима, необходимо использовать первый способ, в противном случае желателен второй. Предположим, что приложение установлено и запущено в многопользовательской системе, в которой в недостаточной мере обеспечивается управление одновременным доступом. Сразу же возникают проблемы. Рассмотрим следующий сценарий.

1. **Покупатель обращается к менеджеру заказов вашей компании (пользователь 1), чтобы купить десять единиц товара 1 (болторезный станок) и пять единиц товара 2 (разводной ключ).**
2. **Пользователь 1 запускает обработку заказа, используя первый способ. В первой строке заказа содержится десять единиц товара 1.**

   На складе находятся десять единиц товара 1, и все это количество требуется для выполнения заказа.

   Функция обработки заказа уменьшает количество товара 1 на складе до нуля. После этого начинается самое интересное — еще один покупатель звонит другому менеджеру вашей компании (пользователь 2).
3. **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.**

   Так как заказ оформить нельзя, выполняется откат.
4. **Тем временем пользователь 1 продолжает обрабатывать заявку и пытается заказать пять единиц товара 2.**

   К сожалению, на складе вашей компании найдено всего четыре единицы товара 2. Поэтому для заказа пользователя 1 также выполняется откат — этот заказ нельзя оформить полностью.

   Таблица INVENTORY возвращается в состояние, в котором она была перед тем, как пользователи обратились в компанию. Получилось так, что не оформлен ни один из заказов, хотя заказ пользователя 2 вполне можно было выполнить.

Второй способ не лучше, пусть и по другим причинам. Пользователь 1 может проверить все заказываемые товары и решить, что все они имеются в наличии. Но если в дело вмешается пользователь 2 и запустит обработку заказа на один из этих товаров до того, как пользователь 1 выполнит операцию уменьшения, то транзакция пользователя 1 может закончиться неудачей.

Последовательное выполнение исключает нежелательные взаимодействия

Конфликта транзакций не происходит, если они выполняются последовательно. Главное — быстрее занять очередь. Если обработка неудачной транзакции пользователя 1 заканчивается до начала транзакции пользователя 2, то инструкция отката `ROLLBACK` вернет все товары, заказанные пользователем 1, на склад, сделав их доступными для 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 по окончании транзакции пользователя 1 пользователь 2 сможет увидеть, какое количество нужного товара имеется в наличии.

Если транзакции выполняются последовательно, одна за другой, то они не смогут взаимодействовать, что исключает нежелательные последствия таких взаимодействий. Если результат одновременных транзакций такой же, как и при последовательном выполнении, то эти транзакции называются *упорядочиваемыми* (сериализуемыми).

Последовательное выполнение транзакций не является панацеей. Всегда приходится искать компромисс между производительностью и уровнем защиты от опасных взаимодействий. Чем сильнее 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 такой компромисс, при котором настройки вашей системы создадут достаточную защиту, но не больше необходимой. Слишком жесткий контроль за одновременным доступом может свести производительность системы на нет.

Уменьшение уязвимости данных

Чтобы уменьшить риски потери данных из-за непредвиденных обстоятельств или непредусмотренных действий, можно принять меры предосторожности на нескольких уровнях. Можно настроить СУБД так, чтобы она без вас реализовывала некоторые из этих мер. Иногда вы даже не будете о них знать. Кроме того, администратор базы данных может по своему усмотрению обеспечить и другие меры защиты. О них вы тоже можете быть либо осведомлены, либо нет. И наконец, как разработчик вы можете сами принять определенные меры предосторожности при написании кода.

СОВЕТ

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.

- Используйте транзакции SQL.
- Обеспечьте такой уровень изоляции транзакций, чтобы соблюдалось равновесие между производительностью и защитой.
- Знайте, когда и как запускать транзакции, блокировать объекты базы данных и выполнять резервное копирование.

Теперь поговорим об этих принципах подробно.

Использование SQL-транзакций

Одним из главных инструментов поддержания целостности баз данных является *транзакция*. Она инкапсулирует все SQL-инструкции, которые могут воздействовать на базу данных. Транзакция SQL завершается одной из двух инструкций: `COMMIT` (фиксация) или `ROLLBACK` (откат).

- Если транзакция завершается инструкцией `COMMIT`, то действие всех ее инструкций выполняется в виде одной "пулеметной очереди".
- Если транзакция завершается инструкцией `ROLLBACK`, то выполняется откат, т.е. отмена всех ее инструкций, а база данных возвращается в то состояние, в котором она находилась перед началом транзакции.

ЗАПОМНИ!

Под термином *приложение* мы будем подразумевать программу, написанную на одном из стандартных языков программирования (таких, как Java или C++), или последовательность команд, вводимых с терминала в рамках одного сеанса.

Приложение может включать последовательность SQL-транзакций. Первая из них начинается при запуске приложения, последняя заканчивается в момент его завершения. Каждая выполняемая приложением инструкция `COMMIT` или `ROLLBACK` завершает одну SQL-транзакцию и начинает следующую. Например, приложение с тремя транзакциями имеет следующий вид.

```
Начало приложения
   Различные SQL-инструкции (SQL-транзакция1)
COMMIT или ROLLBACK
   Различные SQL-инструкции (SQL-транзакция2)
COMMIT или ROLLBACK
   Различные SQL-инструкции (SQL-транзакция3)
COMMIT или ROLLBACK
Конец приложения
```

Я использую термин *SQL-транзакция* по той причине, что приложение может использовать другие инструменты (например, сетевые) и выполнять другие виды транзакций. Далее под термином *транзакция* подразумевается именно SQL-транзакция.

Обычная транзакция SQL может выполняться в одном из двух режимов: `READ-WRITE` (чтение-запись) или `READ-ONLY` (только чтение). Для транзакции 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). По умолчанию действуют установки `READ-WRITE` и `SERIALIZABLE`. Если нужно использовать любые другие установки, то их следует указать с помощью инструкции `SET TRANSACTION`, например:

```
SET TRANSACTION READ ONLY ;
```

или

```
SET TRANSACTION READ ONLY REPEATABLE READ ;
```

или

```
SET TRANSACTION READ COMMITTED ;
```

В одном приложении может находиться множество инструкций `SET TRANSACTION`, но в каждой транзакции можно использовать только одну из них, причем она обязательно должна быть первой выполняемой SQL-инструкцией в транзакции. Если вы хотите использовать инструкцию `SET TRANSACTION`, выполните ее либо в начале приложения, либо после инструкции `COMMIT` или `ROLLBACK`.

Инструкцию `SET TRANSACTION` необходимо выполнять в начале каждой транзакции, для которой используются установки, не совпадающие с предусмотренными по умолчанию. Дело в том, что после инструкции `COMMIT` или `ROLLBACK` каждая новая транзакция автоматически получает установки, действующие по умолчанию.

В инструкции `SET TRANSACTION` можно также задать значение параметра `DIAGNOSTIC SIZE`, определяющего количество сбойных ситуаций, информацию о которых должна сохранять СУБД. Такое ограничение необходимо, потому что при выполнении SQL-инструкции может произойти несколько ошибок. Значение по умолчанию этого параметра определяется конкретной СУБД, поэтому лучше его не трогать.

Транзакция по умолчанию

Параметры транзакции SQL, действующие по умолчанию, обычно подходят для большинства пользователей. Впрочем, при необходимости для транзакции (с помощью инструкции `SET TRANSACTION`) можно задать и другие значения параметров.

Транзакция по умолчанию создается исходя из двух неявных допущений:

- база данных будет со временем изменяться;
- всегда лучше обезопасить себя, чем жалеть впоследствии.

Транзакция по умолчанию устанавливает режим `READ WRITE`, который позволяет выполнять инструкции, изменяющие базу данных. Кроме того, такая транзакция устанавливает уровень изоляции `SERIALIZABLE`, который является максимально безопасным. Значение параметра `DIAGNOSTIC SIZE` зависит от используемой СУБД и описано в ее документации.

Уровни изоляции

В идеальном случае транзакция должна быть полностью независимой от других транзакций, даже от тех, которые выполняются одновременно с ней. Такое положение вещей и называется *изоляцией*. Однако на практике в многопользовательской системе полная изоляция не всегда достижима. Она может обернуться слишком высокой потерей производительности. Здесь снова встает вопрос о компромиссе: какой уровень изоляции нужен на самом деле и какой долей производительности вы готовы при этом пожертвовать.

"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 инструкции `COMMIT`.

Проблема возникает, если первый пользователь прерывает транзакцию и делает откат. Все последующие действия второго пользователя после этого будут выполняться на основе неправильного значения. Приведем в качестве примера следующую ситуацию. Предположим, имеется некое приложение для обслуживания склада. Один пользователь уменьшает количество товара на складе, а второй считывает новое, меньшее значение. Первый пользователь выполняет откат своей транзакции (восстанавливает первоначальное количество), но второй, думая, что товара осталось мало, заказывает его у поставщика, в результате чего на складе образуется товарный излишек. И это еще не худший случай.

ВНИМАНИЕ!

Если вам нужны точные результаты, то уровнем изоляции READ UNCOMMITED лучше не пользоваться.

Уровень READ UNCOMMITED можно применять тогда, когда требуется статистически обрабатывать приблизительные значения следующего характера:

- максимальная задержка в оформлении заказов;
- средний возраст продавцов, не выполняющих норму;
- средний возраст новых сотрудников.

В подобных случаях получения приблизительной информации вполне достаточно. Дополнительное снижение производительности, требуемое для получения более точного результата, не оправдывает себя.

Проблемы неповторяющегося чтения

Следующим, более высоким уровнем изоляции является READ COMMITED: изменение, внесенное другой транзакцией, невидимо для вашей транзакции до тех пор, пока другой пользователь не завершит ее с помощью инструкции COMMIT. Этот уровень обеспечивает лучший результат, чем READ UNCOMMITED, но и он не лишен недостатков.

Для пояснения рассмотрим классический пример с товарами, имеющимися в наличии на складе.

1. Пользователь 1 отправляет запрос в базу данных, чтобы узнать количество единиц некоторого товара, имеющееся на складе. Пусть это количество равно 10.
2. Почти в то же время пользователь 2 начинает, а затем с помощью инструкции COMMIT завершает транзакцию, которая регистрирует заказ на 10 единиц того же товара, уменьшая, таким образом, его запасы до нуля.
3. Далее пользователь 1, думая, что в наличии имеется 10 единиц нужного ему товара, пытается оформить заказ на 5 единиц. Но и такого количества уже нет. Пользователь 2, по сути, опустошил склад.

Первоначальная операция чтения, выполненная пользователем 1 по имевшемуся в наличии количеству, является неповторяющейся. Поскольку это количество было изменено прямо под носом у пользователя 1, все предположения, сделанные на основе полученных им данных, являются неправильными.

Риск фиктивного чтения

Уровень изоляции REPEATABLE READ гарантирует, что проблемы, связанные с неповторяющимся чтением, уже не возникнут. Однако на этом уровне возможно *фиктивное чтение* — проблема, возникающая тогда, когда данные,

считываемые пользователем, изменяются в результате выполнения другой транзакции, причем как раз во время чтения.

Предположим, например, что пользователь 1 выполняет команду, условие отбора которой (предложение `WHERE` или `HAVING`) формирует некий набор строк. И сразу же после этого пользователь 2 выполняет, а затем с помощью инструкции `COMMIT` завершает операцию, в результате которой данные, хранящиеся в одной из этих строк, изменяются. Вначале эти данные удовлетворяли условию отбора, заданному пользователем 1, а теперь уже не удовлетворяют. Или, возможно, некоторые строки, которые вначале не соответствовали этому условию, теперь соответствуют. 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 запускает еще один SQL-запрос. В нем условия отбора те же, что и в первоначальной инструкции, 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 вроде бы надежная информация оказалась негодной.

Обеспечение надежного (хоть и более медленного) чтения

Уровню изоляции `SERIALIZABLE` не свойственны проблемы, характерные для остальных трех уровней. Одновременные транзакции с уровнем `SERIALIZABLE` будут выполняться без вредоносного взаимного влияния, т.е. так, как будто они выполняются не параллельно, а последовательно. Если вы задали этот уровень изоляции, то только отказы оборудования или программ могут привести к невыполнению транзакции, и зная, что ваша система работает корректно, можно не беспокоиться о правильности результатов операций с базой данных.

Конечно, за сверхвысокую надежность приходится расплачиваться снижением производительности, так что во всем нужно знать меру. В табл. 15.1 описаны все четыре уровня изоляции и решаемые ими проблемы.

Таблица 15.1. Уровни изоляции и решаемые ими проблемы

Уровень изоляции	Решаемые проблемы		
	"Черновое" чтение	Неповторяющееся чтение	Фиктивное чтение
`READ UNCOMMITED`	Нет	Нет	Нет
`READ COMMITED`	Да	Нет	Нет

Окончание табл. 15.1

Уровень изоляции	Решаемые проблемы		
	"Черновое" чтение	Неповторяющееся чтение	Фиктивное чтение
REPEATABLE READ	Да	Да	Нет
SERIALIZABLE	Да	Да	Да

Неявная инструкция начала транзакции

Некоторые реализации SQL требуют, чтобы о начале транзакции сообщалось явным образом с помощью специальной инструкции, такой как `BEGIN` или `BEGIN TRAN`. Стандарт SQL этого не требует. Если еще никакая транзакция не начата и на выполнение отправляется соответствующая инструкция SQL, то стандартная SQL-среда начинает выполнение транзакции по умолчанию. Например, инструкции `CREATE TABLE`, `SELECT` и `UPDATE` выполняются только в контексте транзакции. Достаточно запустить одну из них на выполнение, и по умолчанию начнется транзакция.

Инструкция `SET TRANSACTION`

Время от времени для транзакции приходится использовать параметры, отличающиеся от установленных по умолчанию. Эти нестандартные параметры можно задавать с помощью инструкции `SET TRANSACTION`, которая должна быть первой инструкцией транзакции. Инструкция `SET TRANSACTION` позволяет задавать режим, уровень изоляции и размер области диагностики.

Чтобы изменить, например, все три параметра, можно выполнить следующую инструкцию.

```
SET TRANSACTION
   READ ONLY,
   ISOLATION LEVEL READ UNCOMMITTED,
   DIAGNOSTICS SIZE 4 ;
```

Эти настройки не позволяют изменять базу данных (режим `READ ONLY`). Кроме того, здесь устанавливается самый слабый и, следовательно, наиболее опасный уровень изоляции (`READ UNCOMMITED`). Для области диагностики выбран размер 4. Как видите, параметры подобраны таким образом, чтобы транзакция использовала поменьше системных ресурсов.

Теперь сравните предыдущую инструкцию с приведенной ниже.

```
SET TRANSACTION
   READ WRITE,
   ISOLATION LEVEL SERIALIZABLE,
   DIAGNOSTICS SIZE 8 ;
```

Эти установки позволяют изменять базу данных (READ WRITE), задают наивысший уровень изоляции (SERIALIZABLE) и назначают большую область диагностики. Эта транзакция предъявляет гораздо более высокие требования к системным ресурсам. Эти значения могут оказаться действующими по умолчанию — все зависит от конкретной СУБД. Естественно, в инструкции SET TRANSACTION можно использовать и другие значения уровня изоляции и размера области диагностики.


СОВЕТ

Не задавайте уровень изоляции выше необходимого. На первый взгляд кажется, что для надежности всегда лучше выбирать значение SERILIZABLE. Но это может оказаться плохой идеей (хотя все зависит от конкретной СУБД и от решаемой вами задачи), поскольку такой высокий уровень изоляции может сильно снизить общую производительность системы. Если в своей транзакции вы не собираетесь модифицировать базу данных, смело задавайте режим READ ONLY. Не позволяйте базе данных без необходимости "съедать" системные ресурсы.

Инструкция COMMIT

Стандарт SQL не требует явного использования ключевого слова для обозначения начала транзакции, но зато существуют две инструкции ее завершения: COMMIT и ROLLBACK. Первая из них используется тогда, когда вы уже дошли до конца транзакции и собираетесь подтвердить изменения, внесенные в базу данных. Инструкция COMMIT может включать необязательное ключевое слово WORK (COMMIT WORK). Если при выполнении инструкции COMMIT произойдет ошибка или системный сбой, вам придется выполнить откат транзакции и повторить ее снова.

Инструкция ROLLBACK

В конце транзакции иногда требуется отменить изменения, внесенные в базу данных во время ее выполнения, т.е. восстановить базу данных в том состоянии, в котором она была перед самым началом транзакции. Для этого можно воспользоваться инструкцией ROLLBACK, которая является отказоустойчивой.


СОВЕТ

Даже если во время выполнения инструкции ROLLBACK в системе произойдет аварийный сбой, после перезагрузки компьютера инструкцию можно будет запустить снова — она должна вернуть базу данных в то состояние, в котором она находилась перед началом транзакции.

Блокировка объектов базы данных

Уровень изоляции, который установлен по умолчанию или с помощью инструкции SET TRANSACTION, указывает СУБД, сколько усилий ей нужно приложить, чтобы ваша работа не пересекалась с работой других пользователей. Главная защита со стороны СУБД от нежелательных транзакций — это блокировка используемых вами объектов базы данных. Приведем несколько примеров такой блокировки.

- Заблокированной может быть табличная строка, с которой вы работаете в текущий момент времени. Пока вы ее используете, никто другой не имеет к ней доступа.
- Заблокированной может быть вся таблица, если вы выполняете операцию, которая влияет на таблицу в целом.
- Все операции записи блокируются, а чтение табличных данных допускается. Иногда же запись разрешена, а чтение блокируется.

В каждой СУБД управление блокировкой реализовано по-своему. И хотя в одних реализациях "броня" толще, чем в других, большинство нынешних СУБД в состоянии защитить данные от негативных последствий одновременного доступа.

Резервное копирование данных

Резервное копирование — это комплекс мер, регулярно выполняемый администратором базы данных. Резервное копирование всех элементов системы должно проводиться через определенные интервалы времени, длительность которых зависит от частоты обновления этих элементов. Если ваша база данных обновляется ежедневно, то и ее резервное копирование следует проводить ежедневно. Приложения, формы и отчеты тоже могут меняться, хотя и не столь часто. Администратор должен успевать создавать и их резервные копии.


СОВЕТ

Храните несколько версий резервных копий. Иногда ущерб, причиненный базе данных, становится очевидным лишь спустя некоторое время. Чтобы вернуться к последней работоспособной версии, вам, возможно, придется откатиться на несколько версий назад.

Существует множество способов резервного копирования.

- Создание резервных таблиц и копирование в них данных с помощью SQL.
- Использование определяемого реализацией механизма, который создает резервную копию всей базы данных или ее фрагментов. Обычно этот механизм намного удобнее и эффективнее, чем инструментарий SQL.
- В вашей операционной системе может быть механизм, предназначенный для полного резервного копирования системы, 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.

Точки сохранения и субтранзакции

В идеальном случае транзакции должны быть атомарными — такими же неделимыми, какими представлялись атомы древним грекам. Однако в действительности атомы неделимыми не являются. С выходом стандарта SQL:1999 транзакции в базах данных также перестали быть атомарными. Транзакция теперь может состоять из нескольких *субтранзакций*. Каждая субтранзакция завершается инструкцией `SAVEPOINT`. Она используется в сочетании с инструкцией `ROLLBACK`. До появления *точек сохранения* (точек программы, в которых вступает в силу инструкция `SAVEPOINT`) инструкция `ROLLBACK` применялась только для отмены всей транзакции, теперь же ее можно использовать для отката транзакции до точки сохранения. Вы спросите, для чего это нужно?

В основном инструкция `ROLLBACK` используется для защиты данных от повреждения путем восстановления базы данных после прерывания транзакции из-за ошибки. Естественно, если во время транзакции произошла ошибка, то бессмысленно делать откат только к ближайшей точке сохранения. Ведь для того, чтобы вернуть базу данных в состояние, в котором она находилась перед началом транзакции, нужен откат *всей* транзакции. Впрочем, бывают ситуации, когда имеет смысл откатить именно *часть* транзакции.

Предположим, вы выполняете над данными сложную последовательность операций. В середине этого процесса вы получаете результаты, которые свидетельствуют о том, что вами выбран неверный путь. Если бы вы были более предусмотрительны и установили точку сохранения (с помощью инструкции `SAVEPOINT`) в соответствующем месте этой последовательности, то можно было бы выполнить откат к ней, а затем попробовать другой вариант. Если нужно изменить действие лишь части кода, а остальной код (предшествующий проблемному) является безупречным, то лучше поступить так, чем отменять всю текущую транзакцию и запускать новую ради попытки пойти другим путем.

Чтобы поместить в SQL-код точку сохранения, 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 инструкцию SAVEPOINT. Если в вашей СУБД ее нет, то использовать ее вы, разумеется, не сможете.

ТЕХНИЧЕСКИЕ ПОДРОБНОСТИ

СВОЙСТВА ACID

Возможно, вы слышали от разработчиков, что базам данных нужна ACID (англ. "acid" — кислота). Нет, конечно, они не собираются подвергать свои творения психоделическим эффектам в стиле 1960-х годов или растворять в пузырящемся месиве содержимое базы данных. ACID — это аббревиатура, образованная от слов "*a*tomicity" (атомарность), "*c*onsistency" (согласованность), "*i*solation" (изоляция) и "*d*urability" (надежность). Эти четыре характеристики необходимы для защиты базы данных от повреждения.

- **Атомарность**. Транзакции в базах данных должны быть атомарными в классическом понимании этого слова. Это значит, что вся транзакция обрабатывается как неделимая единица. Либо она выполняется целиком, либо база данных восстанавливается в том состоянии, в котором она была бы, если бы транзакция не выполнялась вообще.
- **Согласованность**. Как ни странно, само значение термина "согласованность" непостоянно; в разных приложениях он трактуется по-разному. Например, при переводе денежных средств в банковском приложении с одного счета на другой нужно, чтобы общая сумма денег на обоих счетах в конце транзакции осталась такой же, какой была в ее начале. В другом приложении критерий согласованности может быть другим.
- **Изоляция**. Транзакции в базах данных должны быть (в идеале) полностью изолированы от других транзакций, выполняемых в то же время. Если выполнять транзакции последовательно, то можно добиться полной изоляции. Если же система должна обрабатывать транзакции с максимальной скоростью, то увеличение производительности иногда достигается за счет более низкого уровня изоляции.
- **Надежность**. Необходимо, чтобы после завершения или отката транзакции база данных находилась в рабочем состоянии, т.е. содержала неповрежденные, надежные и обновленные данные. Даже если в системе во время транзакции произойдет аварийный сбой, надежная СУБД должна гарантировать возвращение базы данных в рабочее состояние.

Ограничения в транзакциях

Проверка достоверности данных заключается не только в проверке правильности их типа. Возможно, также потребуется следить за тем, чтобы в одних столбцах не было пустых значений, а в других значения не выходили за пределы определенного диапазона (см. главу 5).

Говорить о связи ограничений с транзакциями имеет смысл потому, что первые, оказывая влияние на работу последних, не позволяют пользователям совершать противоправные действия. Предположим, вам нужно добавить данные в таблицу, в которой имеется столбец с ограничением NOT NULL. Иногда для ввода записи сначала создают в таблице пустую строку, а затем заполняют ее значениями. Однако ограничение NOT NULL не позволит воспользоваться данным методом, поскольку SQL не даст ввести строку с пустым значением в столбце с ограничением NOT NULL, даже если данные в этот столбец предполагается ввести еще до конца транзакции. Для решения этой проблемы в SQL существует возможность определять ограничения как допускающие задержку (DEFERRABLE) или не допускающие ее (NOT DEFERRABLE).

Ограничения, определенные как NOT DEFERRABLE, применяются немедленно, а определенные как DEFERRABLE первоначально могут быть заданы как DEFERRED (отсроченные)или IMMEDIATE (немедленные). Если ограничение типа DEFERRABLE задано как IMMEDIATE, то оно действует подобно ограничению NOT DEFERRABLE — немедленно. Если же ограничение типа DEFERRABLE задано как DEFERRED, то его действие может быть отсрочено.

Для добавления пустых записей или выполнения других операций, которые временно могут противоречить ограничениям типа DEFERRABLE, используйте следующую инструкцию:

```
SET CONSTRAINTS ALL DEFERRED ;
```

Она определяет все ограничения типа DEFERRABLE как DEFERRED. На ограничения типа NOT DEFERRABLE эта инструкция не действует. После выполнения всех операций, способных нарушить ваши ограничения, и достижения таблицей того состояния, когда они уже не нарушены, эти ограничения можно применить заново. Соответствующая инструкция выглядит так:

```
SET CONSTRAINTS ALL IMMEDIATE ;
```

Если вы допустили ошибку и данные не соответствуют каким-либо ограничениям, вы это обнаружите сразу же после выполнения указанной инструкции.

Если ограничения, ранее установленные как DEFERRED, явно не задаются вами как IMMEDIATE, то SQL сделает это за вас, т.е. применит все отсроченные ограничения при попытке завершить транзакцию с помощью инструкции

COMMIT. Если к этому моменту все еще имеет место нарушение ограничений, транзакция будет отменена и SQL выдаст сообщение об ошибке.

Механизм ограничений в SQL защищает от ввода некорректных данных (или, что так же важно, от недопустимого *отсутствия* данных), позволяя при этом по ходу транзакции временно нарушать установленные ограничения.

Чтобы продемонстрировать, насколько важна возможность отсрочки ограничений, рассмотрим пример с платежными ведомостями.

Предположим, в таблице EMPLOYEE (сотрудники) существуют столбцы EmpNo (идентификатор сотрудника), EmpName (фамилия сотрудника), DeptNo (номер отдела) и Salary (оклад). Столбец DeptNo является внешним ключом, который ссылается на таблицу DEPT (отделы). Допустим также, что в таблице DEPT существуют столбцы DeptNo и DeptName (название отдела), причем DeptNo — это первичный ключ.

Вы хотите иметь такую же таблицу, как и DEPT, но с еще одним столбцом, Payroll (платежная ведомость), в котором для каждого отдела имеется сумма значений Salary сотрудников этого отдела.

Эквивалент такой таблицы можно создать с помощью следующего представления (в предположении, что вы используете СУБД, которая поддерживает рассматриваемую функциональность).

```
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 одним способом.

```
CREATE VIEW DEPT3 AS
     SELECT D.*,
           (SELECT SUM(E.Salary)
              FROM EMPLOYEE E
             WHERE D.DeptNo = E.DeptNo) AS Payroll
       FROM DEPT D ;
```

Теперь предположим, что для большей эффективности вы не собираетесь вычислять значение функции SUM каждый раз, когда ссылаетесь на столбец DEPT3.Payroll. Вместо этого вы хотите, чтобы сама таблица DEPT содержала столбец Payroll. Значения в этом столбце вы будете обновлять при каждом внесении изменений в столбец Salary.

Чтобы обеспечить правильность значений в столбце Salary, в определение таблицы можно вставить ограничение.

```
CREATE TABLE DEPT
   (DeptNo CHAR(5),
```

```
    DeptNameCHAR(20),
    Payroll DECIMAL(15, 2),
    CHECK (Payroll = (SELECT SUM(Salary)
                        FROM EMPLOYEE E
                       WHERE E.DeptNo= DEPT.DeptNo)));
```

Теперь предположим, что вам нужно увеличить на 100 значение `Salary` для сотрудника с идентификатором `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 = '123') ;
```

(Подзапрос используется для того, чтобы получить ссылку на значение `DeptNo` сотрудника с идентификатором `123`.)

Здесь возникает трудность: ограничения проверяются после каждой инструкции. Теоретически должны проверяться *все* ограничения. Но на практике СУБД проверяет только те из них, которые относятся к значениям, изменяемым в ходе действия инструкции.

После первой из двух приведенных выше инструкций `UPDATE` СУБД проверяет все ограничения, которые имеют отношение к измененным значениям. В число этих ограничений входит то, которое определено в таблице `DEPT`, поскольку оно относится к столбцу `Salary` таблицы `EMPLOYEE`, а значения в этом столбце изменяются инструкцией `UPDATE`. После выполнения первой инструкции `UPDATE` это ограничение оказалось нарушенным. Допустим, перед ее выполнением база данных находится в полном порядке и каждое значение `Payroll` в таблице `DEPT` равно сумме значений `Salary` из соответствующих строк таблицы `EMPLOYEE`. Как только первая инструкция `UPDATE` увеличит значение `Salary`, это равенство выполняться не будет. Такая ситуация исправляется второй инструкцией `UPDATE`, после выполнения которой значения базы данных будут соответствовать заданным ограничениям. Но в промежутке между этими обновлениями ограничения нарушены.

Инструкция `SET CONSTRAINTS DEFERRED` позволяет временно отключить все или только заданные ограничения. Действие этих ограничений задерживается до тех пор, пока не будет выполнена или инструкция `SET CONSTRAINTS IMMEDIATE`, или одна из инструкций `COMMIT` и `ROLLBACK`. Поэтому предыдущие

две инструкции UPDATE мы должны поместить между инструкциями 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 таблицу 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 нужно отсрочить. Для этого при создании ограничений им следует присваивать 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 = '123' ;
UPDATE DEPT D
   SET Payroll = Payroll + 100
 WHERE D.DeptNo = (SELECT E.DeptNo
                     FROM EMPLOYEE E
                    WHERE E.EmpNo = '123') ;
   SET CONSTRAINTS PayEqSumsal IMMEDIATE;
```

Если для ограничения в инструкции CREATE не указано имя, то SQL создает его неявно. Оно находится в специальных таблицах каталога. Но все-таки лучше явно задавать имена ограничений.

Теперь предположим, что во второй инструкции UPDATE в качестве значения приращения вы по ошибке указали число 1000. Это значение в ней является

разрешенным, поскольку существующее ограничение было отсрочено. Но при выполнении инструкции SET CONSTRAINTS...IMMEDIATE заданные ограничения будут проверены. Если окажется, что они не соблюдены, инструкция SET CONSTRAINTS сгенерирует исключение. Если же вместо инструкции SET CONSTRAINTS...IMMEDIATE выполняется инструкция COMMIT, а ограничения не соблюдаются, то вместо COMMIT выполнится инструкция ROLLBACK.


ЗАПОМНИ!

Подведем итог. Временно отменять действие ограничений можно лишь в транзакциях. Как только транзакция завершается (выполнением инструкции ROLLBACK или COMMIT), ограничения сразу же вступают в силу. Если правильно использовать эту возможность, то транзакция не создаст данных, которые нарушали бы какие-либо ограничения, доступные для других транзакций.

Предотвращение внедрения зловредного SQL-кода

Довольно сложно защитить данные от нестабильности платформы, отказа оборудования и одновременного доступа. Но что если кто-то намеренно пытается украсть или испортить ваши данные либо причинить вам какой-то вред? Это может вызвать гораздо более серьезные проблемы. Существует множество способов, которыми злоумышленник может атаковать компьютерную систему, но наиболее типичной для приложений базы данных является атака типа внедрения SQL-кода.

При атаке типа внедрения SQL-кода злоумышленник пытается внедрить вредоносный код в приложение базы данных. Такой код способен передать управление базой данных злоумышленнику, после чего тот может тайком изменить данные в ущерб владельцу или пользователям либо просто удалить целые таблицы.

Слабым местом любого приложения является ввод данных от пользователя, включая запрос учетных данных для входа. Как разработчик приложения, кодирующий текстовые поля для ввода пользователем имени пользователя и пароля, вы ожидаете, что пользователь введет собственно имя и пароль. Однако хакер может ввести что-то, чего вы не ожидаете, что-то, что заставит приложение реагировать так, чтобы сообщить злоумышленнику требуемые ему данные. Эта информация позволит ему проникнуть немного дальше. Как только он получает права системного администратора, игра заканчивается, и ваши данные оказываются в его власти.

В атаках типа внедрения SQL-кода используются преимущества наличия динамического SQL в коде приложения. Существуют два вида SQL-кода: статический и динамический. *Статический SQL-код* жестко запрограммирован в приложении и становится фиксированным на этапе компиляции, поэтому его трудно взломать. А вот *динамический SQL-код* компонуется и запускается на этапе выполнения программы. Атака типа внедрения SQL-код использует этот факт, добавляя специальный код во вводимые данные. Этот дополнительный код включается в динамическую инструкцию SQL, которая должна принимать данные, вводимые в текстовое поле. Атакующий код может инициировать утечку секретной информации или даже уничтожение базы данных.

Существуют средства защиты от атак типа внедрения SQL-кода. Прежде всего, они включают тщательную проверку любых пользовательских данных перед их включением в динамическую инструкцию SQL. Дополнительные сведения по этой теме можно найти в Интернете.

Глава 16

Использование SQL в приложениях

В ЭТОЙ ГЛАВЕ...

- SQL в приложении
- Совместное использование SQL с процедурными языками
- Как избежать несовместимости языков
- Встраивание SQL в процедурный код
- Вызов SQL-модулей из процедурного кода
- Выполнение SQL-инструкций в среде быстрой разработки

В предыдущих главах мы большей частью рассматривали инструкции SQL по отдельности, т.е. формулировали конкретные вопросы касательно данных и для получения ответов на эти вопросы создавали SQL-запросы. Такой подход, предполагающий интерактивное использование SQL, прекрасно подходит для изучения возможностей языка, но на практике SQL применяют по-другому.

Несмотря на то что синтаксис SQL напоминает синтаксис английского языка, изучить его все равно нелегко. Сегодня подавляющее большинство пользователей не владеют им в достаточной мере, и можно предположить, что даже если эта книга и завоюет широкую популярность, то все равно основная масса пользователей компьютеров так никогда и не освоит SQL. Если рядовому пользователю поставить задачу, связанную с базой данных, то он, скорее всего, и не подумает садиться за терминал, чтобы ввести SQL-инструкцию `SELECT`. Системные аналитики и разработчики приложений, свободно владеющие SQL,

также не занимаются вводом разовых запросов с консоли — они разрабатывают приложения, реализующие такие запросы.


СОВЕТ

Чтобы многократно выполнять одну и ту же операцию, не обязательно каждый раз вводить ее заново с клавиатуры. Напишите приложение и затем запускайте его столько раз, сколько нужно. SQL-код может быть частью такого приложения, но в этом случае он работает не совсем так, как в интерактивном режиме.

SQL в приложении

В главе 2 SQL был назван неполным языком программирования. Для использования в приложении его необходимо объединить с *процедурным* языком, таким как Visual Basic, C, C++, C#, Java или Python. 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 с SQL использовать процедурные языки, то можно будет создавать мощные приложения с широким набором возможностей. Недавно появились объектно-ориентированные инструменты RAD (Rapid Application Development — быстрая разработка приложений), которые позволяют встраивать SQL-код в приложения, создаваемые из отображаемых на экране объектов, а не посредством написания процедурного кода. В качестве примера таких инструментов можно привести Microsoft Visual Studio и среду разработки Eclipse с открытым исходным кодом.

Следите за звездочкой

В предыдущих главах мы часто использовали символ "звездочка" (*) для краткого обозначения всех столбцов в таблице. Если в таблице много столбцов, то звездочка позволяет сократить объем вводимого кода. Если же SQL используется в приложении, лучше раз и навсегда отказаться от звездочки. После того как приложение будет написано, вы или кто-то другой, возможно, добавите в таблицу новые столбцы или удалите старые, и в этом случае значение понятия "все столбцы" изменится. В результате приложение, использующее звездочку для обозначения всех столбцов, может получить совсем не тот результат, на который рассчитывает.

Такое изменение в таблице не окажет влияния на уже существующие программы, пока эти программы не будут перекомпилированы (чтобы исправить

обнаруженную ошибку или внести какие-либо усовершенствования). Тогда воздействие символа шаблона (*) распространится на *новый состав* всех столбцов таблицы. Это изменение может привести к аварийному завершению работы приложения, не связанному с текущей отладкой (т.е. с исправлением ошибки или текущими усовершенствованиями), либо иметь другие последствия, которые превратят отладку программы в настоящий кошмар.


СОВЕТ

Чтобы обезопасить себя, откажитесь от использования звездочки и указывайте имена всех столбцов в *явном* виде (подробнее о метасимволах см. в главе 6).

Преимущества и недостатки SQL

Основное достоинство SQL заключается в извлечении (или выборке) данных. Если важная информация хранится где-то в недрах одно- или многотабличной базы данных, то ее можно найти и извлечь с помощью инструкций SQL. Особенности этого языка таковы, что вам не обязательно знать порядок расположения строк или столбцов в таблице. Реализованный в SQL механизм обработки транзакций гарантирует, что на операции, выполняемые с базой данных одним пользователем, не повлияют любые другие пользователи, которые одновременно с первым стремятся получить доступ к тем же самым таблицам.

В то же время главным недостатком SQL является его недоразвитый интерфейс пользователя. В SQL нет инструментов для форматирования выводимой информации и генерации отчетов. Результат выполнения запроса передается на экран построчно.

Порой качество, которое в одной ситуации является несомненным достоинством, в другой может оказаться недостатком. Например, одно из преимуществ SQL заключается в возможности работать сразу с целой таблицей. Сколько бы в таблице ни было строк — одна, сто или сто тысяч, — нужные данные можно извлечь из нее с помощью единственной инструкции `SELECT`. Однако SQL не позволяет с такой же легкостью работать с отдельными строками. В таких задачах придется использовать или курсоры, которые описаны в главе 19, или процедурный язык программирования.

Сильные и слабые стороны процедурных языков

В отличие от SQL процедурные языки легко оперируют отдельными строками, позволяя разработчику приложения точно управлять способом обработки таблицы. Детализированное управление является несомненным достоинством процедурных языков. Впрочем, и здесь есть свои слабые стороны. Разработчик приложения должен точно знать, каким образом данные хранятся в таблицах.

В этом случае порядок расположения строк и столбцов имеет значение, и его приходится учитывать.


ЗАПОМНИ!

Благодаря своей "пошаговой" природе процедурные языки являются очень гибкими, и с их помощью можно создавать удобные интерфейсы ввода и просмотра данных. Кроме того, для вывода на печать они позволяют создавать очень сложные отчеты, имеющие любую требуемую структуру.

Проблемы, возникающие при совместном использовании SQL с процедурными языками

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 полезное для нас объединение на практике, приходится преодолевать определенные трудности.

Различные режимы работы

При объединении SQL с процедурными языками немалая трудность связана с тем, что SQL одновременно работает с целым набором таблиц, в то время как процедурные языки — только с одной строкой одной таблицы. Иногда это не так уж и важно. Операции с наборами таблиц можно выполнять отдельно от операций со строками, используя в каждом случае соответствующий инструмент.

Однако в некоторых случаях приходится проверять каждую запись таблицы на соответствие определенным условиям и в зависимости от этого выполнять с ней те или иные действия. Для такого процесса требуются как возможности SQL по извлечению данных, так и инструменты ветвления процедурных языков. Эту комбинацию возможностей можно получить с помощью SQL-кода, который встроен в программу, написанную на обычном процедурном языке (более подробно мы поговорим об этом в разделе "Внедрение SQL-кода").

Несовместимость типов данных

Еще одно препятствие на пути к интеграции SQL с любым процедурным языком состоит в том, что типы данных SQL отличаются от типов данных большинства процедурных языков программирования. Это неудивительно, ведь даже типы данных одного процедурного языка отличаются от типов данных другого.

ЗАПОМНИ!

Общего стандарта типов данных в языках программирования не существует. В ранних версиях SQL, предшествовавших SQL-92, одной из основных проблем была именно несовместимость данных. Однако в SQL-92 (а также в последующих реализациях стандарта) эта проблема стала решаться с помощью инструкции `CAST`. В главе 9 мы уже говорили о том, как использовать инструкцию `CAST` для преобразования данных процедурных языков в типы, распознаваемые в SQL, разумеется, если эти типы потенциально совместимы.

Вставка инструкций SQL в процедурные языки

Несмотря на то что процесс интеграции SQL в процедурные языки связан с определенными трудностями, поверьте — эта задача выполнима. Во многих случаях такая интеграция — единственный способ решения поставленной задачи за разумное время. О трех методах подобной интеграции — внедрении SQL-кода, модульном языке и инструментах RAD — мы поговорим в следующих разделах.

Внедрение SQL-кода

Самый распространенный способ объединения SQL с процедурными языками называется *внедрением SQL-кода*. Как следует из названия, инструкции SQL просто вставляются в нужные места процедурной программы.

Естественно, внезапно появившись в программе, написанной, скажем, на языке C, инструкция SQL может стать "сюрпризом" для компилятора. По этой причине программы с внедренным SQL-кодом перед компиляцией или интерпретацией пропускают через *препроцессор*. О встроенном SQL-коде препроцессор предупреждается директивой `EXEC SQL`.

В качестве примера использования встроенного SQL-кода рассмотрим программу, написанную на языке Pro*C компании Oracle, являющемся вариантом языка C. Эта программа получает доступ к таблице с данными о сотрудниках компании (`EMPLOYEE`), предлагает пользователю ввести имя сотрудника, а затем выводит оклад и размер комиссионного вознаграждения для этого сотрудника. Затем она предлагает ввести новые значения оклада и комиссионных этого же сотрудника и обновляет 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; /* Код, возвращаемый функцией scanf */
     /* Регистрация */
     strcpy(uid.arr, "FRED");     /* Копируем имя пользователя */
     uid.len=strlen(uid.arr);
     strcpy(pwd.arr, "TOWER");  /* Копируем пароль */
     pwd.len=strlen(pwd.arr);
     EXEC SQL WHENEVER SQLERROR STOP;
     EXEC SQL WHENEVER NOT FOUND STOP;
     EXEC SQL CONNECT :uid;
     printf("Подключение для пользователя: percents \n", uid.arr);
     printf("Введите имя сотрудника для обновления:   ");
     scanf("percents", ename.arr);
     ename.len=strlen(ename.arr);
     EXEC SQL SELECT SALARY, COMM INTO :salary, :comm
                FROM EMPLOYEE
               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;
     if (sret == EOF !! sret == 0) /* Устанавливаем индикатор */
        salary_ind = -1; /* Устанавливаем индикатор для значений NULL */
     printf("Введите новые комиссионные: ");
     sret=scanf("percentf", &comm);
     comm_ind = 0;        /* Устанавливаем индикатор */
     if (sret == EOF !! sret == 0)
        comm_ind = -1; /* Устанавливаем индикатор для значений NULL */
     EXEC SQL UPDATE EMPLOYEE
                 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 нужно быть экспертом по языку C, чтобы понять суть того, что (и как) делает эта программа. Ниже описана последовательность выполнения инструкций.

1. SQL объявляет базовые переменные.
2. Код C выполняет процедуру регистрации пользователя.
3. SQL подготавливает обработку ошибок и подключается к базе данных.
4. Язык C запрашивает у пользователя имя сотрудника, которое заносится в переменную `:ename`.
5. Инструкция `SQL SELECT` извлекает данные об окладе и комиссионных указанного сотрудника и сохраняет их в двух базовых переменных, `:salary` и `:comm`.
6. Далее код C отображает имя сотрудника, его оклад и комиссионные, а затем запрашивает у пользователя их новые значения. После этого проверяется, введены ли затребованные значения, и если нет, то устанавливается индикатор отсутствия ввода.
7. На основе введенных значений SQL обновляет базу данных.
8. Код C отображает сообщение о завершении операции.
9. SQL подтверждает транзакцию, и код C завершает выполнение программы.


СОВЕТ

Смешивать команды двух языков так, как здесь, можно благодаря препроцессору. Он отделяет инструкции SQL от команд базового языка, вынося их в отдельную внешнюю процедуру. Каждая инструкция SQL заменяется вызовом соответствующей внешней процедуры. Теперь за свою работу может приниматься компилятор этого языка.


ЗАПОМНИ!

Способ, с помощью которого инструкции SQL передаются базе данных, зависит от конкретной реализации. Однако это совершенно не должно беспокоить разработчика приложений. Все эти задачи берет на себя препроцессор. Вам нужно лишь позаботиться о таких вещах, с которыми не приходится сталкиваться при интерактивном использовании SQL: о базовых переменных и совместимости типов данных.

Объявление базовых переменных

Между программой, написанной на процедурном языке, и SQL-кодом должна передаваться информация. Для этого используются базовые переменные. Чтобы SQL распознал эти переменные, перед использованием их необходимо объявить. Объявления находятся в блоке объявлений, который расположен перед программным блоком. Блок объявлений начинается следующей директивой.

```
EXEC SQL BEGIN DECLARE SECTION ;
```

Конец этого блока помечается с помощью такой директивы.

```
EXEC SQL END DECLARE SECTION ;
```

Перед каждой SQL-инструкцией должна стоять директива EXEC. Конец SQL-блока может быть, а может и не быть помечен директивой завершения. В языке COBOL такой директивой является END-EXEC, а в языке C это точка с запятой.

Преобразование типов данных

В зависимости от совместимости типов данных, поддерживаемых базовым языком и SQL, для преобразования некоторых из них придется задействовать инструкцию CAST. Можно применять базовые переменные, которые были объявлены с помощью директивы DECLARE SECTION, но при использовании их в SQL-инструкциях не забывайте ставить перед их именами двоеточие (:), как в следующем примере.

```
INSERT INTO FOODS
      (FOODNAME, CALORIES, PROTEIN, FAT, CARBOHYDRATE)
VALUES
      (:foodname, :calories, :protein, :fat, :carbo) ;
```

Модульный язык

Использовать SQL совместно с процедурным языком программирования можно и по-другому — с помощью *модульного языка*. Он позволяет разместить все инструкции SQL в отдельном модуле.


ЗАПОМНИ!

Модуль SQL представляет собой список SQL-инструкций. Каждая из этих инструкций включается в SQL-процедуру, в спецификации которой определены ее имя, а также имена и типы ее параметров.

В любой SQL-процедуре содержится только одна SQL-инструкция. В любом месте программы на базовом языке можно явно вызвать эту процедуру. Вызов SQL-процедуры подобен вызову обычных подпрограмм, написанных на базовом языке.

Таким образом, использование модуля SQL вместе со связанной с ним программой на базовом языке сводится к написанию вручную того кода, который получается после обработки препроцессором внедренного SQL-кода.


СОВЕТ

Внедренный SQL-код получил большее распространение, чем модульный язык. И хотя большинство поставщиков предлагают какой-либо вариант модульного языка, мало кто из них в своей документации уделяет ему особое внимание. Тем не менее модульные языки обладают рядом преимуществ.

- **SQL-программисты не обязаны быть экспертами в процедурных языках.** Так как SQL полностью отделен от процедурного языка, к написанию модулей SQL можно привлечь самых лучших специалистов в этой области. Причем не важно, имеют они опыт работы с процедурным языком или нет. В действительности решение вопроса о том, какой процедурный язык использовать, можно даже отложить до того времени, пока не будут написаны и отлажены модули SQL.
- **К написанию программы на процедурном языке можно привлечь специалиста, который совершенно не знаком с SQL.** Так же, как специалисты в области SQL могут не знать процедурный язык, программисты процедурного языка могут совершенно не волноваться об инструкциях SQL.
- **Процедурный код не содержит фрагментов с инструкциями SQL, что позволяет использовать стандартный отладчик процедурного языка.** Это существенно сокращает время, необходимое для разработки программы.


ЗАПОМНИ!

То, что с одной стороны может выглядеть как преимущество, с другой может быть расценено как недостаток. Поскольку модули SQL отделены от процедурного кода, понять логику работы такой программы будет труднее, чем при использовании внедренного SQL.

Объявления модулей

Синтаксис объявлений в модулях имеет следующий вид.

```
MODULE [имя_модуля]
   [NAMES ARE имя_набора_символов]
   LANGUAGE {ADA|C|COBOL|FORTRAN|MUMPS|PASCAL|PLI|SQL}
   [SCHEMA имя_схемы]
   [AUTHORIZATION идентификатор_подтверждения_полномочий]
   [объявления_временных_таблиц...]
   [объявления_курсоров...]
   [объявления_динамических_курсоров...]
   процедуры...
```

Квадратные скобки означают, что имя модуля не является обязательным. Тем не менее лучше присвоить модулю имя, чтобы впоследствии избежать недоразумений.


СОВЕТ

Необязательное предложение `NAMES ARE` определяет набор символов, используемый для имен. Если опустить это предложение, будет использован набор символов, установленный в вашей СУБД по умолчанию. Предложение `LANGUAGE` сообщает модулю, на каком

языке написаны программы, которые будут его вызывать. Компилятору нужно обязательно знать, какой это язык, поскольку он преобразует инструкции SQL так, чтобы для вызывающей программы они выглядели как обычные подпрограммы, написанные на том же языке, что и сама программа.

Несмотря на то что предложения SCHEMA и AUTHORIZATION являются необязательными, непременно стоит использовать хотя бы одно из них. Первое из них задает схему по умолчанию, а второе — *идентификатор подтверждения полномочий*, который определяет привилегии модуля. Если его не указать, то для предоставления вашему модулю привилегий СУБД будет использовать идентификатор полномочий текущего сеанса. Если у вас нет прав на выполнение операции, которую описывает ваша процедура, то, разумеется, она и не выполнится.


СОВЕТ

Если вашей процедуре нужны временные таблицы, объявите их с помощью специального предложения. Объявляйте курсоры и динамические курсоры *перед* объявлением процедуры, в которой они будут использованы. Если объявить курсор *после* процедуры, то ее выполнение будет разрешено до тех пор, пока она не попытается использовать курсор. (Более подробная информация о курсорах содержится в главе 19.)

Модульные процедуры

И наконец, после всех этих объявлений в модуле находится его функциональная часть — сами процедуры. У процедуры в модульном SQL-коде есть имя, объявления параметров и выполняемые SQL-инструкции. Программа, написанная на процедурном языке, вызывает процедуру по имени и передает ей значения через объявленные параметры. Синтаксис процедуры выглядит следующим образом.

```
PROCEDURE имя_процедуры
   (объявление_параметра[, объявление_параметра]...)
   SQL-инструкция ;
   [SQL-инструкции] ;
```

Объявление параметра должно иметь такой вид:

```
имя_параметра тип_данных
```

или такой:

```
SQLSTATE
```

Объявляемые вами параметры могут быть входными, выходными или и теми и другими одновременно. SQLSTATE является параметром состояния, с помощью которого выводится информация об ошибках. (Более подробно о параметрах мы поговорим в главе 21.)

Объектно-ориентированные инструменты быстрой разработки

Используя современные инструменты быстрой разработки, можно создавать сложные приложения и при этом не знать, как написать хотя бы одну строку кода на процедурном языке (C++, C#, Python или Java). Вместо того чтобы писать код, вы выбираете из библиотеки объекты и помещаете их в нужные места на экране.


ЗАПОМНИ!

У объектов стандартных типов есть характерные особенности (свойства), и каждому типу объектов свойственны собственные события. С объектом можно также связать какой-либо метод. *Метод* — это процедура, написанная на процедурном языке (простите за тавтологию). В то же время можно создавать очень сложные приложения, не написав при этом ни одного метода.


СОВЕТ

Несмотря на то что можно создавать сложные приложения, не пользуясь процедурным языком, рано или поздно SQL вам все же понадобится. SQL настолько богат функциональными средствами, что их трудно или даже невозможно выразить в объектной парадигме. Поэтому полнофункциональные инструменты быстрой разработки позволяют вставлять SQL-инструкции в объектно-ориентированные приложения. Примером объектно-ориентированной среды, в которой можно работать с SQL, может служить Visual Studio. Microsoft Access — это еще одна среда разработки приложений, позволяющая использовать SQL совместно с процедурным языком VBA.

В главе 4 описывалось создание таблиц базы данных с помощью Access. Это лишь малая часть широких возможностей Access. Основная цель этой программы — разработка приложений, предназначенных для обработки записей в таблицах базы данных. Используя Access, можно поместить объекты в формы, а затем назначить им свойства, события и методы. Для обработки форм и объектов используется код VBA, который может содержать внедренные SQL-инструкции.

ВНИМАНИЕ!

Несмотря на то что инструменты быстрой разработки, подобные Access, могут помочь в создании эффективных приложений за сравнительно короткое время, обычно такие инструменты работают только на одной или нескольких платформах. Например, Access работает только в Windows. Не забывайте об этом, если хотите предусмотреть возможный перенос своих приложений на другую платформу или если вам действительно важно создавать платформо-независимые приложения.

Такие инструменты быстрой разработки, как Access, являются предвестниками окончательного объединения процессов проектирования реляционных и объектно-ориентированных баз данных. При этом основные достоинства реляционного подхода и SQL сохранятся. Они будут усилены возможностью быстрой — и сравнительно свободной от ошибок — разработки, свойственной объектно-ориентированному программированию.

Использование SQL в Microsoft Access

В основном с Microsoft Access имеют дело пользователи, стремящиеся разрабатывать относительно простые приложения без программирования "вручную". Процедурный язык VBA (Visual Basic for Applications) и SQL встроены в Access, однако в документации информация о них очень скудная. Если вы собираетесь создавать более сложные приложения на основе двух этих языков, вам следует прочитать специализированные книги по этой теме. За прошедшее десятилетие подход к программированию, реализованный в Access, не претерпел значительных изменений. Учтите, что реализация SQL в Access является неполной, а кроме того, чтобы найти информацию об SQL, вам придется освоить дедуктивный метод Шерлока Холмса.

ЗАПОМНИ!

В главе 3 мы уже говорили о трех основных компонентах SQL: языке определения данных (DDL), языке манипулирования данными (DML) и языке управления данными (DCL). В Access главным образом реализовано только одно из этих подмножеств: DML. Да, вы сможете выполнять операции по созданию таблиц с помощью диалекта Access SQL, но это проще делать с использованием графических инструментов, о которых мы говорили в главе 4. То же самое относится и к реализации функций безопасности, о которых речь шла в главе 14.

Чтобы оценить возможности SQL в Access, лучше воспользоваться готовым примером. Рассмотрим базу данных вымышленной некоммерческой исследовательской организации Oregon Lunar Society. Эта организация состоит из

нескольких независимых исследовательских групп, среди которых есть группа по изучению поверхности Луны. Нам нужно составить список научных статей, написанных членами группы. Для решения этой задачи мы воспользуемся таким инструментом, как запрос по образцу (Query By Example — QBE), и постараемся получить нужные данные. Запрос, показанный на рис. 16.1, извлекает данные из таблиц `RESEARCHTEAMS`, `AUTHORS` и `PAPERS`, используя при этом таблицы-пересечения `AUTH-RES` и `AUTH-PAP`, которые были добавлены для упрощения отношений "многие ко многим".

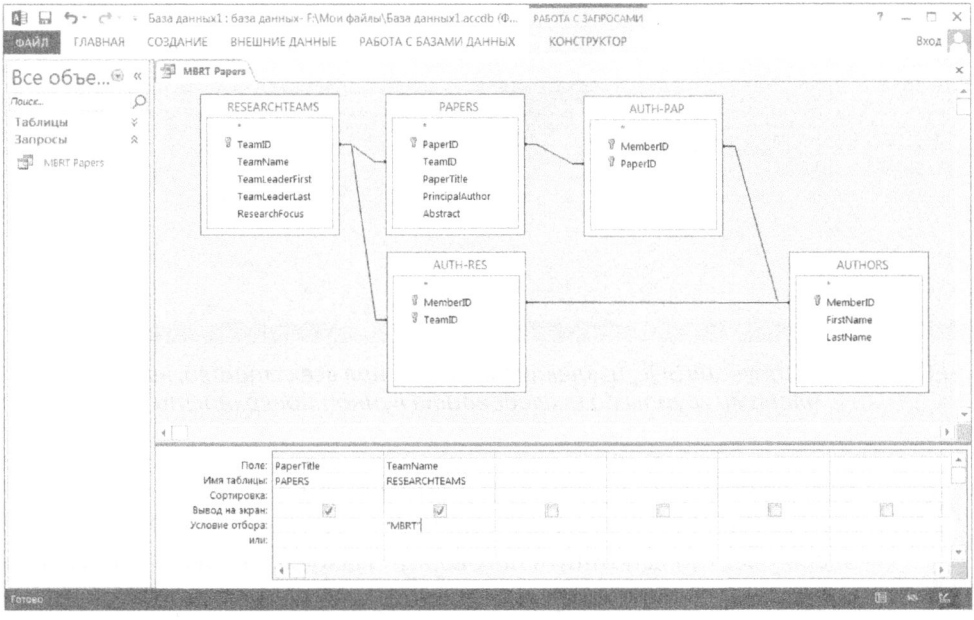


Рис. 16.1. Запрос, открытый в режиме конструктора

Чтобы получить доступ к панели инструментов, щелкните на вкладке Главная (Home), а затем — на значке Режим (View) раскрывающегося меню в левом верхнем углу окна, чтобы отобразить доступные режимы базы данных. Один из них — режим SQL, как показано на рис. 16.2.

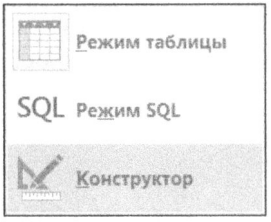


Рис. 16.2. Одним из пунктов меню является режим SQL

Если выбрать в меню режим SQL, откроется окно редактирования, в котором отобразится код SQL-инструкции, сгенерированный программой Access.

СОВЕТ

Именно код инструкции SQL (рис. 16.3) будет отправлен процессору базы данных. Этот процессор, непосредственно взаимодействующий с базой данных, понимает только инструкции SQL. Прежде чем отправлять процессору информацию, вводимую в среде QBE, ее необходимо преобразовать в SQL-код.

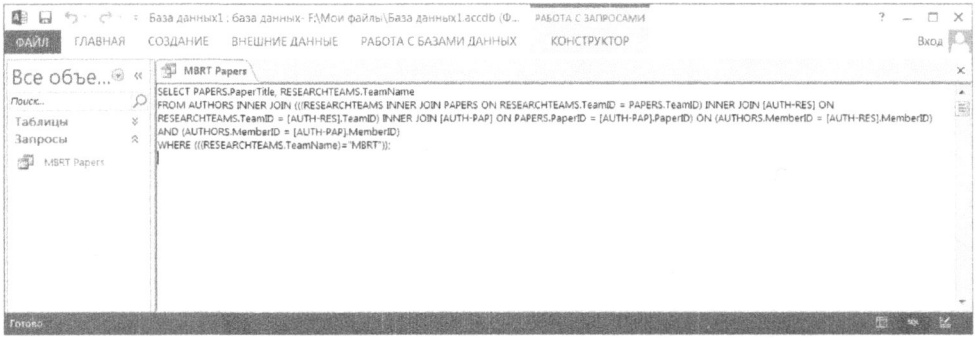


Рис. 16.3. Инструкция SQL, извлекающая названия всех статей, написанных членами группы по исследованию лунной поверхности

ВНИМАНИЕ!

Вы, наверное, заметили, что синтаксис инструкции SQL, показанный на рис. 16.3, несколько отличается от стандарта ANSI/ISO SQL. Здесь уместно вспомнить поговорку "Находясь в Риме, поступай как римлянин". Работая в среде Access, нужно использовать реализованный в этой СУБД диалект SQL. То же самое относится и к другим реализациям SQL — все они тем или иным образом отличаются от стандарта.

Если хотите создать новый запрос, используя диалект Access SQL, возьмите за основу созданный самой программой код и в окне редактирования измените нужные фрагменты инструкции `SELECT`. Для запуска написанного запроса щелкните на вкладке **Конструктор** (Design), а затем — на кнопке с восклицательным знаком в верхней части экрана. Результат запроса отобразится в режиме таблицы (Datasheet View).

Практическое использование SQL

В ЭТОЙ ЧАСТИ...

- Использование ODBC
- Использование JDBC
- Работа с XML-данными

Глава 17

Доступ к данным с помощью ODBC и JDBC

В ЭТОЙ ГЛАВЕ...

- Что такое ODBC
- Компоненты ODBC
- Использование ODBC в среде "клиент/сервер"
- Использование ODBC в Интернете
- Использование ODBC в локальной сети
- Использование JDBC

С каждым годом компьютеры все активнее и активнее подключаются к сетям, как локальным, так и глобальным. Как следствие, возникла необходимость в обеспечении совместного доступа к базам данных по сети, однако этому препятствует несовместимость системного программного обеспечения и приложений, работающих на разных компьютерах. Важными этапами на пути преодоления такой несовместимости стал стандарт SQL.

К сожалению, стандарт SQL 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 привыкли

к ним и зависят от них. Крупным организациям для совместного использования различных реализаций СУБД необходим другой подход, не требующий от производителей приводить их продукты к "наименьшему общему знаменателю". Этот подход и реализует открытый интерфейс доступа к базам данных — ODBC (Open DataBase Connectivity).

ODBC

ODBC — это стандартный интерфейс между базой данных и приложением, взаимодействующим с ней. Наличие подобного стандарта позволяет любому приложению на клиентском компьютере получать доступ к любой базе данных на сервере с помощью SQL. Единственное требование заключается в том, чтобы клиентская и серверная части поддерживали стандарт ODBC. ODBC 4.0 — текущая версия стандарта.

Приложение получает доступ к конкретной базе данных, используя специально разработанный для нее *драйвер* (в данном случае драйвер ODBC). Клиентская часть драйвера, работающая напрямую с приложением, должна строго соответствовать стандарту ODBC. Приложению все равно, какая СУБД установлена на сервере. Серверная часть драйвера адаптирована к конкретной базе данных. Благодаря такой архитектуре не только не нужно настраивать приложения на определенную СУБД — им даже не обязательно знать, какая именно СУБД используется. Драйвер скрывает различия между серверами баз данных.

Интерфейс ODBC

Интерфейс ODBC — это унифицированный набор определений, необходимых для организации взаимодействия приложения и базы данных. Каждое такое определение принимается в качестве стандарта. Интерфейс ODBC определяет следующее:

- библиотеку вызовов функций;
- стандартный синтаксис SQL;
- стандартные типы данных SQL;
- стандартный протокол соединения с базой данных;
- стандартные коды ошибок.

Вызовы функций ODBC обеспечивают подключение приложения к серверу базы данных, выполнение инструкций SQL и возврат результатов приложению.

СОВЕТ

Чтобы выполнить какое-либо действие с базой данных, в качестве аргумента функции ODBC необходимо указать соответствующую инструкцию SQL. При условии использования стандартного для ODBC синтаксиса SQL результат выполнения этой функции не будет зависеть от того, какая база данных установлена на сервере.

Компоненты ODBC

Интерфейс ODBC состоит из четырех функциональных компонентов, именуемых *уровнями ODBC*. Благодаря каждому из них достигается гибкость ODBC, позволяющая взаимодействовать любым ODBC-совместимым клиентам и серверам. Между пользователем и данными, которые он хочет получить, находятся четыре уровня интерфейса ODBC.

- **Приложение.** Это та часть интерфейса ODBC, с которой непосредственно работает пользователь. Естественно, приложения могут быть не только в ODBC-совместимых системах. Однако включение приложения в интерфейс ODBC имеет определенный смысл. Приложение должно взаимодействовать с источником данных при посредничестве ODBC и подключаться с помощью диспетчера драйверов ODBC в строгом соответствии со стандартом ODBC.
- **Диспетчер драйверов.** Это динамически подключаемая библиотека (DLL), обычно поставляемая компанией Microsoft. Она загружает необходимые драйверы для системных источников данных (возможно, нескольких) и направляет вызовы функций приложений с помощью драйверов к соответствующим источникам данных. Диспетчер драйверов непосредственно обрабатывает некоторые вызовы функций ODBC, а также перехватывает и обрабатывает определенные типы ошибок. Несмотря на то что стандарт ODBC внедрила компания Microsoft, он стал общепринятым (его приняли даже бескомпромиссные сторонники систем с открытым исходным кодом).
- **Драйвер DLL.** В связи с тем что источники данных могут различаться, причем в некоторых случаях весьма существенно, необходим способ преобразования стандартных вызовов функций ODBC в языковый код конкретного источника данных. Этим и занимается драйвер DLL. Каждый драйвер получает вызовы функций через стандартный интерфейс ODBC и переводит их в код, понятный источнику данных. Как только источник данных готов вернуть 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 качестве источника данных может выступать файл, хранящийся вне СУБД, доступ к данным которого реализован *индексно-последовательным методом* (indexed sequential access method — ISAM). Такие файлы могут быть расположены как на локальном, так и на удаленном компьютере. Для каждого вида источников данных требуется свой драйвер.

ODBC в среде "клиент/сервер"

В среде "клиент/сервер" интерфейс между клиентской и серверной частями называется API (Application Programming Interface — интерфейс прикладного программирования). API может быть как специальным, так и стандартным. *Специальным* называется интерфейс, разработанный для взаимодействия с конкретной серверной частью. Программным кодом, который формирует этот интерфейс, является драйвер, и в специальной системе он называется собственным драйвером. *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 для работы как с приложением, так и с СУБД, они передают команды и информацию достаточно быстро и с минимальными задержками.


СОВЕТ

Если система "клиент/сервер" рассчитана на доступ к конкретному источнику данных и заведомо не будет использовать другой, лучше воспользоваться собственным драйвером из поставки СУБД. С другой стороны, если системе требуется обеспечивать доступ к данным, хранящимся в различных форматах, использование ODBC API избавит разработчика от выполнения огромного объема ненужной работы.

Драйверы ODBC оптимизированы для работы с определенными источниками данных серверной части, однако все они имеют одинаковый внешний интерфейс с диспетчером драйверов. Любой драйвер, не оптимизированный для работы с конкретным клиентом, скорее всего, проиграет в быстродействии собственному драйверу, который специально разработан для данного клиента. Основным недостатком первого поколения драйверов ODBC была их низкая производительность по сравнению с собственными драйверами. Однако последние измерения показали, что производительность драйверов ODBC 4.0 вполне сравнима с производительностью собственных драйверов.

На сегодняшний день технология достигла того уровня, когда уже не нужно жертвовать производительностью ради преимуществ стандартизации.

ODBC в Интернете

Операции с базами данных в Интернете кое в чем серьезно отличаются от операций с базами данных в среде "клиент/сервер". Самое заметное отличие, с точки зрения пользователя, заключено в клиентской части системы, содержащей интерфейс пользователя. В среде "клиент/сервер" интерфейс пользователя — это часть приложения, которая связывается с источником данных на сервере посредством ODBC-совместимых инструкций SQL. В веб-среде клиентская часть системы по-прежнему находится на локальном компьютере, но взаимодействует с источником данных на сервере с помощью стандартного протокола HTTP.

Любой пользователь, располагающий соответствующим клиентским программным обеспечением (и соответствующими полномочиями), может получить доступ к данным, находящимся в Интернете. Это означает, что можно создать приложение на своем рабочем компьютере, а позже получить доступ к нему со своего мобильного устройства. На рис. 17.1 показаны основные различия между системой "клиент/сервер" и системой, созданной с помощью веб-технологий.

Серверные расширения

В системе, созданной на основе веб-технологий, клиентский компьютер и веб-сервер взаимодействуют с помощью протокола HTTP. *Серверное расширение* — это компонент системы, который преобразует команды, передаваемые по сети, 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 к виду, понятному веб-серверу. Затем данные отсылаются клиентскому компьютеру по Интернету и отображаются у пользователя. На рис. 17.2 приведена схема такого взаимодействия.

Клиентские расширения

Такие приложения, как Mircosoft Access 2016, предназначены для обработки данных, которые хранятся либо локально на компьютере пользователя, либо на сервере, расположенном в локальной или глобальной сети, либо в облачных хранилищах Интернета. Хранилище, предоставляемое компанией Microsoft,

называется OneDrive. Доступ к приложению в облаке можно также получить с помощью браузера. Браузеры создаются и оптимизируются для организации простого и интуитивно понятного интерфейса доступа к веб-серверам любых типов. Наиболее популярные браузеры, такие как Google Chrome, Mozilla Firebox, Microsoft Internet Explorer и Apple Safari, изначально не предназначались (и не оптимизировались) для использования в качестве клиентской части базы данных. Чтобы добиться требуемого уровня взаимодействия с базой данных в Интернете, необходимы дополнительные функциональные возможности. Для обеспечения этих возможностей были разработаны различные *клиентские расширения*, которые включают элементы управления ActiveX, аплеты Java и сценарии. Расширения взаимодействуют с сервером с помощью протокола HTTP, используя стандартный язык — HTML. Любой HTML-код, получающий доступ к базе данных, перед отправкой источнику данных преобразуется серверным расширением в ODBC-совместимый SQL-код.

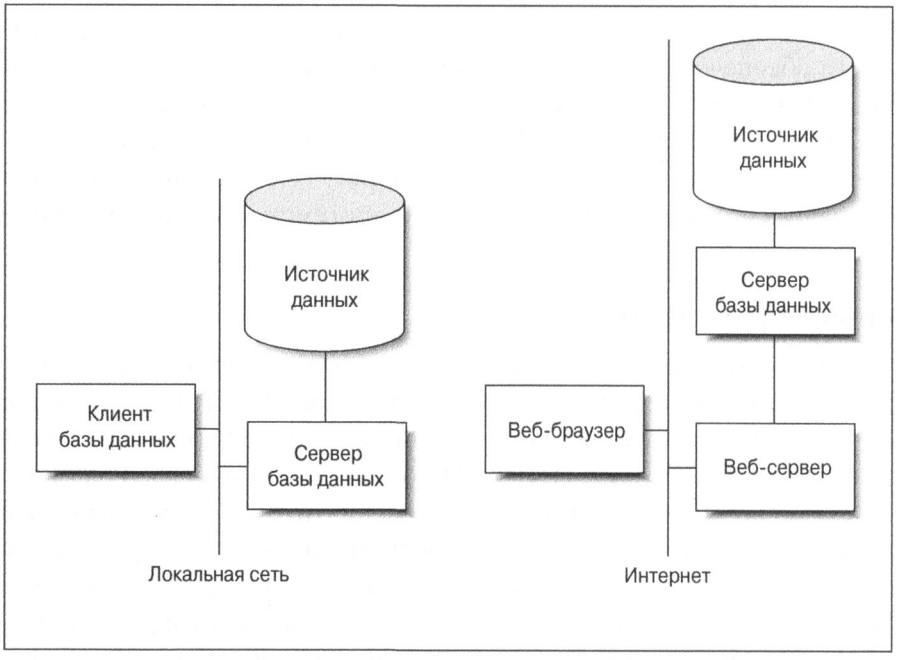


Рис. 17.1. Система "клиент/сервер" в сравнении с веб-ориентированной системой

Элементы управления ActiveX

Элементы управления Microsoft ActiveX работают с Microsoft Internet Explorer — одним из самых популярных браузеров в мире (хотя его позиции значительно пошатнулись с ростом популярности таких браузеров, как Google Chrome и Mozilla Firefox).

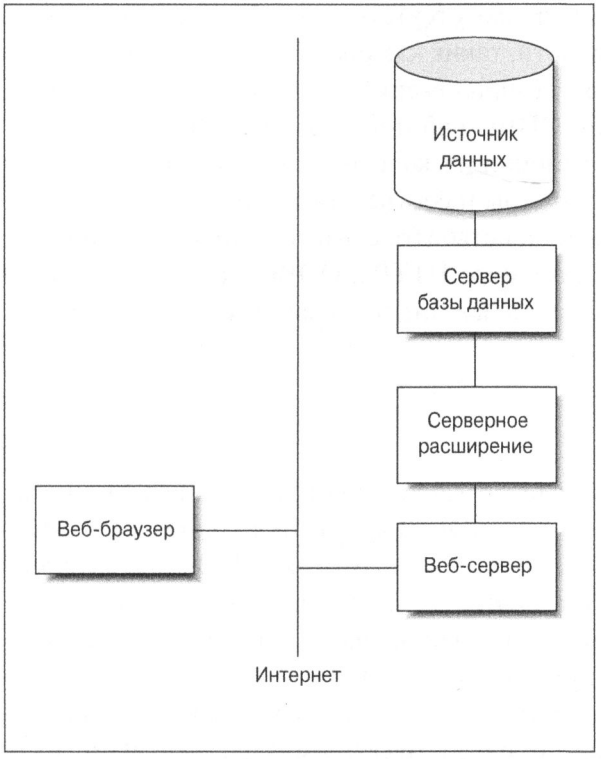


Рис. 17.2. Веб-ориентированная СУБД с серверным расширением

Сценарии

Сценарии — наиболее гибкий инструмент создания клиентских расширений. Использование языка сценариев, такого как JavaScript или Microsoft VBScript, позволяет максимально контролировать происходящее на клиентском компьютере. С помощью сценариев можно проверять достоверность данных, вводимых в поля формы, и отбраковывать неправильно заполненные формы еще на клиентском компьютере. Это сэкономит ваше время и нагрузку на интернет-канал. Безусловно, контроль данных можно также выполнять на сервере путем применения ограничений к значениям элементов данных. Как и аплеты Java, сценарии встраиваются в веб-страницу и выполняются при ее открытии пользователем.

ODBC в локальной сети

Интранет — это локальная или региональная сеть, работающая как упрощенная версия Интернета. Поскольку вся сеть принадлежит одной

организации, как правило, отсутствует необходимость в применении комплексных мер безопасности, таких как брандмауэры. Все инструменты, разработанные для создания веб-приложений, подходят также для создания приложений локальных сетей. ODBC работает в локальных сетях точно так же, как и в Интернете. При наличии нескольких различных источников данных клиенты, использующие браузеры и соответствующие клиентские и серверные расширения, могут взаимодействовать с этими источниками посредством SQL-кода, передаваемого с помощью HTTP и ODBC. Драйвер ODBC преобразует SQL-код в собственный язык команд базы данных и выполняет его.

JDBC

Интерфейс Java-приложений для взаимодействия с базами данных (Java DataBase Connectivity — JDBC) имеет много общего с ODBC, но в то же время содержит несколько существенных отличий. Одно из отличий явствует из названия. Как и ODBC, JDBC — универсальный интерфейс доступа к базам данных, не зависящий от источника данных на сервере. Различие состоит в том, что клиентское приложение для JDBC может быть написано только на Java, а не на каком-то другом языке программирования (например, C++ или Visual Basic). Еще одно отличие состоит в том, что Java и JDBC от начала и до конца разрабатывались для использования в Интернете.

Java — это язык программирования (похожий на C++), разработанный компанией Sun Microsystems специально для создания клиентских веб-ориентированных программ. После установления соединения между клиентской машиной и сервером соответствующий аплет Java загружается на компьютер клиента, где и выполняется. Аплет, внедренный на веб-страницу, предоставляет функции взаимодействия с базой данных, реализуя гибкий доступ клиента к данным на сервере. На рис. 17.3 показан механизм работы веб-приложения с базой данных с использованием аплета Java, запущенного на клиентской машине.

Аплет — это небольшое приложение, находящееся на сервере. Когда клиент подключается по Интернету к серверу, аплет загружается на клиентский компьютер и запускается. Аплеты Java разработаны таким образом, чтобы запускаться в своей *песочнице*. Песочница — это определенное (изолированное) место в памяти клиентского компьютера, где 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 конфиденциальной информации или нанести серьезный вред данным.

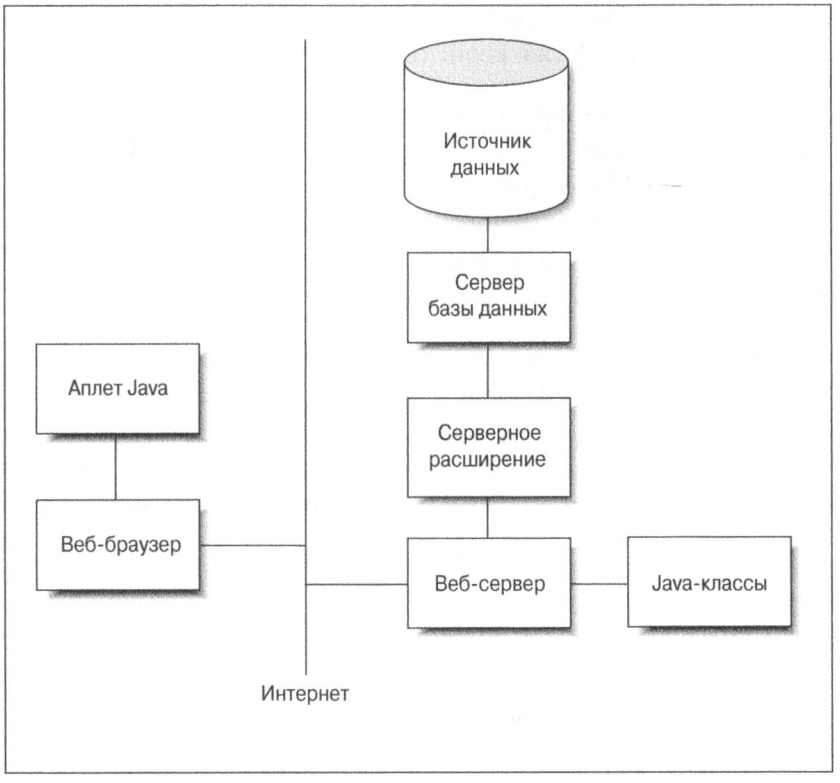


Рис. 17.3. Веб-приложение, предназначенное для работы с базой данных с использованием аплета Java

Главное достоинство аплетов Java заключается в том, что они постоянно обновляются. Поскольку аплеты загружаются с сервера при каждом использовании (а не остаются постоянно на машине клиента), клиент всегда снабжен последней версией аплета, доступной на момент его запуска.

СОВЕТ

Если вы отвечаете за поддержку сервера своей организации, то вам никогда не придется волноваться о совместимости с любым из клиентов при обновлении программного обеспечения сервера. Просто убедитесь в том, что загружаемый аплет Java совместим с новой конфигурацией сервера, и если все веб-браузеры поддерживают аплеты Java, то они автоматически также станут совместимыми с сервером. Java — это полнофункциональный язык программирования, позволяющий создавать надежные приложения доступа к базам данных в любых конфигурациях "клиент/сервер". При таком использовании доступ Java-приложений к базам данных посредством JDBC подобен доступу к данным приложений C++ посредством ODBC. В то же

время при использовании в Интернете (или в локальной сети) работа приложений Java в корне отличается от работы приложений C++.

Когда система функционирует в Интернете, условия ее работы отличаются от условий в среде "клиент/сервер". Клиентская часть приложения, которая работает с Интернетом, — это браузер с минимальными вычислительными возможностями. Эти возможности должны быть расширены, чтобы переложить на клиента часть работы с базой данных, и такое расширение функций обеспечивают аплеты Java.


ВНИМАНИЕ!

Загрузка данных с неизвестного сервера связана с рядом потенциальных опасностей. Если же вы загружаете Java-аплет, то уровень опасности резко снижается, хотя и не до нуля. Стоит с осторожностью относиться к сомнительному серверу и не позволять исполняемому коду беспрепятственно "заходить" на ваш компьютер.

Как и ODBC, JDBC передает SQL-инструкции 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 назад приложению. Выгода от использования 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 корректного двухстороннего взаимодействия. Несмотря на то что интерфейс JDBC предназначен для работы в веб-среде, он может также работать в среде "клиент/сервер", обеспечивая посредничество для взаимодействия приложения, написанного на языке Java, с серверной частью базы данных.

Глава 18

Работа с XML-данными

В ЭТОЙ ГЛАВЕ...

- Совместное использование SQL и XML
- XML, базы данных и Интернет

Одной из новых функциональных возможностей, заявленных стандартом ISO/IEC SQL:2008, стала поддержка файлов XML (eXtensible Markup Language — расширяемый язык разметки), которые все в большей мере 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" для обмена данными между пользователями.

Связь между XML и 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, в отличие от упомянутых языков, он оснащен средствами, которые позволяют ему быть в курсе того, какие данные он "транспортирует". Если HTML занимается только форматированием текста и графики в документе, то XML структурирует содержимое документа. Однако сам XML не предназначен для форматирования. Для решения этой задачи XML необходимо дополнить *таблицей стилей*, которая, как и в HTML, позволяет форматировать XML-документы.

Структура XML-документа обусловлена его XML-схемой, которая может служить примером *метаданных* (т.е. данных, которые описывают другие данные). Схема XML описывает, где и в каком порядке элементы могут встречаться в документе. Она может также описать тип данных элемента и наложить ограничения на его значения.

SQL и XML структурируют данные двумя различными способами.

- SQL является прекрасным инструментом для работы с числовыми и текстовыми данными, классифицированными по типам и имеющими точно описанный размер.

  SQL был создан как стандартный инструмент для обработки информации, хранящейся в реляционных базах данных.

- Если речь идет о произвольных данных, которые не так-то легко классифицировать, то лучше использовать XML.

  Движущей силой для разработчиков этого языка было стремление реализовать универсальный стандарт для передачи данных между разнородными компьютерами и отображения данных на веб-сайтах.

SQL и XML дополняют возможности друг друга. Каждый язык имеет определенные преимущества, позволяя пользователю получить всю необходимую информацию в любое время и в любом месте.

Тип данных XML

Тип данных `XML` был включен в стандарт SQL:2003. Это означает, что СУБД, совместимые со стандартом, могут хранить данные в формате XML и напрямую работать с ними без их предварительного преобразования из какого-либо типа данных SQL в тип `XML`.

Несмотря на то что тип `XML` (включая его подтипы) встроен в любую совместимую СУБД, он работает как тип данных, определяемый пользователем (UDT). Его подтипами являются:

- `XML(DOCUMENT(UNTYPED));`
- `XML(DOCUMENT(ANY));`
- `XML(DOCUMENT(XMLSCHEMA));`
- `XML(CONTENT(UNTYPED));`
- `XML(CONTENT(ANY));`
- `XML(CONTENT(XMLSCHEMA));`
- `XML(SEQUENCE).`

Тип данных XML обеспечивает непосредственное взаимодействие SQL и XML, поскольку позволяет приложениям выполнять SQL-операции над XML-содержимым и, наоборот, XML-операции над SQL-содержимым. Вы сможете использовать столбцы с типом данных XML совместно со столбцами, содержащими любые предопределенные типы данных SQL (см. главу 2), объединяя их в предложениях WHERE запросов к базе данных. Ваша СУБД, следуя истинным традициям реляционных баз данных, должна самостоятельно определить оптимальный вариант выполнения запроса.

Когда использовать тип данных XML

Необходимость хранения данных в формате XML зависит от того, что вы планируете с ними делать. Принятие такого решения целесообразно в следующих случаях.

- Когда нужно хранить целый блок данных для последующего их извлечения тем же целым блоком.
- Когда необходимо выполнить запрос целого XML-документа, для чего некоторые СУБД предлагают расширенные возможности инструкции EXTRACT, которые позволят извлечь нужное содержимое из XML-документа.
- Когда необходим строгий контроль типов данных (строгая типизация) в инструкциях SQL. Тип данных XML позволяет гарантировать использование достоверных XML-значений, а не просто произвольных текстовых строк.
- Для обеспечения совместимости с будущими, еще не установленными системами хранения, которые могут не поддерживать существующие типы, например CHARACTER LARGE OBJECT (CLOB; см. главу 2).
- Для получения преимуществ в будущих оптимизациях, которые будут поддерживать только данные типа XML.

Вот пример использования типа данных XML.

```
CREATE TABLE CLIENT (
   ClientName      CHAR (30)      NOT NULL,
   Address1        CHAR (30),
   Address2        CHAR (30),
   City            CHAR (25),
   State           CHAR (2),
   PostalCode      CHAR (10),
   Phone           CHAR (13),
   Fax             CHAR (13),
   ContactPerson   CHAR (30),
   Comments        XML(SEQUENCE)
) ;
```

Эта инструкция SQL позволяет сохранить XML-документ в столбце Comments таблицы CLIENT. Результирующий документ будет выглядеть примерно так.

```
<Comments>
   <Comment>
      <CommentNo>1</CommentNo>
      <MessageText>Клиника VetLab оснащена оборудованием для анализа
         крови у пингвинов?</MessageText>
      <ResponseRequested>Да</ResponseRequested>
   </Comment>
   <Comment>
      <CommentNo>2</CommentNo>
      <MessageText>Спасибо за быстрые результаты анализов
         морского леопарда!</MessageText>
      <ResponseRequested>Нет</ResponseRequested>
   </Comment>
</Comments>
```

Когда не стоит использовать тип данных XML

Возможность использования типа данных XML не означает необходимость. Иногда в этом нет никакого смысла, поскольку бо́льшая часть данных в реляционных базах гораздо лучше обрабатывается в их текущем формате, нежели в формате XML. Приведем два примера, когда тип XML не используется:

- при естественном разбиении данных в реляционной структуре на таблицы, строки и столбцы;
- если необходимо обновлять не весь документ, а только некоторые его части.

Преобразование данных из формата SQL в формат XML и обратно

Для обмена данными между реляционными базами и документами XML различные элементы базы данных должны быть преобразованы в эквивалентные элементы документа XML и наоборот. В следующих разделах вы узнаете, какие элементы следует преобразовывать.

Преобразование наборов символов

В SQL поддержка наборов символов зависит от конкретной реализации. Это означает, что СУБД DB2 компании IBM может поддерживать наборы

символов, которые не поддерживаются СУБД SQL Server компании Microsoft. SQL Server, в свою очередь, поддерживает наборы символов, которые не поддерживаются в Oracle. Несмотря на то что самые распространенные наборы символов поддерживаются практически всеми СУБД, использование менее популярного набора символов может усложнить перевод базы данных и приложения с одной платформы на другую.

У XML нет никаких проблем совместимости с наборами символов, поскольку он поддерживает только один набор: Unicode. С точки зрения обмена данными между любыми реализациями SQL и XML это очень хорошо. Поставщики реляционных СУБД должны описывать соответствие строк каждого поддерживаемого ими символьного набора символам Unicode и наоборот. К счастью, XML не поддерживает множественные символьные наборы, что освобождает поставщиков от проблем отношений "многие ко многим", связанных с бесчисленными преобразованиями символов в разных направлениях.

Преобразование идентификаторов

В отличие от SQL, XML более строго подходит к определению состава символов, допустимых в идентификаторах. Прежде чем стать частью документа XML, символы, допустимые в SQL и недопустимые в XML, должны быть соответствующим образом преобразованы. SQL поддерживает идентификаторы с разделителями. Это означает, что все виды дополнительных символов (например, `%`, `$` и `&`) считаются допустимыми, если они заключены в двойные кавычки. Но такие символы недопустимы в XML. Кроме того, имена в XML, начинающиеся с символов `XML` (в любых комбинациях), уже зарезервированы и, таким образом, не могут использоваться. Именно поэтому идентификаторы SQL, начинающиеся с указанных символов, должны быть изменены.

При согласованном переходе от SQL к XML все идентификаторы преобразуются в Unicode. Любые SQL-идентификаторы, которые являются также допустимыми именами XML, остаются неизменными. Символы идентификатора SQL, недопустимые для имен XML, заменяются шестнадцатеричным кодом. Полученный результат имеет вид "_x*NNNN*_" или "_x*NNNNNNNN*_", где *N* — шестнадцатеричная цифра в верхнем регистре. Например, символ подчеркивания "_" будет представлен как "_x005F_", а двоеточие — как "_x003A_". Эти представления включают в себя коды, используемые для указанных символов в таблице Unicode. В случае, если SQL-идентификатор начинается с символов *x*, *m* и *l*, перед таким сочетанием будет поставлен префикс вида "_xFFFF_".

Преобразовать символы из формата XML в формат SQL гораздо проще. Все, что для этого необходимо сделать, — найти в XML-имени последовательности вида "_x*NNNN*_" или "_x*NNNNNNNN*_". Всякий раз, когда вы находите такую

последовательность, заменяйте ее соответствующими символами из таблицы Unicode. Если же XML-имя начинается с символов "_XFFFF_", то игнорируйте их.


ЗАПОМНИ!

Следуя этим простым правилам, можно преобразовать SQL-идентификатор в XML-имя, а затем снова вернуться к SQL-идентификатору. Однако это не действует для преобразования XML-имени в SQL-идентификатор и обратно.

Преобразование типов данных

В стандарте SQL определено, что типы данных SQL преобразуются в наиболее близкий тип данных в схеме XML. *Наиболее близкий* означает, что все значения, допустимые для типа данных SQL, должны быть допустимы и для типа данных в схеме XML, а наименее возможные значения, не допустимые для типа SQL, будут допустимы для типа данных в схеме XML. *Фасеты XML* (набор ограничений на множество возможных значений), такие как `maxInclusive` (фасет максимума, включающий границу) и `minInclusive` (фасет минимума, включающий границу), могут ограничивать значения, допускаемые типом схемы XML, значениями, допустимыми в соответствующем типе SQL. Например, если тип данных SQL ограничивает значения типа `INTEGER` диапазоном от `-2157483648` до `2157483647`, то в XML значение `minInclusive` может быть установлено равным числу `-2157483648`. Ниже приведен пример такого преобразования.

```
<xsd:simpleType>
   <xsd:restriction base="xsd:integer"/>
      <xsd:maxInclusive value="2157483647"/>
      <xsd:minInclusive value="-2157483648"/>
      <xsd:annotation>
         <sqlxml:sqltype name="INTEGER"/>
      </xsd:annotation>
   </xsd:restriction>
</xsd:simpleType>
```


СОВЕТ

В аннотации хранится информация из определения SQL-типа, которая не используется в XML. Она может пригодиться впоследствии, при обратном преобразовании XML-документа в SQL-код.

Преобразование таблиц

Одну таблицу можно преобразовать в отдельный XML-документ. Аналогично можно преобразовать все таблицы в схеме или все таблицы в каталоге.

В процессе преобразования все привилегии сохраняются. Пользователь, имеющий полномочия на выполнение запросов SELECT с доступом только к нескольким столбцам таблицы, сможет преобразовать в XML-документ только эти столбцы. В действительности преобразование порождает два документа: один содержит данные из таблицы, другой — схему XML, описывающую первый документ. Ниже приведен пример преобразования таблицы SQL в документ, содержащий данные XML.

```
<CUSTOMER>
   <row>
      <FirstName>Abe</FirstName>
      <LastName>Abelson</LastName>
      <City>Springfield</City>
      <AreaCode>714</AreaCode>
      <Telephone>555-1111</Telephone>
   </row>
   <row>
      <FirstName>Bill</FirstName>
      <LastName>Bailey</LastName>
      <City>Decatur</City>
      <AreaCode>714</AreaCode>
      <Telephone>555-2222</Telephone>
   </row>
...
</CUSTOMER>
```

Корневому элементу присваивается имя таблицы (CUSTOMER). Каждая строка таблицы содержится в отдельном элементе <row> (в данном примере таких строк две). Кроме того, каждый элемент <row> содержит последовательность столбцевых элементов с именами соответствующих столбцов в исходной таблице (FirstName, LastName, City, AreaCode, Telephone). Каждый элемент столбца содержит соответствующее значение данных.

Обработка пустых значений

Поскольку данные в SQL могут включать пустые значения, необходимо решить, как представлять их в XML-документе. Пустое значение может быть представлено либо значением nil, либо отсутствием всякого значения. При выборе первого варианта элементы столбцов, которые содержат пустые значения, будут отмечены атрибутом xsi:nil="true".

```
<row>
   <FirstName>Bill</FirstName>
   <LastName>Bailey</LastName>
   <City xsi:nil="true" />
```

```
   <AreaCode>714</AreaCode>
   <Telephone>555-2222</Telephone>
</row>
```

Если вы выбрали вариант отсутствия элемента, то реализация преобразования SQL-данных в XML-документ будет выглядеть так.

```
<row>
   <FirstName>Bill</FirstName>
   <LastName>Bailey</LastName>
   <AreaCode>714</AreaCode>
   <Telephone>555-2222</Telephone>
</row>
```

При выборе этого варианта поле, содержащее пустое значение, просто отсутствует и нигде не упоминается.

Создание схемы XML

При преобразовании данных из SQL в XML один из создаваемых документов содержит данные, другой — информацию о схеме. В качестве примера рассмотрим схему для документа CUSTOMER, приведенного в разделе "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 вариант с атрибутом `nil`. Вариант с отсутствием элемента требует несколько иного определения элементов.

```
<xsd:element name="City" type="CHAR_25" minOccurs="0"/>
```

Функции SQL для работы с XML-данными

В стандарте SQL определен ряд операторов, функций и псевдофункций, которые после применения к данным XML выдают результат в формате SQL и наоборот. К таким функциям относятся `XMLELEMENT`, `XMLFOREST`, `XMLCONCAT` и `XMLAGG` (они описаны далее), а также ряд других, которые часто используются при публикации данных на сайтах. Часть этих функций основана на инструментах XQuery — стандартного языка запросов, разработанного специально для XML-данных. Рассмотрение этого языка — довольно обширная тема, которая выходит за рамки данной книги.

XMLDOCUMENT

Функция `XMLDOCUMENT` получает одно XML-значение, а возвращает другое. Новое XML-значение — это узел документа, создаваемый согласно правилам конструктора документа в XQuery.

XMLELEMENT

Функция `XMLELEMENT` преобразует реляционное значение в элемент XML. Она может использоваться в инструкции `SELECT` для преобразования SQL-содержимого базы данных в формат XML и его публикации на сайте. Рассмотрим следующий пример.

```
SELECT c.LastName
       XMLELEMENT ( NAME"City", c.City ) AS "Result"
  FROM CUSTOMER c
 WHERE LastName="Abelson" ;
```

Эта инструкция вернет такой результат.

LastName	***Result***
Abelson	<City>Springfield</City>

XMLFOREST

На основе списка реляционных значений функция XMLFOREST создает лес (совокупность деревьев) XML-элементов. Каждый аргумент этой функции создает новый элемент. Рассмотрим пример.

```
SELECT c.LastName
   XMLFOREST (c.City,
              c.AreaCode,
              c.Telephone ) AS "Result"
  FROM CUSTOMER c
 WHERE LastName="Abelson" OR LastName="Bailey" ;
```

При выполнении этого кода будет получен следующий результат.

LastName	***Result***
Abelson	<City>Springfield</City>
	<AreaCode>714</AreaCode>
	<Telephone>555-1111</Telephone>
Bailey	<City>Decatur</City>
	<AreaCode>714</AreaCode>
	<Telephone>555-2222</Telephone>

XMLCONCAT

Функция XMLCONCAT реализует альтернативный способ создания леса элементов путем конкатенации ее XML-аргументов. В качестве примера рассмотрим следующий код.

```
SELECT c.LastName,
   XMLCONCAT(
      XMLELEMENT ( NAME"first", c.FirstName,
      XMLELEMENT ( NAME"last", c.LastName)
   ) AS "Result"
  FROM CUSTOMER c ;
```

В результате получим следующее.

LastName	***Result***
Abelson	<first>Abe</first>

```
                          <last>Abelson</last>
Bailey                    <first>Bill</first>
                          <last>Bailey</last>
```

XMLAGG

XMLAGG — это итоговая функция, которая получает несколько XML-документов или их фрагментов и создает один XML-документ. Результат выполнения функции содержит лес элементов. Рассмотрим пример.

```
SELECT XMLELEMENT
   ( NAME"City",
      XMLATTRIBUTES ( c.City AS "name" ) ,
      XMLAGG ( XMLELEMENT ( NAME"last" c.LastName ) )
   ) AS "CityList"
  FROM CUSTOMER c
 GROUP BY City ;
```

При обработке таблицы CUSTOMER этот запрос выдаст следующий результат.

```
CityList
<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>
```

XMLCOMMENT

Функция XMLCOMMENT позволяет приложениям создавать XML-комментарии. Вот как выглядит ее синтаксис.

```
XMLCOMMENT ('Содержимое комментария'
   [RETURNING
      { CONTENT | SEQUENCE } ] )
```

Приведем пример.

```
XMLCOMMENT ('Резервное копирование базы данных каждую ночь в 2 часа.')
```

В результате будет создан XML-комментарий следующего вида.

```
<!--Резервное базы данных каждую ночь в 2 часа.-->
```

XMLPARSE

Функция XMLPARSE создает XML-значение, выполняя грамматический анализ строки. Его можно использовать следующим образом.

```
XMLPARSE (DOCUMENT '     GREAT JOB! '
          PRESERVE WHITESPACE )
```

Приведенный в качестве примера код создает значение XML с типом данных XML(DOCUMENT(UNTYPED)) либо XML(DOCUMENT(ANY)). Какой из этих двух подтипов будет использован, зависит от конкретной реализации.

XMLPI

Эта функция позволяет приложениям создавать инструкции (правила) обработки XML-кода. Она имеет следующий синтаксис.

```
XMLPI NAME объект
     [ , строковое_выражение]
     [RETURNING
     { CONTENT | SEQUENCE } ] )
```

Элемент *идентификатор* представляет идентификатор объекта инструкции обработки; элемент *строковое_выражение* — содержимое инструкции обработки. Эта функция создает XML-комментарий следующего формата:

```
<? объект строковое_выражение ?>
```

XMLQUERY

Эта функция вычисляет выражение XQuery и возвращает результат приложению SQL. Она имеет следующий синтаксис.

```
XMLQUERY (выражение_XQuery
   [ PASSING { By REF | BY VALUE }
         список_аргументов]
     RETURNING { CONTENT | SEQUENCE }
{ BY REF | BY VALUE } )
```

А вот пример использования функции XMLQUERY.

```
SELECT max_average,
   XMLQUERY (
         'for $batting_average in
              /player/batting_average
          where /player/lastname = $var1
          return $batting_average'
         PASSING BY VALUE
            'Mantle' AS var1,
         RETURNING SEQUENCE BY VALUE )
  FROM offensive_stats
```

XMLCAST

Функция XMLCAST аналогична обычной SQL-функции CAST, но имеет некоторые ограничения. Она позволяет приложениям преобразовывать значения из одного XML-типа в другой, а также в один из типов SQL. Аналогичным образом ее можно использовать и для обратных преобразований, т.е. из SQL-типа в XML-тип. Ниже приведены налагаемые ограничения.

- По крайней мере один из используемых типов (источника либо приемника) должен иметь тип XML.
- Ни один из используемых типов не должен иметь тип коллекции SQL, записи, структуры или ссылки.
- Только значение одного из типов XML или пустое SQL-значение может быть преобразовано в тип XML(DOCUMENT(UNTYPED)) или XML(DOCUMENT(ANY)).

Приведем пример:

```
XMLCAST ( CLIENT.ClientName AS XML(CONTENT(UNTYPED))
```

Функция XMLCAST может быть преобразована в обычную SQL-функцию CAST. Единственной причиной использования отдельного ключевого слова является применение вышеназванных ограничений.

Предикаты

Предикат возвращает значение TRUE или FALSE. В SQL имеются предикаты, предназначенные для работы с XML.

DOCUMENT

Предикат DOCUMENT определяет, является ли заданное XML-значение документом XML. Он проверяет значение на принадлежность к типу XML(DOCUMENT(UNTYPED)) или XML(DOCUMENT(ANY)). Его синтаксис таков:

```
XML-значение IS [NOT] [ANY | UNTYPED] DOCUMENT
```

Если вычисление XML-выражения дает истинный результат, то предикат возвращает значение TRUE; в противном случае — FALSE. Если результатом является пустое значение, предикат возвращает значение UNKNOWN. Если в предикате не указано ключевое слово ANY или UNTYPED, по умолчанию подразумевается ANY.

CONTENT

Этот предикат определяет, является ли XML-значение экземпляром типа XML(ANY CONTENT) или XML(UNTYPED CONTENT). Он имеет следующий синтаксис:

```
XML-значение IS [NOT] [ANY | UNTYPED] CONTENT
```

Если в предикате не указано ключевое слово ANY или UNTYPED, то по умолчанию подразумевается ANY.

XMLEXISTS

Предикат XMLEXISTS проверяет наличие заданного значения. Его синтаксис таков:

```
XMLEXISTS (выражение_XQuery [список_аргументов ])
```

Предикат вычисляет выражение XQuery, используя значения, заданные в списке аргументов. Если значение, возвращаемое выражением, оказывается пустым (NULL), предикат возвращает значение UNKNOWN. Если результатом выражения является пустая последовательность XQuery, то результатом предиката является значение FALSE, в противном случае — TRUE. Этот предикат можно использовать для определения того, включает ли XML-документ конкретное содержимое, прежде чем использовать его фрагмент в каком-либо выражении.

VALID

Этот предикат служит для проверки соответствия некоторого значения зарегистрированной схеме XML. Синтаксис этого предиката сложнее, чем у других предикатов.

```
XML-значение IS [NOT] VALID
   [XML valid параметр_соответствия_ограничениям]
   [XML valid according-to XML-схема]
```

Этот предикат проверяет, принадлежит ли заданное XML-значение к одному из пяти предопределенных подтипов XML: XML(SEQUENCE), XML(CONTENT (ANY)), XML(CONTENT(UNTYPED)), XML(DOCUMENT(ANY)) или XML(DOCUMENT (UNTYPED)). Дополнительно он может проверить, зависит ли достоверность XML-значения от ограничений и соответствует ли это XML-значение заданной схеме XML.


ЗАПОМНИ!

Существуют четыре возможных значения аргумента *параметр_соответствия_ограничениям* в синтаксисе предиката.

- WITHOUT IDENTITY CONSTRAINTS. Если компонент *параметр_соответствия_ограничениям* не задан явно, то предполагается именно это его значение. Если используется параметр DOCUMENT, то он действует подобно объединению предиката DOCUMENT и предиката VALID с параметром WITH IDENTITY CONSTRAINTS GLOBAL.
- WITH IDENTITY CONSTRAINTS GLOBAL. Этот компонент синтаксиса предполагает проверку соответствия значения не только схеме XML, но и правилам XML для отношений ID/IDREF. ID и IDREF представляют собой типы атрибутов XML, идентифицирующие элементы документа.
- WITH IDENTITY CONSTRAINTS LOCAL. Этот компонент синтаксиса предполагает проверку соответствия значения только схеме XML, но не правилам XML для отношений ID/IDREF или правилам схемы XML для ограничений идентичности.
- DOCUMENT. Этот компонент синтаксиса означает, что значение XML-выражения является документом и синтаксис WITH IDENTITY CONSTRAINTS GLOBAL корректен с использованием предложения XML valid according-to. Предложение XML valid according-to идентифицирует схему, на соответствие которой и проверяется XML-значение.

Преобразование данных XML в таблицы SQL

До недавнего времени проблема отношений между SQL и XML сводилась к необходимости преобразования содержимого таблиц SQL в формат XML, чтобы обеспечить их доступность в Интернете. Стандарт SQL:2008 предложил решение обратной задачи — преобразования данных XML в таблицы SQL, чтобы к ним можно было обращаться с помощью запросов, созданных с использованием стандартных инструкций SQL. Для выполнения этой операции предназначена псевдофункция XMLTABLE, имеющая следующий синтаксис.

```
XMLTABLE ( [объявление_пространства_имен,]
выражение_XQuery
[PASSING список_аргументов]
COLUMNS определения_столбцов_таблицы_XML )
```

В этой конструкции *список_аргументов* имеет следующий вид.

```
значение_выражения AS идентификатор
```

А *определения_столбцов_таблицы_XML* представляют собой список разделенных запятыми элементов такого вида:

```
имя_столбца FOR ORDINALITY
```

или такого:

```
имя_столбца тип_данных
[BY REF | BY VALUE]
[умолчания]
[PATH выражение_XQuery]
```

В качестве иллюстрации приведем пример использования псевдофункции XMLTABLE для извлечения данных из документа XML в псевдотаблицу SQL. Псевдотаблицы не существуют постоянно, однако в остальном они идентичны обычным таблицам SQL. Если хотите сделать псевдотаблицу постоянной, сначала создайте таблицу с помощью инструкции CREATE TABLE, а затем вставьте в нее XML-данные.

```
SELECT clientphone.*
  FROM
   clients_xml ,
      XMLTABLE(
         'for $m in
            $col/client
         return
            $m'
      PASSING clients_xml.client AS "col"
      COLUMNS
         "ClientName" CHARACTER (30) PATH 'ClientName' ,
         "Phone" CHARACTER (13) PATH 'phone'
      ) AS clientphone
```

При выполнении этой инструкции получим следующий результат.

```
ClientName                     Phone
------------------------------ -------------
Abe Abelson                    (714)555-1111
Bill Bailey                    (714)555-2222
Chuck Wood                     (714)555-3333

(3 rows in clientphone)
```

Преобразование нестандартных типов данных в XML

В SQL к нестандартным типам данных относятся домены, индивидуальные типы UDT (типы данных, определяемые пользователем), записи, массивы и мультимножества. Для преобразования каждого из этих типов данных в формат XML используется соответствующий XML-код. Примеры преобразования таких типов данных будут рассмотрены далее.

Домены

Для того чтобы преобразовать домен SQL в формат XML, вначале необходимо создать этот домен. Для создания домена воспользуемся инструкцией CREATE DOMAIN.

```
CREATE DOMAIN WestCoast AS CHAR (2)
 CHECK (State IN ('CA', 'OR', 'WA', 'AK')) ;
```

Теперь создадим таблицу, которая использует этот домен.

```
CREATE TABLE WestRegion (
   ClientName    Character (20)    NOT NULL,
   State         WestCoast         NOT NULL
) ;
```

Ниже приведена схема XML для преобразования домена в XML.

```
<xsd:simpleType>
   Name='DOMAIN.Sales.WestCoast'>

   <xsd:annotation>
      <xsd:appinfo>
         <sqlxml:sqltype kind='DOMAIN'
            schemaName='Sales'
            typeName='WestCoast'
            mappedType='CHAR_2'
            final='true'/>
      <xsd:appinfo>
   </xsd:annotation>

   <xsd:restriction base='CHAR_2'/>

</xsd:simpleType>
```

После применения этого преобразования получим XML-документ примерно следующего вида.

```
<WestRegion>
<row>
   .
   .
   <State>AK</State>
   .
   .
   </row>
   .
   .
</WestRegion>
```

Индивидуальные типы UDT

С индивидуальными типами UDT можно делать то же, 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 тип XML используется следующая схема XML.

```
<xsd:simpleType>
   Name='UDT.Sales.WestCoast'>

   <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, который совпадает с элементом, созданным для описанного выше домена.

Записи

Тип ROW позволяет поместить несколько элементов или даже целую запись в отдельное поле строки таблицы. Тип ROW создается как часть описания таблицы.

```
CREATE TABLE CONTACTINFO (
   Name     CHARACTER (30)
   Phone    ROW (Home CHAR (13), Work CHAR (13))
) ;
```

Теперь воспользуемся следующей схемой для преобразования этого типа в формат XML.

```
<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-код.

```
<Phone>
   <Home>(888)555-1111</Home>
   <Work>(888)555-1212</Work>
</Phone>
```

Массивы

Если хотите поместить в одно поле несколько элементов одного и того же типа, вместо типа `ROW` воспользуйтесь типом `Array`. В качестве примера объявим в таблице `CONTACTINFO` столбец `Phone` как массив, а затем создадим схему XML, которая преобразует массив в XML-код.

```
CREATE TABLE CONTACTINFO (
   Name     CHARACTER (30),
   Phone    CHARACTER (13) ARRAY [4]
) ;
```

Теперь воспользуемся следующей схемой для преобразования этого типа в формат XML.

```
<xsd:complexType Name='ARRAY_4.CHAR_13'>

   <xsd:annotation>
      <xsd:appinfo>
         <sqlxml:sqltype kind='ARRAY'
            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 таким.

```
<Phone>
   <element>(888)555-1111</element>
   <element>xsi:nil='true'/>
   <element>(888)555-3434</element>
</Phone>
```

Элемент в массиве вида `xsi:nil='true'` означает, что второй телефонный номер в исходной таблице содержит пустое значение.

Мультимножества

Номера телефонов из предыдущего примера могут храниться не только в массиве, но и в мультимножестве. Для преобразования мультимножества воспользуемся следующим запросом.

```
CREATE TABLE CONTACTINFO (
   Name    CHARACTER (30),
   Phone   CHARACTER (13) MULTISET
) ;
```

Преобразуем этот тип в формат 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 таким.

```
<Phone>
   <element>(888)555-1111</element>
   <element>xsi:nil='true'/>
   <element>(888)555-3434</element>
</Phone>
```

Содружество SQL и XML

SQL реализует стандартный способ хранения данных в структурированном виде. Удобная структура хранения позволяет пользователям обслуживать хранилища данных практически любого объема и эффективно извлекать из них нужную информацию. XML превратился из стандарта де-факто в официальный стандарт передачи данных по Интернету между несовместимыми системами. При объединении этих двух языков ценность каждого из них существенно возрастает. Теперь SQL может работать с данными, которые не вписываются в строгую реляционную парадигму, когда-то сформулированную доктором Коддом. А средства XML могут эффективно извлекать информацию из реляционных баз данных и отсылать их обратно. В результате информация становится все более и более доступной, причем более простыми методами. В конце концов, именно для этой цели возникло такое тесное содружество SQL и XML.

Глава 19

SQL и JSON

В ЭТОЙ ГЛАВЕ...

- Переход от JSON к SQL
- Элементы модели данных SQL/JSON
- Преобразование данных с помощью функций SQL/JSON
- Проверка строки на корректность формата JSON

На заре компьютерных вычислений базы данных напрочь отсутствовали. Данные хранились в виде простых файлов, лишенных всякой организационной структуры. Понятно, что это не идеальный вариант, поэтому в 1950- и 1960-х годах были разработаны архитектуры баз данных, прежде всего иерархическая и сетевая. Эти ранние архитектуры были в значительной степени вытеснены реляционной архитектурой, которая стала доминирующей в 1980-е годы и в последующие десятилетия. Реляционные базы данных основаны на реляционной архитектуре, а SQL является универсальным языком, предназначенным для работы с реляционными базами данных.

В последние годы, главным образом благодаря появлению концепции больших данных (Big Data), в разных прикладных областях начали внедряться нереляционные архитектуры. Совместно их называют архитектурами баз данных NoSQL. Каждая из них имеет собственный способ организации и хранения данных.

Стало ясно, что можно многого добиться, если данные, хранящиеся в одной базе NoSQL, отправлять и использовать в другой базе NoSQL. Чтобы обойти проблему разных форматов файлов в разных базах данных NoSQL, был разработан формат обмена данными, который все базы данных NoSQL могли бы использовать для обмена данными друг с другом. Этот формат называется JSON,

что является аббревиатурой от *JavaScript Object Notation* (объектная нотация JavaScript).

Несмотря на то что JSON является производным от JavaScript, его можно использовать с любым языком программирования, который его поддерживает. Часть "Object" в имени относится к тому факту, что данные могут передаваться в виде объектов JSON. Однако не стоит зацикливаться на этом. Данные могут быть переданы в виде массивов, чисел и строк, а также значений `true`, `false` и `null`.

Совместное использование JSON и SQL

В наши дни, когда базы данных NoSQL становятся все более и более популярными, ценность обмена данными с реляционными СУБД стала очевидной. В реляционных базах данных хранятся данные за десятилетия, и они могут быть полезны для приложений, предназначенных для работы с базами данных NoSQL. Верно и обратное утверждение. Огромное количество данных хранится в базах NoSQL, которые могут использоваться приложениями, предназначенными для работы с реляционными базами данных. Чтобы удовлетворить эту потребность, в спецификацию ISO/IEC SQL была добавлена функциональность, позволяющая преобразовывать данные JSON в формат, поддерживаемый в SQL, и наоборот: преобразовывать SQL-совместимые реляционные данные в данные JSON. Инструментом для выполнения таких преобразований является модель данных SQL/JSON, которая определяет различные элементы данных, а также набор встроенных функций для работы с этими элементами.

Загрузка и хранение данных JSON в реляционной базе данных

Чтобы сохранить данные JSON в реляционной базе данных, их передают в СУБД в виде строки символов или двоичной строки. Эта строка хранится как обычный столбец базы данных. После сохранения она может быть извлечена и обработана встроенными функциями SQL/JSON, описанными далее.

Генерирование данных JSON на основе реляционных данных

В ответ на SQL-запрос встроенные функции могут генерировать объекты или массивы JSON независимо от того, являются ли их источником данные JSON, которые были преобразованы в столбец таблицы SQL, или обычные данные SQL, которые изначально не были данными JSON.

Запрос данных JSON, хранящихся в реляционных таблицах

В инструкции SQL был встроен новый язык путей SQL/JSON, что позволяет запрашивать данные JSON, хранящиеся в таблицах реляционных баз данных.

Модель данных SQL/JSON

Поскольку SQL и JSON хранят данные совершенно разными способами, для совместного использования данных должен существовать способ преодоления этого разрыва. В качестве такого "моста" и применяется модель данных SQL/JSON.

Данные JSON могут быть представлены в различных форматах, включая массивы, объекты, элементы, литеральные пустые значения, литеральные истинные значения, литеральные ложные значения, числа и строки. В SQL отсутствует аналог массивов JSON, объектов JSON или элементов JSON. Кроме того, пустые значения, числа и строки в JSON не совсем совпадают с пустыми значениями, числами и строками SQL. С другой стороны, в JSON отсутствует аналог для данных даты и времени SQL. Для устранения этих проблем и был определен набор элементов SQL/JSON. Они находятся в среде SQL, но могут взаимодействовать с данными JSON, хранящимися вне этой среды.

Элементы SQL/JSON

Данные JSON, хранящиеся в виде символов или двоичных строк, в процессе анализа могут быть разбиты на отдельные элементы SQL/JSON. В качестве элемента SQL/JSON может использоваться:

- скалярное значение SQL/JSON;
- пустое значение SQL/JSON;
- массив SQL/JSON;
- объект SQL/JSON.

Скалярное значение SQL/JSON

Скалярное значение SQL/JSON определено как непустое значение любого из следующих типов SQL:

- символьная строка, включающая набор символов Unicode;
- числовое значение;
- булево значение;
- значение даты/времени.

Пустое значение SQL/JSON

Пустое значение SQL/JSON определяется как значение, отличное от произвольного значения любого типа SQL. Оно даже отличается от любого пустого значения SQL.

Массив SQL/JSON

Массив SQL/JSON определяется как упорядоченный список из нуля или более элементов SQL/JSON. Элементы в массиве разделяются запятыми, а сам массив заключен в квадратные скобки, например:

```
[ 3.1415927, "string theory", false]
```

Нумерация в массивах SQL начинается с 1, а это означает, что первый элемент считается элементом 1. В то же время SQL/JSON следует стандарту JavaScript, согласно которому нумерация в массивах начинается с 0. Первый элемент массива SQL/JSON считается элементом 0.

Объект SQL/JSON

Объект SQL/JSON определяется как неупорядоченная коллекция из нуля или более элементов SQL/JSON, где элемент — это пара, первое значение которой является строкой символов Unicode, а второе — элементом SQL/JSON. Первое значение элемента SQL/JSON называется ключом, а второе — связанным значением. Элементы иногда называются парами "ключ/значение", а иногда — парами "имя/значение". Объекты SQL/JSON можно сериализовать, разделяя элементы запятыми и заключая весь список в фигурные скобки. Примером может служить следующая строка:

```
{ "name" : "Joe Friday", "badge" : 714, "objective" : "the facts" }
```

Последовательности SQL/JSON

Последовательность SQL/JSON определяется как упорядоченный список, состоящий из нуля или более элементов SQL/JSON.

Синтаксический анализ JSON

Синтаксический анализ — это импорт данных, находящихся в каком-либо формате хранения, в модель данных SQL/JSON. Обычно таким форматом является строка символов Unicode, хотя возможны и другие, зависящие от реализации форматы.

Сериализация JSON

Сериализация JSON противоположна операции синтаксического анализа. Это экспорт значения из модели данных SQL/JSON обратно в некоторый формат хранения. Следует отметить, что значения даты и времени SQL/JSON не могут быть сериализованы. Другая проблема заключается в том, что последовательности SQL/JSON, длина которых больше единицы, также не могут быть сериализованы.

Функции SQL/JSON

Операции с данными JSON выполняются с помощью встроенных функций. Эти функции SQL/JSON относятся к двум группам: функции запросов и функции конструктора. Первые сравнивают выражения языка путей SQL/JSON (XPath) со значениями JSON, создавая значения типов SQL/JSON, которые затем преобразуются в типы SQL. Язык путей SQL/JSON будет описан далее.

Функции конструктора используют значения типов SQL для получения значений JSON (объектов или массивов), которые представлены в символьных или двоичных строковых типах SQL.

Общий синтаксис JSON API

Существует несколько функций запросов, причем все они имеют общий синтаксис. Все они требуют выражения XPath, значения JSON, которое будет вставлено в это выражение XPath для запроса и обработки, и, возможно, необязательных значений параметров, передаваемых в выражение XPath.

Применяется следующий синтаксис.

```
<общий синтаксис JSON API> ::=
   <элемент контекста JSON> <запятая>
      <спецификация пути JSON>
         [ AS <имя пути к таблице JSON> ]
         [ <передаваемое предложение JSON> ]
<элемент контекста JSON> ::=  <выражение JSON>
<спецификация пути JSON> ::= <строковый символьный литерал>
<передаваемое предложение JSON> ::= PASSING <аргумент JSON>
   [ { <запятая> <аргумент JSON> } ]
<аргумент JSON> ::= <выражение JSON> AS <идентификатор>
```

Выражение, содержащееся в *<выражение JSON>*, которое, в свою очередь, содержится в *<элемент контекста JSON>*, имеет строковый тип.

Выражение JSON

Как отмечалось выше, элемент контекста JSON является просто выражением JSON. Выражение JSON может быть определено следующим образом.

```
<выражение JSON> ::= <выражение> [ <входное предложение JSON> ]

<входное предложение JSON> ::= FORMAT <представление JSON>

<представление JSON> ::= JSON [ ENCODING { UTF8 | UTF16 | UTF32 } ]
   | параметр представления, определяемый реализацией JSON >
```

Как видите, выражение JSON является выражением, включающим необязательное входное предложение. Последнее задает формат представления JSON, определяющего кодировку. В качестве этой кодировки может применяться один из трех форматов Unicode либо альтернативный формат, определяемый реализацией.

Выражение XPath

За элементом контекста JSON и запятой следует спецификация пути JSON, которая должна быть символьным литералом. Имя пути к таблице и предложение PASSING являются необязательными частями спецификации пути JSON.

Предложение PASSING

Предложение PASSING применяется для передачи параметров выражению пути SQL/JSON.

Возвращаемое предложение JSON

Когда данные JSON возвращаются приложению в результате выполнения функции, разработчик приложения может указать тип данных, формат и кодировку текста JSON, созданного функцией. Предложение JSON OUTPUT имеет следующий синтаксис.

```
<возвращаемое предложение JSON> ::=
   RETURNING <тип данных>
   [ FORMAT <представление JSON> ]

<представление JSON> ::=
   JSON [ ENCODING { UTF8 | UTF16 | UTF32 } ]
 | параметр представления, определяемый реализацией JSON >
```

Если параметр FORMAT не указан, то по умолчанию применяется FORMAT JSON.

Функции запросов

К функциям запроса SQL/JSON относятся JSON_EXISTS, JSON_VALUE, JSON_QUERY и JSON_TABLE. Эти функции сравнивают выражения XPath со значениями JSON. В результате возвращаются значения типов SQL/JSON, которые преобразуются в типы данных SQL. Язык запросов XPath будет описан далее.

JSON_EXISTS

Функция JSON_EXISTS определяет, будет ли значение JSON удовлетворять условию отбора, заданному в спецификации пути. Эта функция имеет следующий синтаксис.

```
<предикат существования JSON> ::=
   JSON_EXISTS <левая скобка>
   <общий синтаксис JSON API>
       [ <поведение ошибки существования JSON> ON ERROR ]
   <правая скобка>
<поведение ошибки существования JSON> ::=
   TRUE | FALSE | UNKNOWN | ERROR
```

Если необязательное предложение ON ERROR отсутствует, то заданным по умолчанию предположением будет FALSE ON ERROR. Функция JSON_EXISTS оценивает выражение XPath и возвращает True, если выражение XPath находит один или несколько элементов SQL/JSON.

Образцы данных, которые можно использовать для изучения функций запросов, включая JSON_EXISTS, можно найти на страницах 24 и 25 раздела 6 технического отчета SQL ISO/IEC TR 19075-6: 2017 (E), доступного по следующему адресу:

```
http://standards.iso.org/ittf/PubliclyAvailableStandards/
c067367_ISO_IEC_TR_19075-6_2017.zip
```

Данные состоят из двух столбцов таблицы T: K и J. K — это первичный ключ таблицы, а J — данные, состоящие из пар "ключ/значение" и массивов пар "ключ/значение". Функция JSON_EXISTS проверяет существование заданного символьного литерала в спецификации XPath.

```
SELECT T.K
  FROM T
 WHERE JSON_EXISTS (T.J, 'lax $.where') ;
```

Первичные ключи строк, которые содержат слово 'where', возвращаются как набор результатов запроса SELECT. Ключевое слово 'lax' относится к обработке ошибок, которая более щадящая, чем "строгая" обработка ошибок. Это не влияет на результат данного запроса. $ — это метод доступа, который обращается к слову 'where' в текущем объекте JSON.

JSON_VALUE

Функция JSON_VALUE извлекает скалярное значение SQL из значения JSON. Она имеет следующий синтаксис.

```
<функция преобразования JSON> ::=
   JSON VALUE  <левая скобка>
   <общий синтаксис JSON API>
      [ <возвращаемое предложение JSON> ]
      [ <поведение пустого значения JSON> ON EMPTY ]
      [ <поведение ошибки значения JSON> ON ERROR ]
   <правая скобка>
<возвращаемое предложение JSON> ::= RETURNING <тип данных>
<поведение пустого значения JSON> ::=
   ERROR
 | NULL
 | DEFAULT <выражение>
<поведение ошибки значения JSON> ::= ERROR
 | NULL
 | DEFAULT <выражение>
```

Как можно было предположить, выражение *<поведение пустого значения JSON>* определяет возвращаемый результат в случае, когда выражение XPath будет пустым.

- NULL ON EMPTY означает, что результат функции JSON_VALUE пустой.
- ERROR ON EMPTY означает, что будет сгенерировано исключение.
- DEFAULT *<выражение>* ON EMPTY означает, что выражение оценивается и приводится к целевому типу.
- *<поведение ошибки значения JSON>* трактуется аналогичным образом. Оно определяет, что нужно делать в случае, если имеется необработанная ошибка.

В предыдущем примере с функцией JSON_EXISTS возвращались все строки, в которых в столбце J содержалось ключевое слово 'where'. В случае функции JSON_VALUE возвращается значение, связанное с целевым ключевым словом. Если взять набор данных из примера с функцией JSON_EXISTS, в котором ключевое слово 'who' сопоставляется с с именем человека, то следующий код SQL вернет имена сотрудников из всех строк, в которых находится ключевое слово 'who'.

```
SELECT  T.K,
   JSON_VALUE (T.J, 'lax $.who') AS Who
  FROM  T ;
```

Результирующий набор будет включать столбец с именем K, содержащий первичные ключи возвращаемых строк, и столбец с именем Who, содержа-

щий имена, которые были связаны с ключевым словом 'who' в исходных данных.

По умолчанию функция JSON_VALUE возвращает тип данных символьной строки, определяемый реализацией. С помощью предложения RETURNING можно задать другие типы данных.

JSON_QUERY

Функция JSON_VALUE отлично справляется с извлечением скалярных значений из значения SQL/JSON, но она не способна извлечь массив или объект SQL/JSON из значения SQL/JSON. Для этого предназначена функция JSON_QUERY, имеющая следующий синтаксис.

```
<запрос JSON> ::=
   JSON_QUERY <левая скобка>
      <общий синтаксис JSON API>
      [ <возвращаемое предложение JSON> ]
      [ <поведение оболочки запроса JSON> ]
      [ <поведение запроса в кавычках JSON> QUOTES
         [ON SCALAR STRING] ]
      [ <поведение пустого запроса JSON> ON EMPTY ]
      [ <поведение ошибки запроса JSON> ON ERROR ]
   <правая скобка>
```

Предложения ON EMPTY и ON ERROR подобны аналогичным предложениям функции JSON_VALUE, к тому же они обрабатываются схожим образом. Разница заключается в том, что пользователь может указать поведение в случае пустого значения или ошибки.

- Если <*возвращаемое предложение JSON*> не указано, по умолчанию применяется RETURNING JSON FORMAT.
- Если <*поведение пустого запроса JSON*> не указано, по умолчанию применяется NULL ON EMPTY.
- Если <*поведение ошибки запроса JSON*> не указано, по умолчанию применяется NULL ON ERROR.
- Если <*поведение оболочки запроса JSON*> не указано, по умолчанию применяется WITHOUT ARRAY.
- Если <*поведение оболочки запроса JSON*> задано как WITH и ключевые слова CONDITIONAL и UNCONDITIONAL не указаны, то по умолчанию применяется UNCONDITIONAL.
- Если значение <*элемент контекста JSON*>, которое содержится в предложении <*общий синтаксис JSON API*>, является пустым значением, то результат выполнения <*запрос JSON*> является пустым значением.

Используя те же данные, которые применялись для функций JSON_EXISTS и JSON_VALUE, можно добавить массив данных в набор результатов вместе с результатами, полученными с помощью функций JSON_VALUE.

```
SELECT  T.K,
   JSON_VALUE (T.J, 'lax $.who') AS Who,
   JSON_VALUE (T.J, 'lax $.where' NULL ON EMPTY) AS Nali,
   JSON_QUERY (T.J, 'lax $.friends') AS Friends
  FROM T
 WHERE JSON_EXISTS (T.J, 'lax $.friends')
```

Предложение WHERE JSON_EXISTS исключает произвольные строки, которые не содержат пару "ключ/значение" для friends. Если указано предложение WITH ARRAY WRAPPER, то возвращаемые элементы массива будут заключены в квадратные скобки.

JSON_TABLE

Функция JSON_TABLE намного сложнее, чем другие функции запросов. Она получает данные JSON и генерирует реляционную выходную таблицу на основании корректных входных данных. Определение синтаксиса для простейшего варианта функции JSON_TABLE занимает более одной страницы. Ну а в случае добавления вложенных выражений XPath и планов сложность синтаксиса еще больше возрастает. К сожалению, в книге просто нет места для исчерпывающего описания функции JSON_TABLE, поэтому я отсылаю вас к странице 35 и последующим страницам технического отчета SQL Technical Report ISO/IEC TR 19075-6:2017(E), который можно загрузить по следующему адресу:

```
http://standards.iso.org/ittf/PubliclyAvailableStandards/
c067367_ISO_IEC_TR_19075-6_2017.zip
```

Функции конструктора

Функции конструктора SQL/JSON позволяют создавать объекты, массивы и агрегаты JSON на основе информации, которая хранится в реляционных таблицах. При выполнении операций эти функции получают данные в направлении, которое противоположно направлению приема данных функциями запроса SQL/JSON.

JSON_OBJECT

Функция JSON_OBJECT конструирует объекты JSON на основе явно указанных пар "имя/значение". Она имеет следующий синтаксис:

```
<конструктор объекта JSON> ::=
   JSON_OBJECT <левая скобка>
      [ <имя и значение JSON> [ { <запятая>
```

```
            <имя и значение JSON> } ... ]
      [ <предложение пустого конструктора JSON> ]
      [ <ограничение уникального ключа JSON> ] ]
      [ <возвращаемое предложение JSON> ]
   <правая скобка>
<имя и значение JSON> ::=
   [KEY] <имя JSON> VALUE <выражение JSON>
 | <имя JSON> <двоеточие> <выражение JSON>
<имя JSON> ::= <символьное выражение>
<предложение пустого конструктора JSON> ::=
   NULL ON NULL
 | ABSENT ON NULL
<ограничение уникальности ключа JSON> ::=
   WITH UNIQUE [KEYS]
 | WITHOUT UNIQUE [KEYS]
```

Для этого синтаксиса должны соблюдаться следующие правила.

- *<имя JSON>* не может быть NULL.
- *<выражение JSON>* может быть NULL.
- *<предложение пустого конструктора JSON>*, если задано ограничение NULL ON NULL, генерирует пустое значение SQL/JSON. Если задано ограничение ABSENT ON NULL, то пропускается пара "ключ/значение" из результирующего объекта SQL/JSON.
- При отсутствии *<предложение пустого конструктора JSON>* по умолчанию применяется ограничение NULL ON NULL.

JSON_OBJECTAGG

Разработчик приложений может конструировать объект JSON путем агрегирования данных реляционной таблицы. Если таблица содержит два столбца, один — с именами JSON, а второй — со значениями JSON, то функция JSON_OBJECTAGG может использовать эти данные для создания объекта JSON. Функция имеет следующий синтаксис.

```
<конструктор агрегата объекта JSON> ::=
   JSON_OBJECTAGG <левая скобка>
      <имя и значение JSON>
      [ <предложение пустого конструктора JSON> ]
      [ <ограничение уникальности ключа JSON> ]
      [ <возвращаемое предложение JSON> ]
   <правая скобка>
```

Если *<предложение пустого конструктора JSON>* отсутствует, то по умолчанию используется ограничение NULL ON NULL.

JSON_ARRAY

Чтобы создать массив JSON на основании списка элементов данных из реляционной таблицы, можно использовать функцию JSON_ARRAY, которая имеет следующий синтаксис.

```
<конструктор массива JSON> ::=
   <конструктор массива JSON по перечислению>
 | <конструктор массива JSON по запросу>
<конструктор массива JSON по перечислению> ::=
   JSON_ARRAY <левая скобка>
      [ <выражение JSON> [ { <запятая> <выражение JSON> }... ]
        <предложение пустого конструктора JSON> ]
      <возвращаемое предложение JSON>
   <правая скобка>
<конструктор массива JSON по запросу> ::=
   JSON_ARRAY <левая скобка>
   <выражение запроса>
   [ <входное выражение JSON> ]
   [ <выражение пустого конструктора JSON> ]
   [ <возвращаемое выражение JSON> ]
```

Функция JSON_ARRAY имеет два варианта, один из которых генерирует результаты на основании входного списка значений SQL, а второй — на основании запроса, вызываемого внутри функции. Если необязательное *<предложение пустого конструктора JSON>* отсутствует, то по умолчанию используется ограничение ABSENT ON NULL, которое является противоположным поведению, заданному по умолчанию для функции JSON_OBJECT.

JSON_ARRAYAGG

Точно так же, как можно создавать объект JSON путем агрегирования реляционных данных, можно создать массив JSON на основе агрегации реляционных данных. Чтобы реализовать подобное поведение, в стандарт SQL была включена функция JSON_ARRAYAGG, имеющая следующий синтаксис.

```
<конструктор агрегата массива JSON> ::=
   JSON_ARRAYAGG <левая скобка>
      <выражение JSON>
      [ <предложение сортировки агрегированного массива JSON> ]
      [ <предложение пустого конструктора JSON> ]
      [ <возвращаемое предложение JSON> ]
   <правая скобка>
<предложение сортировки агрегированного массива JSON> ::=
   ORDER BY <список спецификаций сортировки>
```

Если *<предложение пустого конструктора JSON>* отсутствует, то по умолчанию применяется ограничение ABSENT ON NULL. Кроме того,

<предложение сортировки агрегированного массива JSON> позволяет разработчику упорядочивать выводимые элементы в соответствии с одной или несколькими спецификациями сортировки, аналогично тому, как это делает предложение `ORDER BY` в стандартных SQL-запросах.

Предикат `IS JSON`

Предикат `IS JSON` проверяет, действительно ли строка, предположительно содержащая данные JSON, является таковой. Синтаксис предиката таков.

```
<предикат JSON> ::=
   <строковое выражение> [ <входное предложение JSON> ]
         IS [NOT] JSON
      [ <ограничение типа предиката JSON> ]

      [ <ограничение уникальности ключа JSON> ]
<ограничение типа предиката JSON> ::=
   VALUE
 | ARRAY
 | OBJECT
 | SCALAR
```

Если необязательное *<входное предложение JSON>* не указано, то по умолчанию применяется `FORMAT JSON`. Если *<ограничение уникальности ключа JSON>* не указано, по умолчанию применяется `WITHOUT UNIQUE KEYS`.

Пустые значения в JSON и SQL

Учтите, что пустые значения в JSON не совпадают с пустыми значениями в SQL. В SQL строка нулевой длины (`""`) отличается от пустого значения SQL, которое означает отсутствие явно заданного значения. В JSON пустое значение является фактическим значением, представляемым литералом JSON (`"null"`). Пустые значения JSON должны отличаться от пустых значений SQL. Синтаксис SQL/JSON позволяет разработчику приложения выбирать, включать ли пустые значения SQL в создаваемый объект или массив JSON или исключать их.

Язык XPath в SQL/JSON

XPath является языком запросов, используемым функциями запросов SQL/JSON. Инструкции этого языка получают элемент контекста, спецификацию пути и предложение `PASSING` в качестве входных значений (возможно еще предложение `ON ERROR` и др.) и вызывают функцию `JSON_EXISTS`, `JSON_VALUE`,

JSON_QUERY или JSON_TABLE. Эти функции выполняются "движком" XPath, который возвращает результаты пользователю.

В языке XPath знак доллара ($) представляет элемент текущего контекста, а точка (.) — элемент объекта. Элементы массива заключаются в квадратные скобки. Таким образом:

- `$.name` обозначает значение атрибута `name` текущего объекта JSON;
- `$.phones[last]` обозначает последний элемент массива, сохраненного в атрибуте `phones` текущего объекта JSON.

Дополнительные сведения

В рамках одной главы довольно сложно осветить все вопросы, связанные с JSON. Чтобы получить дополнительные сведения о том, как JSON применяется в SQL Server, обратитесь по следующему адресу:

```
https://docs.microsoft.com/en-us/sql/relational-databases/json/
json-data-sql-server?view=sql-server-2017
```

Расширенные возможности SQL

В ЭТОЙ ЧАСТИ...

- Создание курсоров
- Составные инструкции
- Обработка ошибок
- Применение триггеров

Глава 20

Обработка наборов данных с помощью курсоров

В ЭТОЙ ГЛАВЕ...

- Определение области действия курсора с помощью инструкции `DECLARE`
- Открытие курсора
- Построчная выборка данных
- Закрытие курсора

SQL отличается от большинства популярных языков программирования тем, что операции в нем выполняются сразу над целым набором строк таблицы, в то время как процедурные языки обрабатывают данные построчно. Благодаря использованию *курсоров* в SQL можно извлекать (а также обновлять и удалять) данные одной записи, т.е. также действовать построчно, тем самым упрощая использование SQL в приложении, написанном на другом языке программирования.

По сути, курсор подобен указателю на конкретную строку таблицы. С его помощью можно извлечь, обновить или удалить строку, на которую он ссылается.

Курсоры незаменимы, когда требуется выбрать строки из таблицы, проверить их содержимое, а также выполнить различные операции в зависимости от 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 извлекать строки из таблицы по одной и передавать их в процедурный код для обработки. Поместив SQL-код в цикл, можно строка за строкой полностью обработать всю таблицу.

В случае использования внедренного SQL обобщенная процедура такой обработки выглядит следующим образом.

```
EXEC SQL DECLARE CURSOR инструкция
EXEC SQL OPEN инструкция
Проверка достижения конца таблицы
Процедурный код
Начало цикла
    Процедурный код
    EXEC SQL FETCH
    Процедурный код
    Проверка достижения конца таблицы
Конец цикла
EXEC SQL CLOSE инструкция
Процедурный код
```

В приведенном фрагменте кода использованы следующие инструкции SQL: DECLARE CURSOR, OPEN, FETCH и CLOSE. Каждую из них мы детально рассмотрим в данной главе.


СОВЕТ

Если для выполнения операций с выбранными строками можно обойтись обычными инструкциями SQL, то лучше так и поступить. Используйте для построчной обработки процедурный язык лишь в том случае, если возможностей SQL недостаточно для выполнения необходимых операций.

Объявление курсора

Чтобы использовать курсор, необходимо сначала сообщить СУБД о его существовании. Это делается с помощью инструкции DECLARE CURSOR. Фактически она не инициирует никаких действий, а лишь объявляет имя курсора для СУБД и определяет запрос, с которым он будет работать. Синтаксис объявления курсора следующий.

```
DECLARE имя_курсора [<чувствительность>]
   [<перемещаемость>]
CURSOR [<режим_фиксации>] [<возвращаемость>]
FOR выражение_запроса
   [ORDER BY порядок_сортировки]
   [FOR разрешение_обновления] ;
```

Примечание: имя курсора определяет его однозначно, следовательно, оно должно отличаться от любого другого имени курсора в текущем модуле или программе.


СОВЕТ

Чтобы код был понятен другим пользователям, курсору следует присвоить осмысленное имя. Оно должно быть связано либо с данными, обрабатываемыми в запросе, либо с операциями, которые программный код выполняет с этими данными.

При объявлении курсора можно задать некоторые его характеристики.

- **Чувствительность.** Доступны следующие уровни чувствительности: `SENSITIVE`, `INSENSITIVE` и `ASENSITIVE` (по умолчанию).
- **Перемещаемость.** Доступны варианты `SCROLL` и `NO SCROLL` (по умолчанию).
- **Режим фиксации.** Доступны варианты `WITH HOLD` и `WITHOUT HOLD` (по умолчанию).
- **Возвращаемость.** Доступны варианты `WITH RETURN` и `WITHOUT RETURN` (по умолчанию).

Выражение запроса


ЗАПОМНИ!

В качестве *выражения запроса* можно использовать любую допустимую инструкцию `SELECT`. Эта инструкция отбирает строки, по которым может перемещаться курсор. Они образуют область его действия.

Запрос не будет выполняться сразу же после запуска инструкции `DECLARE CURSOR` — извлечение данных инициируется только инструкцией `OPEN`. Построчная обработка данных начинается после входа в цикл, включающий инструкцию `FETCH`.

Предложение `ORDER BY`

Порядок обработки извлекаемых из таблицы данных может зависеть от того, что именно программный код должен делать с данными. Сортировку строк перед их обработкой можно выполнять с помощью предложения `ORDER BY`, синтаксис которого имеет следующий вид:

```
ORDER BY спецификация_сортировки[, спецификация_сортировки]...
```

Можно использовать несколько спецификаций сортировки, каждая из которых имеет следующий синтаксис:

```
(имя_столбца) [COLLATE BY имя_схемы_сортировки] [ASC|DESC]
```

Чтобы иметь возможность сортировать по столбцу, его имя должно стоять в списке SELECT выражения запроса. Сортировку по столбцам таблицы, не вошедшим в список запроса, осуществить нельзя. Предположим, со строками таблицы CUSTOMER требуется выполнить операцию, не поддерживаемую в SQL. Для этого можно создать такой курсор.

```
DECLARE cust1 CURSOR FOR
   SELECT CustID, FirstName, LastName, City, State, Phone
     FROM CUSTOMER
    ORDER BY State, LastName, FirstName ;
```

В этом примере инструкция SELECT возвращает строки, упорядоченные вначале по штату (State), а затем по фамилии (LastName) и имени (FirstName). Первыми в списке будут стоять заказчики из штата Аляска (код АК), затем заказчики из штата Алабама (код AL) и т.д. В пределах штата заказчики сортируются по фамилиям (Абель предшествует Адамсу). Там, где фамилии совпадают, сортировка выполняется по имени (Генри Адамс предшествует Джорджу Адамсу).

Приходилось ли вам когда-нибудь делать 40 копий 20-страничного документа на копировальном аппарате без сортировщика? Поверьте — страшно утомительная работа! Нужно отвести на столах место для двадцати стопок листов, которые будут соответствовать двадцати страницам документа, и ходить между ними сорок раз туда и обратно, раскладывая страницы сорока копий по всем стопкам. Такой процесс называется *сортировкой*. Аналогичный процесс возможен и в SQL.

Схема сортировки — это набор правил, определяющий способ сравнения строк в наборе символов. Любой набор символов имеет стандартную схему сортировки (которая действует по умолчанию), однако при желании ее можно изменить. Для этого следует использовать необязательное предложение COLLATE BY. Любая СУБД изначально поддерживает несколько наиболее распространенных схем сортировки, из которых можно выбрать нужную. Кроме того, можно указать, как упорядочивать данные: *по возрастанию* или *по убыванию*. Для этого в конце предложения нужно добавить ключевое слово ASC или DESC соответственно.

В инструкции DECLARE CURSOR можно задать вычисляемый столбец, не существующий в исходной таблице. У такого столбца нет имени, которое можно было бы указать в предложении ORDER BY. Чтобы обеспечить возможность ссылки на такой столбец, его имя нужно определить в выражении запроса. 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 предложение ORDER BY не содержит предложения COLLATE BY. Таким образом, используется схема сортировки по умолчанию. Обратите внимание на то, что четвертый столбец, заданный в списке выборки, представляет собой результат умножения значений, хранящихся во втором и третьем столбцах. Этот вычисляемый столбец, которому присвоено имя ExtPrice, содержит данные о совокупной стоимости изделий определенной модели. В предложении ORDER BY вначале задается сортировка по названию модели (Model), а затем — по совокупной стоимости (ExtPrice). При этом сортировка по столбцу ExtPrice выполняется по убыванию (поскольку использовано ключевое слово DESC), т.е. вначале указываются более дорогие транзакции.

В предложении ORDER BY по умолчанию используется порядок сортировки по возрастанию. Если в списке спецификаций стоит ключевое слово DESC и следующая сортировка также должна выполняться по убыванию, то для нее тоже следует задать ключевое слово DESC в явном виде. Например, выражение

```
ORDER BY A, B DESC, C, D, E, F
```

эквивалентно выражению

```
ORDER BY A ASC, B DESC, C ASC, D ASC, E ASC, F ASC
```

Разрешение обновления

Иногда необходимо обновить либо удалить строки таблицы, отобранные с помощью курсора, а иногда — гарантированно исключить подобные операции. SQL предоставляет возможность осуществлять контроль с помощью предложения FOR *разрешение_обновления* инструкции DECLARE CURSOR. Чтобы запретить обновление и удаление в области действия курсора, используйте следующее предложение:

```
FOR READ ONLY
```

Чтобы разрешить обновление только некоторых столбцов, используйте следующее предложение (при этом обновление остальных столбцов будет запрещено):

```
FOR UPDATE OF имя_столбца[, имя_столбца]...
```

Столбцы, добавленные в список доступных для обновления, должны быть указаны в выражении запроса инструкции DECLARE CURSOR. Если разрешение обновления отсутствует, то по умолчанию данные всех столбцов, указанных в выражении запроса, доступны для

обновления. В таком случае инструкция UPDATE может обновить все столбцы в строке, на которую указывает курсор, а инструкция DELETE может удалить эту строку.

Чувствительность

Выражение запроса в инструкции DECLARE CURSOR определяет строки, попадающие в область действия курсора. Рассмотрим одну из возможных проблем. Что произойдет, если программный код, расположенный между инструкциями OPEN и CLOSE, изменит содержимое некоторых строк так, что они не будут удовлетворять условиям запроса? Продолжит ли курсор обрабатывать все строки, которые изначально соответствовали условиям запроса, или обнаружит наличие новой ситуации и будет игнорировать неподходящие теперь строки?

Обычная инструкция SQL, такая как UPDATE, INSERT или DELETE, работает с набором строк таблицы (или, возможно, со всей таблицей). Пока такая инструкция активна, механизм SQL-транзакций защищает ее от конкуренции с другими инструкциями, параллельно работающими с теми же данными. Но если вы используете курсор, то дверь для постороннего вмешательства остается широко открытой. Когда вы открываете курсор, данные подвергаются риску стать жертвой одновременных противоречивых операций до тех пор, пока вы не закроете курсор. Если вы открываете один курсор и запускаете обработку таблицы, а затем открываете второй курсор, то в период, пока первый курсор остается активным, действия, предпринимаемые вторым курсором, могут повлиять на объект инструкции, контролируемой первым курсором.


ЗАПОМНИ!

Изменение данных в столбцах, указанных в выражении запроса инструкции DECLARE CURSOR, до того, как все строки запроса будут обработаны, вызовет хаос. Данные могут стать недостоверными и противоречивыми. По этой причине рекомендуется сделать курсор нечувствительным к любым изменениям в области его действия. Для этого в инструкцию DECLARE CURSOR нужно добавить ключевое слово INSENSITIVE. В таком случае, пока курсор остается открытым, он нечувствителен к изменениям таблицы, влияющим на строки, которые попадают в область его действия. Курсор не может быть одновременно нечувствительным и обновляемым. Нечувствительный курсор должен быть открыт только для чтения.

Предположим, например, что вы составили следующие запросы.

```
DECLARE C1 CURSOR FOR SELECT * FROM EMPLOYEE
  ORDER BY Salary ;
DECLARE C2 CURSOR FOR SELECT * FROM EMPLOYEE
    FOR UPDATE OF Salary ;
```

Теперь предположим, что вы открыли оба курсора и с помощью курсора C1 извлекли несколько строк, а затем с помощью курсора C2 обновили данные по зарплате, увеличив оклады служащих. В результате может случиться так, что строка, уже пройденная курсором C1, войдет в состав последующей выборки курсора C1.

Проблем, возникающих в результате взаимодействий нескольких одновременно открытых курсоров, можно избежать только путем изоляции транзакций. В противном случае можно столкнуться с большими проблемами. Поэтому запомните: нельзя работать с несколькими открытыми курсорами. Более подробно об уровнях изоляции транзакций см. в главе 15.

По умолчанию чувствительность курсора задана как ASENSITIVE. Если вы думаете, что значением этого ключевого слова всегда является отсутствие чувствительности, то вы ошибаетесь. Каждая конкретная СУБД понимает его по-своему. В одной СУБД оно может быть эквивалентно SENSITIVE, а в другой — INSENSITIVE. Поэтому обратитесь к документации по своей СУБД.

Перемещаемость

Перемещаемость означает возможность перемещать курсор в пределах результирующего набора. Ключевое слово SCROLL в инструкции DECLARE CURSOR позволяет получить доступ к строкам в любом выбранном вами порядке. Методы управления курсором реализованы в синтаксисе инструкции FETCH, которую мы рассмотрим далее.

Открытие курсора

Несмотря на то что инструкция DECLARE CURSOR определяет, какие строки включать в курсор, она фактически не инициирует никакого действия, являясь лишь объявлением. Дать "жизнь" курсору способна только инструкция OPEN. Она имеет следующий синтаксис:

```
OPEN имя_курсора ;
```

Чтобы открыть курсор, описанный в разделе "Предложение ORDER BY", используйте следующий код.

```
DECLARE revenue CURSOR FOR
   SELECT Model, Units, Price,
          Units * Price AS ExtPrice
     FROM TRANSDETAIL
```

```
    ORDER BY Model, ExtPrice DESC ;
OPEN revenue ;
```

До тех пор пока курсор не открыт, осуществлять выборку строк с его помощью невозможно. После открытия курсора значения переменных, указанные в инструкции DECLARE CURSOR, становятся фиксированными, как и текущие значения всех функций даты/времени. Рассмотрим пример.

```
EXEC SQL DECLARE C1 CURSOR FOR SELECT * FROM ORDERS
   WHERE ORDERS.Customer = :NAME
     AND DueDate < CURRENT_DATE ;
NAME :='Acme Co';      // Инструкция базового языка
EXEC SQL OPEN C1;
NAME :='Omega Inc.';  // Еще одна инструкция базового языка
...
EXEC SQL UPDATE ORDERS SET DueDate = CURRENT_DATE;
```

Инструкция OPEN фиксирует значения всех переменных, указанных в объявлении курсора, а также текущие значения всех функций даты/времени. В результате второе присваивание переменной имени (NAME := 'Omega Inc.') *не влияет* на строки, входящие в область действия курсора. (Значение переменной NAME используется при *следующем* открытии курсора C1.) И даже если инструкция OPEN выполнилась за минуту до полуночи, а инструкция UPDATE — через минуту после полуночи, значение CURRENT_DATE в инструкции UPDATE будет равно значению этой функции, полученному в момент выполнения инструкции OPEN.

ВНУТРЕННЯЯ ФИКСАЦИЯ ТРАНЗАКЦИЙ (ДЛЯ ЗНАЧЕНИЙ ДАТЫ И ВРЕМЕНИ)

Как уже говорилось выше, инструкция OPEN фиксирует значения всех переменных, упомянутых при объявлении курсора. Она также фиксирует значения функций даты/времени. Аналогичная фиксация значений даты и времени происходит и в операциях SET. Рассмотрим следующий пример.

```
UPDATE ORDERS SET RecheckDate = CURRENT_DATE WHERE ...;
```

Теперь предположим, что вы приняли несколько заказов. Выполнение этого запроса начинается за минуту до полуночи и продолжается, допустим, шесть минут. Это не так уж важно. Если инструкция использует функцию CURRENT_DATE (или TIME, или TIMESTAMP), ее значение устанавливается равным дате (и времени) начала инструкции. Таким образом, все строки таблицы ORDERS в этой инструкции получают одинаковое значение текущей даты (RecheckDate). Подобным образом обстоит дело и со значениями типа TIMESTAMP — вся ин-

струкция использует одно и то же значение независимо от длительности ее выполнения.

Ниже приведен интересный пример применения такого правила.

```
UPDATE EMPLOYEE SET KEY = CURRENT_TIMESTAMP ;
```

На первый взгляд кажется, будто эта инструкция присваивает каждому сотруднику уникальное значение в столбце KEY, поскольку время измеряется в долях секунды. Но в действительности во всех строках будет установлено одно и то же значение. Поэтому для создания уникального ключа вам придется найти другой способ.

Таким образом, инструкция OPEN фиксирует значения даты/времени для всех инструкций, ссылающихся на курсор, — она воспринимает их как свое естественное продолжение.

Извлечение данных из отдельных строк

Использование курсора представляет собой трехэтапный процесс.

1. Инструкция DECLARE CURSOR определяет имя курсора и область его действия.
2. Инструкция OPEN отбирает строки таблицы, удовлетворяющие выражению запроса в инструкции DECLARE CURSOR.
3. Инструкция FETCH осуществляет непосредственное извлечение данных.

Курсор может указывать на некоторую строку в своей области действия, или на строку, находящуюся непосредственно перед первой строкой либо сразу после последней строки области действия, или на пустое пространство между двумя строками. Определить, на что именно указывает курсор, можно с помощью предложения ориентации в инструкции FETCH.

Синтаксис

Инструкция FETCH имеет следующий синтаксис:

```
FETCH [[ориентация] FROM] имя_курсора
   INTO спецификация_приемника[, спецификация_приемника]... ;
```

Существуют семь вариантов ориентации:

- NEXT (вперед);
- PRIOR (назад);
- FIRST (к первой записи);
- LAST (к последней записи);

- ABSOLUTE (абсолютное положение);
- RELATIVE (относительное положение);
- *<спецификация_простого_значения>*.

По умолчанию используется вариант NEXT (вперед) — единственно возможное значение этого параметра, существовавшее до выхода стандарта SQL-92. Где бы ни находился курсор, он может перемещаться на следующую строку выборки, заданной выражением запроса. Если курсор находится перед первой записью, то он перемещается к первой записи. Если он указывает на запись *n*, то перемещается к записи *n*+1. Если курсор указывает на последнюю запись выборки, то он перемещается за пределы этой записи, а в системную переменную SQLSTATE заносится код ошибки в связи с отсутствием данных. (Переменная SQLSTATE и другие средства обработки ошибок в SQL рассматриваются в главе 22.)

Спецификациями приемников являются либо базовые переменные, либо параметры — все зависит от того, определен ли курсор во встроенном SQL-коде либо в модуле. Количество и типы приемников должны соответствовать количеству и типам столбцов, заданных в выражении запроса инструкции DECLARE CURSOR. В случае внедренного SQL-кода, если из строки таблицы извлекается список из пяти значений, то в выражении запроса должны фигурировать пять базовых переменных с правильно заданными типами для получения этих значений.

Ориентация перемещаемого курсора

Поскольку SQL-курсор может быть перемещаемым, помимо варианта NEXT можно использовать и другие значения параметра ориентации. Ориентация PRIOR перемещает курсор к предыдущей строке, FIRST — к первой записи, а LAST — к последней записи выборки.

При использовании вариантов ориентации ABSOLUTE и RELATIVE нужно дополнительно указывать целочисленное значение. Например, инструкция FETCH ABSOLUTE 7 перемещает курсор на седьмую строку от начала выборки, а FETCH RELATIVE 7 — на семь строк ниже текущей позиции. Инструкция FETCH RELATIVE 0 не перемещает курсор.

Инструкция FETCH RELATIVE 1 имеет тот же эффект, что и FETCH NEXT. Действие инструкции FETCH RELATIVE -1 идентично действию FETCH PRIOR. Инструкция FETCH ABSOLUTE 1 переходит к первой записи выборки, FETCH ABSOLUTE 2 — ко второй и т.д. Аналогично инструкция FETCH ABSOLUTE -1 переходит к последней записи, FETCH ABSOLUTE -2 — к предпоследней и т.д. Инструкция FETCH ABSOLUTE 0 возвращает код ошибки в связи с отсутствием данных; такой же результат будет при выполнении инструкции FETCH

ABSOLUTE 17, если в выборке есть только 16 строк. Инструкция FETCH <*спецификация_простого_значения*> возвращает запись, заданную спецификацией простого значения.

Позиционные инструкции DELETE и UPDATE

Со строками, на которые в текущий момент указывает курсор, можно выполнять операции удаления и обновления. Синтаксис инструкции DELETE в этом случае будет таким.

```
DELETE FROM имя_таблицы WHERE CURRENT OF имя_курсора;
```

Если курсор не указывает на строку выборки, то инструкция возвращает код ошибки, а само удаление не выполняется.

Инструкция UPDATE имеет следующий синтаксис.

```
UPDATE имя_таблицы
   SET имя_столбца = значение[, имя_столбца = значение]...
 WHERE CURRENT OF имя_курсора ;
```

Значение, помещаемое в каждый выбранный столбец, должно быть выражением или ключевым словом DEFAULT. Если при выполнении операции обновления возникает какая-либо ошибка, обновление не выполняется.

Закрытие курсора

Закончив работу с курсором, сразу же закройте его. Курсоры, оставшиеся открытыми после того, как приложение закончило работу с ними, могут стать источником неприятностей. Кроме того, открытые курсоры поглощают системные ресурсы.

Если закрывается курсор, который был определен как нечувствительный к изменениям, внесенным в то время, пока он был открыт, то при последующем его открытии все эти изменения будут в нем отражены.

В качестве примера закроем курсор, открытый ранее для таблицы TRANSDETAIL:

```
CLOSE revenue ;
```

Глава 21

Процедурное программирование и хранимые модули

В ЭТОЙ ГЛАВЕ...

- Выполнение составных инструкций
- Управляющие конструкции
- Создание циклов
- Использование хранимых процедур и функций
- Создание хранимых модулей

Вот уже в течение многих лет ведущие специалисты в области баз данных занимаются разработкой стандартов. После выхода очередного стандарта сразу начинается подготовка следующего. Семь лет разделяет появление стандарта SQL-92 и публикацию первого компонента SQL:1999. Однако все эти годы продолжалась бурная деятельность, в результате которой организации ANSI и ISO выпустили дополнение под названием "SQL-92/PSM" (Persistent Stored Modules), которое включало описание хранимых модулей. Это дополнение послужило основой для одноименного раздела стандарта SQL:1999. Стандарт SQL/PSM определяет набор управляющих конструкций, подобных используемым в стандартных процедурных языках программирования. Благодаря этому стало возможным решать многие задачи только с помощью SQL, без привлечения других программных средств. Теперь программистам с трудом верится, что для выполнения своей работы им приходилось

постоянно переключаться между инструкциями SQL и командами процедурного языка.

Составные инструкции

До сих пор SQL рассматривался как непроцедурный язык, который работает с наборами данных, а не с отдельными записями. С добавлением средств, рассматриваемых в данной главе, это положение теряет свою актуальность. В то же время, приобретая некоторые черты процедурности, SQL все еще остается языком обработки наборов данных.

"Доисторическая" версия SQL, определяемая стандартом SQL-92, не соответствовала процедурной модели, которая предполагает последовательное выполнение команд для достижения нужного результата. Инструкции SQL были одиночными и, как правило, встроенными в код программы на C++ или Visual Basic. В этих ранних версиях SQL пользователи не могли создать запрос или выполнить некоторые другие операции с помощью последовательности инструкций SQL, поскольку это создавало значительную нагрузку на сеть и вызывало снижение производительности. Стандарт SQL:1999 и все последующие версии предоставили возможность создания составных инструкций, построенных из отдельных простых SQL-инструкций и выполняемых как единое целое, что снижает нагрузку на сеть.

Все инструкции, включенные в составную инструкцию, должны быть расположены между ключевыми словами `BEGIN` и `END`. Например, чтобы вставить данные в несколько связанных таблиц, следует использовать синтаксис, подобный следующему.

```
void main {
   EXEC SQL
      BEGIN
         INSERT INTO students (StudentID, Fname, Lname)
              VALUES (:sid, :sfname, :slname) ;
         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 предмет ошибки */
}
```

Приведенный выше фрагмент программы на языке C содержит встроенную составную инструкцию SQL. Комментарий в конце заменяет код обработки ошибок. Если по какой-то причине составная инструкция не выполнилась, в

переменную SQLSTATE будет помещен код ошибки. Естественно, комментарий, стоящий после ключевого слова END, не способен исправить ошибку; в реальной программе его должен заменить код обработки ошибок. Более детально мы рассмотрим этот вопрос в главе 22.

Атомарность

Составные инструкции создают возможность для возникновения ошибок, которые отсутствовали в простых инструкциях SQL. Выполнение обычной SQL-инструкции может завершиться успешно или нет, и в последнем случае база данных не претерпевает изменений. В случае возникновения ошибки при выполнении составных инструкций ситуация совсем иная.

Рассмотрим пример, приведенный в предыдущем разделе. Что произойдет, если операции вставки (INSERT) в таблицы STUDENTS (студенты) и ROSTER (расписание) прошли, но из-за вмешательства постороннего пользователя операция вставки в таблицу RECEIVABLE (оплата за учебу) не была выполнена. В результате студент будет зачислен в университет, но счет за обучение ему выставлен не будет. Такие ошибки могут слишком дорого обойтись университету.

Все дело в том, что в рассмотренном сценарии отсутствует концепция атомарности. Атомарная инструкция является неделимой: она либо выполняется целиком, либо не выполняется вовсе. Простые SQL-инструкции атомарны по своей природе. Другое дело — составные инструкции. Однако и составную инструкцию можно сделать атомарной, явно определив ее таковой. В следующем примере составная SQL-инструкция становится безопасной благодаря введению *атомарности*.

```
void main {
   EXEC SQL
      BEGIN ATOMIC
         INSERT INTO students (StudentID, Fname, Lname)
              VALUES (:sid, :sfname, :slname) ;
         INSERT INTO roster (ClassID, Class, StudentID)
              VALUES (:cid, :cname, :sid) ;
         INSERT INTO receivable (StudentID, Class, Fee)
              VALUES (:sid, :cname, :cfee)
      END ;
/* Проверка кода ошибки SQLSTATE */
}
```

Добавление ключевого слова ATOMIC после слова BEGIN гарантирует либо выполнение всей инструкции полностью, либо — в случае возникновения ошибки — откат к состоянию базы данных, в котором она находилась до начала выполнения этой инструкции. (Об атомарности мы говорили в главе 15.)

Всегда можно узнать, была ли составная инструкция выполнена успешно. Более подробно об этом речь пойдет в разделе "Состояния".

Переменные

Такие языки программирования высокого уровня, как C и Java, всегда поддерживали переменные. До появления дополнения SQL/PSM переменные в SQL использовать было нельзя. *Переменная* представляет собой идентификатор, который хранит значение определенного типа данных. В составной инструкции можно объявить переменную, присвоить ей значение, а затем использовать.

После выполнения составной инструкции все переменные, объявленные в ней, уничтожаются. Таким образом, переменные в SQL являются *локальными* по отношению к составной инструкции, в которой они объявлены.

Рассмотрим следующий пример.

```
BEGIN
   DECLARE prezpay NUMERIC ;
    SELECT salary
      INTO prezpay
      FROM EMPLOYEE
     WHERE jobtitle = 'президент' ;
END;
```

Курсоры

В составной инструкции можно использовать *курсор*. Как уже говорилось в главе 20, курсоры применяются для построчной обработки данных. В составной инструкции можно объявить курсор, использовать его, а затем забыть о нем — такой курсор уничтожается сразу после завершения составной инструкции. Ниже приведен пример использования курсора.

```
BEGIN
   DECLARE ipocandidate CHARACTER(30) ;
   DECLARE cursor1 CURSOR FOR
      SELECT company
        FROM biotech ;
   OPEN CURSOR1 ;
   FETCH cursor1 INTO ipocandidate ;
   CLOSE cursor1 ;
END;
```

Состояния

Когда говорят о состоянии человека, это, как правило, означает отклонение от нормы, т.е. его состояние связывают с болезнью или травмой. Когда

с человеком все в порядке, его состоянием никто не интересуется. В сочетании со словом "состояние" обычно используют прилагательное "болезненное", "серьезное" или, что хуже всего, "критическое". Аналогичным образом программисты говорят и о состоянии инструкций SQL. Результат выполнения SQL-инструкции может быть успешным, сомнительным или ошибочным. Каждый из таких результатов соответствует определенному *состоянию*.

При выполнении каждой SQL-инструкции сервер базы данных обновляет значение переменной `SQLSTATE`, представляющей собой поле из пяти символов. Переменная `SQLSTATE` хранит информацию об успешном или неудачном завершении последней инструкции. Если операция завершилась ошибкой, то из значения этой переменной можно извлечь информацию о характере проблемы.

Первые два из пяти символов переменной `SQLSTATE` (значение класса) содержат информацию о том, выполнена ли SQL-инструкция успешно, получен ли неопределенный результат или операция завершилась ошибкой. В табл. 21.1 приведены четыре возможных класса.

Таблица 21.1. Значения класса в переменной `SQLSTATE`

Класс	Описание	Подробности
`00`	Успешное завершение	Инструкция выполнена успешно
`01`	Предупреждение	Во время выполнения инструкции случилось нечто непредвиденное, но СУБД не может точно определить, произошла ошибка или нет. Внимательно проверьте предыдущую SQL-инструкцию и убедитесь в ее корректном выполнении
`02`	Не найдено	В результате выполнения инструкции не были возвращены данные. Это может быть хорошо или плохо, в зависимости от задачи, решаемой инструкцией, — возможно, вам была нужна именно пустая таблица
Другое	Произошло исключение	Два символа кода класса и три символа кода подкласса сообщают о характере произошедшей ошибки

Обработка состояний

Ваша программа должна проверять значение переменной `SQLSTATE` после выполнения каждой SQL-инструкции. Что же можно предпринять, используя полученную информацию?

- **При получении значения класса 00 вряд ли необходимо предпринимать дополнительные действия.** Продолжайте делать то, что запланировали.
- **При получении значения класса 01 или 02 можно предпринять специальные действия.** Если вы ожидали получения значений "Предупреждение" или "Не найдено", то выполнение должно продолжиться. Если же вы не ожидали получить эти коды классов, то, возможно, вам следует перейти к выполнению специальной процедуры, обрабатывающей подобные ситуации.
- **Получение какого-либо другого кода класса указывает на то, что в программе имеются ошибки.** В этом случае следует перейти к процедуре обработки исключения. Конкретная процедура зависит от содержимого трех символов подкласса и двух символов класса ошибки переменной SQLSTATE. Если существует вероятность возникновения нескольких разных исключений, то для каждого из них следует написать собственную процедуру обработки. Различные исключения зачастую должны обрабатываться по-разному. Одни ошибки несущественны или легко поправимы; другие могут оказаться фатальными, но вы все равно должны завершить приложение.

Объявление обработчиков состояний

Обработчик состояния тоже можно поместить в составную инструкцию. При создании обработчика вначале следует объявить состояние, которое он должен обслуживать. Объявленное состояние может быть исключением или же переменной, имеющей истинное значение. В табл. 21.2 представлены возможные состояния, а также кратко описаны причины возникновения каждого из них.

Таблица 21.2. Состояния, которые можно определить в обработчике

Состояние	Описание
SQLSTATE VALUE '*xxyyy*'	Конкретное значение SQLSTATE
SQLEXCEPTION	Значение класса SQLSTATE, отличное от 00, 01 или 02
SQLWARNING	Значение класса SQLSTATE, равное 01
NOT FOUND	Значение класса SQLSTATE, равное 02

Ниже приведен пример объявления состояния.

```
BEGIN
   DECLARE constraint_violation CONDITION
       FOR SQLSTATE VALUE '23000' ;
END ;
```

Приведенный пример — не из реальной практики, поскольку как SQL-инструкция, которая может вызвать особое состояние, так и сам обработчик, вызываемый при возникновении особого состояния, тоже должны быть заключены в рамки блока `BEGIN...END`.

Действия обработчика и их результаты

Как только возникает определенное состояние, вызывается соответствующий обработчик, выполняющий определенное действие. Такое действие представляет собой простую или составную инструкцию SQL. После успешного завершения обработчика наступает так называемый *эффект обработчика*. Ниже представлен список трех возможных эффектов.

- `CONTINUE`. Выполнение продолжается сразу после инструкции, которая инициировала вызов обработчика.
- `EXIT`. Выполнение продолжается после составной инструкции, содержащей обработчик.
- `UNDO`. Отмена всех предыдущих инструкций, входящих в составную инструкцию, и продолжение выполнения после инструкции, которая содержит обработчик.

Эффект `CONTINUE` лучше всего применять в случае, если обработчик способен устранить вызвавшую его проблему. Эффект `EXIT` подходит тогда, когда обработчику не под силу устранить проблему, но при этом нет необходимости отменять изменения, внесенные составной инструкцией. Эффект `UNDO` позволяет вернуть базу данных в состояние, в котором она находилась до начала выполнения составной инструкции. 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 (:cid, :cname, :sid) ;
END ;
```

Если выполнение какой-либо из двух инструкций INSERT вызывает нарушение заданного ограничения, например, при попытке добавить запись с первичным ключом, уже имеющимся в таблице, то в переменную SQLSTATE помещается значение '23000', а в параметр constraint_violation — значение true. Обработчик этого события отменяет изменения, внесенные в таблицы обеими инструкциями INSERT (эффект UNDO). Инструкция RESIGNAL возвращает управление процедуре, которая вызвала процедуру, выполняющуюся в текущий момент.

Если обе инструкции INSERT завершились успешно, то следующей выполняется инструкция, идущая после ключевого слова END.

Ключевое слово ATOMIC является обязательным в случае эффекта UNDO. Это требование не распространяется на обработчики с эффектами CONTINUE и EXIT.

Необрабатываемые состояния

Предположим, что в предыдущем примере возникнет состояние, при котором значение SQLSTATE будет отличным от '23000'. Ваш обработчик не может обработать подобное состояние. И что тогда?

Так как в текущей процедуре такое состояние не предусмотрено, выполняется инструкция RESIGNAL. После этого состояние может быть обработано на более высоком уровне. Если же и на этом уровне состояние нельзя обработать, то оно передается на еще более высокий уровень и так далее, пока состояние не будет обработано или не возникнет сообщение об ошибке в главном приложении.


ЗАПОМНИ!

В данном примере я хотел подчеркнуть, что если во время выполнения SQL-инструкции возможно возникновение исключений, то необходимо написать обработчики для них всех. Если этого не сделать, будет намного сложнее искать источник ошибки, появление которой — лишь вопрос времени.

Присваивание

С появлением дополнения SQL/PSM пользователи SQL получили наконец возможность, которую всегда имели даже самые примитивные процедурные языки: возможность присваивания значения переменной. Инструкция присваивания имеет следующий вид:

```
SET приемник = источник ;
```

Здесь *приемник* — это имя переменной, а *источник* — выражение.

Вот несколько примеров инструкций присваивания.

```
SET vfname = 'Джон' ;
SET varea = 3.1416 * :radius * :radius ;
SET vWIMPmass = NULL ;
```

Управляющие конструкции

Основным недостатком исходного стандарта SQL-86, не позволявшим считать SQL полноценным процедурным языком, было отсутствие управляющих конструкций. До появления SQL/PSM строгую последовательность выполнения команд нельзя было нарушить без использования базового языка, такого как C или Java. Дополнение SQL/PSM вооружило SQL традиционными для других языков управляющими конструкциями, позволив тем самым решать многие задачи без многократного переключения между языками программирования.

Конструкция IF...THEN...ELSE...END IF

Основным средством передачи управления является конструкция `IF...THEN...ELSE...END IF`. Если условие, заданное после ключевого слова `IF`, истинно, выполняются инструкции, следующие после ключевого слова `THEN`; в противном случае выполняются инструкции, следующие после ключевого слова `ELSE`.

```
IF
   vfname = 'Джон'
THEN
   UPDATE students
      SET Fname = 'Джон'
    WHERE StudentID = 314159 ;
ELSE
   DELETE FROM students
    WHERE StudentID = 314159 ;
END IF
```

В приведенном примере, если переменная `vfname` содержит значение `'Джон'`, в списке студентов обновится запись с идентификатором `314159` — в поле `Fname` будет помещено значение `'Джон'`. Если же переменная `vfname` содержит какое-либо другое значение, то из таблицы `STUDENTS` запись с идентификатором `314159` будет удалена.

Блок `IF...THEN...ELSE...END IF` больше всего подходит для ситуаций, когда в зависимости от некоторого условия существуют два варианта дальнейшего развития событий. В то же время довольно часто приходится иметь дело

с бо́льшим количеством вариантов. В таких случаях лучше использовать инструкцию CASE.

Конструкция CASE . . . END CASE

Существуют две разновидности инструкции CASE: простая и с поиском. Оба варианта позволяют выполнять различные действия в зависимости от значений, которые могут иметь проверяемые условия.

Простая инструкция CASE

Простая инструкция CASE вычисляет выражение, заданное в качестве условия. В зависимости от его значения выполнение программы продолжится по одному из нескольких возможных путей. Рассмотрим следующий пример.

```
CASE vmajor
   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, :sfname, :slname) ;
   WHEN 'Philosophy'
   THEN INSERT INTO skeptics (StudentID, Fname, Lname)
            VALUES (:sid, :sfname, :slname) ;
   ELSE INSERT INTO undeclared (StudentID, Fname, Lname)
            VALUES (:sid, :sfname, :slname) ;
END CASE
```

Инструкции предложения ELSE выполняются, если значение переменной vmajor не попадает ни в одну из категорий, заданных предложениями WHEN.

Предложение ELSE не является обязательным. Но если оно отсутствует и условие инструкции CASE не будет обработано ни одним из предложений THEN, SQL сгенерирует исключение.

Инструкция CASE с поиском

Инструкция CASE с поиском аналогична простой инструкции CASE, за исключением того, что она вычисляет несколько условий, а не одно.

```
CASE
   WHEN vmajor
      IN ('Computer Science', 'Electrical Engineering')
      THEN INSERT INTO geeks (StudentID, Fname, Lname)
               VALUES (:sid, :sfname, :slname) ;
   WHEN vclub
      IN ('Amateur Radio', 'Rocket', 'Computer')
      THEN INSERT INTO geeks (StudentID, Fname, Lname)
               VALUES (:sid, :sfname, :slname) ;
```

```
   WHEN vmajor
      IN ('Sports Medicine', 'Physical Education')
      THEN INSERT INTO jocks (StudentID, Fname, Lname)
                VALUES (:sid, :sfname, :slname) ;
   ELSE
      INSERT INTO skeptics (StudentID, Fname, Lname)
           VALUES (:sid, :sfname, :slname) ;
END CASE
```

Мы избегаем исключений, помещая фамилии всех студентов, которые не являются компьютерщиками (таблица `geeks`) или спортсменами (таблица `jocks`), в таблицу скептиков (`SKEPTICS`). Конечно, такой сценарий не во весх случаях является корректным, но у нас всегда есть возможность добавить в инструкцию `CASE` еще несколько предложений `WHEN`.

Цикл LOOP...END LOOP

Цикл `LOOP` позволяет многократно выполнить некоторую последовательность инструкций SQL. После выполнения последней инструкции SQL, находящейся в блоке `LOOP...ENDLOOP`, управление передается первой инструкции цикла, и вся последовательность выполняется еще раз. Цикл `LOOP...ENDLOOP` выглядит так.

```
SET vcount = 0 ;
LOOP
   SET vcount = vcount + 1 ;
   INSERT INTO asteroid (AsteroidID)
        VALUES (vcount) ;
END LOOP
```

Приведенный фрагмент кода создает заготовку для таблицы `ASTEROID` с уникальными идентификаторами. Остальные подробности об астероидах вы сможете вносить по мере их открытия в обсерватории.

Следует отметить, что данный фрагмент кода имеет один существенный недостаток: он содержит бесконечный цикл. В нем нет условия завершения, следовательно, добавление строк в таблицу `ASTEROID` будет продолжаться до тех пор, пока СУБД не исчерпает ресурсы памяти. После этого система в лучшем случае сгенерирует исключение, а в худшем — полностью зависнет.

На практике инструкция `LOOP` должна иметь возможность выхода из цикла до возникновения исключения. Для этого предназначена инструкция `LEAVE`.

Инструкция LEAVE

Как только выполнение программы доходит до инструкции `LEAVE` с меткой, она передает управление инструкции, следующей за данной меткой.

```
AsteroidPreload:
SET vcount = 0 ;
LOOP
   SET vcount = vcount + 1 ;
   IF vcount > 10000
      THEN
         LEAVE AsteroidPreload ;
   END IF ;
   INSERT INTO asteroid (AsteroidID)
        VALUES (vcount) ;
END LOOP AsteroidPreload
```

Приведенный выше код создает 10 тысяч последовательно пронумерованных записей в таблице ASTEROID, после чего цикл завершается.

Цикл WHILE...DO...END WHILE

Цикл WHILE предлагает альтернативный способ многократного выполнения последовательности инструкций SQL. Если условие инструкции WHILE истинно, то цикл продолжает выполняться; когда оно становится ложным, выполнение цикла прекращается. Рассмотрим следующий пример.

```
AsteroidPreload2:
SET vcount = 0 ;
WHILE
   vcount< 10000 DO
      SET vcount = vcount + 1 ;
      INSERT INTO asteroid (AsteroidID)
           VALUES (vcount) ;
END WHILE AsteroidPreload2
```

Результат работы этого кода точно такой же, как и в предыдущем примере. Это еще раз подтверждает то, что в SQL существует множество способов решения поставленной задачи. В зависимости от обстоятельств можно выбрать любой из них — главное, чтобы используемая СУБД позволяла это.

Цикл REPEAT...UNTIL...END REPEAT

Цикл REPEAT напоминает цикл WHILE. Разница между ними заключается в том, что условие здесь проверяется не до, а после выполнения инструкций цикла. Рассмотрим следующий пример.

```
AsteroidPreload3:
SET vcount = 0 ;
REPEAT
   SET vcount = vcount + 1 ;
   INSERT INTO asteroid (AsteroidID)
```

```
        VALUES (vcount) ;
   UNTIL X = 10000
END REPEAT AsteroidPreload3
```

Во всех трех приведенных выше примерах одна и та же операция выполнялась тремя разными способами (с помощью циклов LOOP, WHILE и REPEAT). В то же время на практике часто встречаются ситуации, когда один из этих способов имеет явные преимущества перед другими. Поэтому вы должны знать особенности всех трех циклов и в конкретных обстоятельствах выбрать наиболее подходящий из них.

Цикл FOR...DO...END FOR

Цикл FOR в SQL объявляет и открывает курсор, извлекает строки курсора, выполняет инструкции тела цикла 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, то их можно использовать в качестве простой альтернативы курсорам, описанным в главе 20. Рассмотрим пример использования цикла FOR.

```
FOR vcount AS Curs1 CURSOR FOR
   SELECT AsteroidID FROM asteroid
DO
   UPDATE asteroid SET Description = 'железо-каменный'
    WHERE CURRENT OF Curs1 ;
END FOR
```

В данном примере происходит обновление каждой строки таблицы ASTEROID за счет ввода значения 'железо-каменный' в поле Description.

Инструкция ITERATE

Инструкция ITERATE позволяет изменять последовательность выполнения команд в SQL-циклах LOOP, WHILE, REPEAT и FOR. Если условие в инструкции цикла является истинным или не задано, то после выполнения инструкции ITERATE сразу же начинается следующая итерация цикла. Если же условие итерации оказывается ложным или неопределенным, то после выполнения инструкции ITERATE цикл прекращается.

```
AsteroidPreload4:
SET vcount = 0 ;
WHILE
   vcount< 10000 DO
      SET vcount = vcount + 1 ;
      INSERT INTO asteroid (AsteroidID)
           VALUES (vcount) ;
```

```
      ITERATE AsteroidPreload4 ;
      SET vpreload = 'DONE' ;
END WHILE AsteroidPreload4
```

В этом примере инструкция ITERATE передает управление в начало цикла WHILE до тех пор, пока переменная vcount не станет равной значению 9999. На следующей итерации переменная vcount увеличится до 10 000, выполнится инструкция INSERT, после чего инструкция ITERATE прекратит цикл и переменной vpreload будет присвоено значение 'DONE', а выполнение кода продолжится с инструкции, следующей за циклом.

Хранимые процедуры

Хранимые процедуры находятся на сервере баз данных, а не на компьютере пользователя, как это было до появления дополнения SQL/PSM. Хранимая процедура должна быть определена, после чего ее можно вызвать с помощью инструкции CALL. Хранение процедуры на сервере приводит к уменьшению сетевого трафика, повышая, таким образом, производительность. Инструкция CALL является единственным сообщением, передаваемым от пользователя на сервер. Рассмотрим пример создания хранимой процедуры.

```
EXEC SQL
   CREATE PROCEDURE ChessMatchScore
      ( IN score CHAR (3),
        OUT result CHAR (10) )
   BEGIN ATOMIC
      CASE score
         WHEN '1-0' THEN
            SET result = 'Белые выиграли' ;
         WHEN '0-1' THEN
            SET result = 'Черные выиграли' ;
         ELSE
            SET result = 'Ничья' ;
      END CASE
   END ;
```

После создания хранимую процедуру можно вызвать с помощью такой инструкции CALL.

```
CALL ChessMatchScore ('1-0', :Outcome) ;
```

Первый аргумент является входным параметром, который передается процедуре ChessMatchScore. Второй аргумент — это встроенная переменная, хранящая значение, которое присваивается выходному параметру. Процедура ChessMatchScore использует этот выходной параметр, чтобы вернуть результат

вызывающей программе. В приведенном примере она возвращает значение 'Белые выиграли'.

Стандарт SQL:2011 немного расширил возможности хранимых процедур. Во-первых, были введены *именованные аргументы*. Вот как выглядит эквивалент предыдущего вызова с использованием именованных аргументов:

```
CALL ChessMatchScore (result => :Outcome, score =>'1-0');
```

Поскольку аргументы являются именованными, их можно записывать в любом порядке без риска перепутать.

Во-вторых, в SQL:2011 для входных аргументов была добавлена возможность задания *значений по умолчанию*. Если вы воспользуетесь такой возможностью, то вам не нужно будет указывать входное значение в инструкции CALL, поскольку будет действовать значение по умолчанию.

Рассмотрим пример.

```
EXEC SQL
   CREATE PROCEDURE ChessMatchScore
      ( IN score CHAR (3) DEFAULT '1-0',
        OUT result CHAR (10) )
   BEGIN ATOMIC
      CASE score
         WHEN '1-0' THEN
            SET result = 'Белые выиграли' ;
         WHEN '0-1' THEN
            SET result = 'Черные выиграли' ;
         ELSE
            SET result = 'Ничья' ;
      END CASE
   END ;
```

Теперь можно вызвать эту процедуру с учетом применения значения по умолчанию.

```
CALL ChessMatchScore (:Outcome) ;
```

Безусловно, это имеет смысл делать только в том случае, если значение по умолчанию действительно является тем значением, которое вы собирались передать процедуре.

Хранимые функции

Хранимые функции во многих аспектах подобны хранимым процедурам, но отличаются способом вызова. Хранимые процедуры вызываются с помощью

инструкции CALL, а вызов хранимой функции может заменить собой аргумент инструкции SQL. Ниже приведен пример определения функции.

```
CREATE FUNCTION PurchaseHistory (CustID)
   RETURNS CHAR VARYING (200)

   BEGIN
      DECLARE purch CHAR VARYING (200)
         DEFAULT '' ;
      FOR x AS SELECT *
                  FROM transactions t
                 WHERE t.customerID = CustID
      DO
         IF purch <> ''
            THEN SET purch = purch || ', ' ;
         END IF ;
         SET purch = purch || t.description ;
      END FOR
      RETURN purch ;
   END ;
```

Функция, заданная подобным образом, создает разделенный запятыми список всех покупок, произведенных клиентом, идентификатор которого извлечен из таблицы TRANSACTIONS. Следующая инструкция UPDATE, содержащая вызов функции PurchaseHistory, заносит в таблицу CUSTOMER информацию обо всех покупках, сделанных клиентом с идентификатором 314259.

```
SET customerID = 314259 ;
UPDATE customer
   SET history = PurchaseHistory (customerID)
 WHERE customerID = 314259 ;
```

Полномочия

В главе 14 рассматривались различные полномочия, предоставляемые пользователям. Владелец базы данных может предоставить другим пользователям следующие полномочия:

- на удаление записей из таблицы (DELETE);
- на вставку записей в таблицу (INSERT);
- на обновление записей в таблице (UPDATE);
- на создание таблицы, которая ссылается на другую таблицу (REFERENCES);
- на использование домена (USAGE).

Стандарт SQL/PSM добавил к уже существующим еще один вид полномочий — полномочия на выполнение (EXECUTE). Ниже приведены два примера.

```
GRANT EXECUTE on ChessMatchScore to TournamentDirector ;
GRANT EXECUTE on PurchaseHistory to SalesManager ;
```

Эти инструкции предоставляют возможность организатору шахматного турнира (TournamentDirector) выполнить процедуру ChessMatchScore, а менеджеру по продажам (SalesManager) — выполнить функцию PurchaseHistory. Пользователи, которые не имеют полномочий на выполнение подпрограммы (EXECUTE), не смогут использовать ее.

Хранимые модули

Хранимые модули могут содержать множество подпрограмм, т.е. процедур и/или функций SQL. Каждый пользователь с полномочиями на выполнение (EXECUTE) в отношении некоторого 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 score CHAR (3),
        OUT result CHAR (10) )
   BEGIN ATOMIC
      CASE result
         WHEN '1-0' THEN
            SET result = 'Белые выиграли' ;
         WHEN '0-1' THEN
            SET result = 'Черные выиграли' ;
         ELSE
            SET result = 'Ничья' ;
      END CASE
   END ;
   FUNCTION PurchaseHistory (CustID)
   RETURNS CHAR VARYING (200)
   BEGIN
      DECLARE purch CHAR VARYING (200)
         DEFAULT '' ;
      FOR x AS SELECT *
                  FROM transactions t
                 WHERE t.customerID = CustID
      DO
         IF purch <> ''
            THEN SET purch = purch || ', ' ;
```

```
        END IF ;
        SET purch = purch || t.description ;
      END FOR
      RETURN purch ;
   END ;
END MODULE ;
```

Подпрограммы данного модуля (процедура и функция) никак не связаны между собой. В общем случае подпрограммы можно разнести по разным модулям или собрать в один модуль, независимо от того, есть у них что-то общее или нет.

Глава 22

Обработка ошибок

В ЭТОЙ ГЛАВЕ...

- Сигнализация об ошибке
- Интерпретация кодов ошибок
- Определение природы ошибки
- Определение места возникновения ошибки

Не правда ли, было бы замечательно, если бы любое написанное вами приложение всегда работало без сбоев? Еще бы! А если при этом еще и выиграть 300 млн долларов в лотерею, то жизнь стала бы раем! К сожалению, оба этих события равновероятны. Те или иные ошибки случаются неизбежно, поэтому полезно знать их причины. В SQL механизмом, передающим информацию об ошибке, является *параметр состояния* (или переменная базового языка) `SQLSTATE`. Основываясь на содержимом переменной `SQLSTATE`, можно выполнить те или иные действия, способные устранить ошибку.

Как только возникает некоторая ситуация, например, когда значение переменной `SQLSTATE` становится ненулевым, директива `WHENEVER` позволяет выполнить заранее заготовленное действие. Кроме того, в области диагностики можно найти подробную информацию о только что выполненной инструкции SQL. В этой главе вы узнаете о возможностях обработки ошибок.

Переменная SQLSTATE

Переменная `SQLSTATE` позволяет выявить множество аномальных ситуаций. Она 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 формально

разделена на две группы: двухсимвольный код класса и трехсимвольный код подкласса. Код класса означает состояние после выполнения инструкции SQL: успешное завершение или с ошибкой (один из основных типов ошибок). Код подкласса предоставляет дополнительную информацию о выполнении конкретной инструкции. Структура переменной SQLSTATE показана на рис. 22.1.

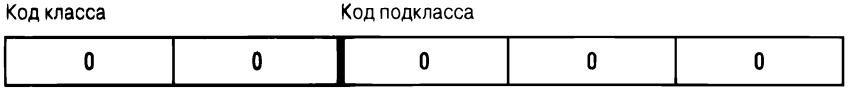


Рис. 22.1. Структура переменной SQLSTATE (значение 00000 говорит об успешном выполнении инструкции SQL)

В стандарте SQL определены все коды классов с буквами от A до H или с цифрами от 0 до 4. Во всех СУБД эти коды имеют одно и то же значение. В то же время коды классов с буквами от I до Z или с цифрами от 5 до 9 определяются конкретными реализациями СУБД. Дело в том, что спецификация SQL не может предусмотреть все ситуации, которые могут произойти в каждой конкретной СУБД. Откровенно говоря, разработчикам нужно реже использовать нестандартные коды классов — это позволит избежать многих проблем переноса приложений из одной СУБД в другую. Лучше вообще обойтись стандартными кодами, а нестандартные коды задействовать только в самых нетривиальных случаях.

Если в переменной SQLSTATE код класса равен 00, значит, инструкция завершилась успешно. Код класса, равный 01, означает, что хотя инструкция и завершилась успешно, она сгенерировала предупреждение. Если запрос не вернул данные, то значение кода класса становится равным 02. Любое другое значение кода класса, содержащееся в переменной SQLSTATE, означает, что выполнение инструкции завершилось с ошибкой.

Поскольку после каждой операции переменная SQLSTATE обновляется, то и проверять ее можно после выполнения каждой инструкции. Если значение переменной SQLSTATE равно 00000 (что соответствует успешному завершению инструкции), можно приступать к выполнению следующей запланированной операции. Если же в ней находится другое значение, то лучше отклониться от нормальной последовательности выполнения кода, чтобы обработать возникшую ситуацию. Какое из нескольких возможных действий следует выполнить, зависит от содержащихся в переменной SQLSTATE значений кодов класса и подкласса.

Чтобы переменную SQLSTATE можно было использовать в программе, написанной на модульном языке (см. главу 16), ссылку на эту переменную нужно поместить в определение процедуры, как показано в следующем примере.

```
PROCEDURE NUTRIENT
   (SQLSTATE, :foodname CHAR (20), :calories SMALLINT,
      :protein DECIMAL (5, 1), :fat DECIMAL (5, 1),
      :carbo DECIMAL (5, 1))
INSERT INTO FOODS
        (FoodName, Calories, Protein, Fat, Carbohydrate)
     VALUES
        (:foodname, :calories, :protein, :fat, :carbo) ;
```

В нужном месте программы, написанной на процедурном языке, можно присвоить параметрам определенные значения (возможно, запросив их ввод пользователем), а затем вызвать саму процедуру. Синтаксис этой операции в разных языках разный, но выглядит примерно так.

```
foodname = "Вареная окра" ;
calories = 29 ;
protein = 2.0 ;
fat = 0.3 ;
carbo = 6.0 ;
NUTRIENT(state, foodname, calories, protein, fat, carbo) ;
```

Значение SQLSTATE возвращается в переменной state. Ваша программа может проверять эту переменную и в зависимости от ее содержимого выполнять те или иные действия.

Директива WHENEVER

Вы спросите: "Зачем знать о том, что инструкция SQL не выполнилась успешно, если с этим уже ничего не поделаешь?" Дело здесь вот в чем. Если произошла ошибка, то нельзя, чтобы приложение продолжило свою работу так, будто ничего не случилось. Нужно иметь возможность узнать об ошибке и затем что-то предпринять, чтобы исправить ее. А если исправить ее невозможно, то по крайней мере нужно сообщить об ошибке пользователю и корректно завершить работу программу.

В SQL для обработки исключений предусмотрен специальный механизм в виде директивы WHENEVER. Фактически она является объявлением, поэтому ее помещают в разделе SQL-объявлений приложения перед выполняемым SQL-кодом. Эта директива имеет следующий синтаксис:

```
WHENEVER состояние действие;
```

Элемент *состояние* может быть выражен одним из двух вариантов: SQLERROR (ошибка SQL) или NOT FOUND (не найден), а элемент *действие* — вариантом CONTINUE (продолжать) или GOTO *адрес*

(перейти по адресу). Если код класса в переменной SQLSTATE не равен 00, 01 или 02, то возникает состояние SQLERROR. А если переменная SQLSTATE равна 02000, то возникает состояние NOT FOUND.

Если выбрано действие CONTINUE, то код выполняется по обычному сценарию. Если же вместо CONTINUE задано GOTO *адрес* (или GO TO *адрес*), управление передается по указанному адресу в программе. По адресу ветвления программы можно поместить условное выражение, которое проверяет значение переменной SQLSTATE и в зависимости от результата проверки обеспечивает выполнение различных действий. Вот несколько примеров такого сценария:

```
WHENEVER SQLERROR GO TO перехват_ошибок ;
```

или

```
WHENEVER NOT FOUND CONTINUE ;
```

Здесь GO TO — макрокоманда. СУБД (точнее, интерпретатор встроенного языка) вставляет после каждой инструкции EXEC SQL следующую проверку.

```
IF SQLSTATE <> '00000'
   AND SQLSTATE <> '00001'
   AND SQLSTATE <> '00002'
THEN GOTO перехват_ошибок ;
```

Вариант CONTINUE означает отсутствие действий, т.е. игнорирование ошибки.

Области диагностики

Несмотря на то что переменная SQLSTATE может предоставить некоторую информацию о причинах неудачного завершения инструкции, эта информация все же неполная. Поэтому стандарт SQL, кроме всего прочего, позволяет получать дополнительную информацию о состоянии и хранить ее в области диагностики.

Управление множеством областей диагностики происходит по принципу "последним поступил — первым обслужен" (LIFO). Это значит, что информация о последней ошибке хранится на вершине стека, а глубже находится информация о предыдущих ошибках. Дополнительная информация о состоянии в области диагностики может быть особенно полезной тогда, когда при выполнении одной инструкции SQL, помимо ошибки, было сгенерировано несколько предупреждений. Переменная SQLSTATE сообщает только об ошибке, а область диагностики может хранить данные 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 последней выполнявшейся инструкции SQL.
- **Информационная часть.** В ней содержится подробная информация о каждом коде (ошибка, предупреждение или успешное выполнение), который был сгенерирован в результате выполнения инструкции.

Заголовок области диагностики

В главе 15 мы рассмотрели инструкцию `SET TRANSACTION`. В этой инструкции предложением `DIAGNOSTICS SIZE` можно задать количество областей диагностики, выделяемых для хранения информации о состоянии. Если в инструкции `SET TRANSACTION` не содержится предложение `DIAGNOSTICS SIZE`, то СУБД выделит информационные области в количестве, установленном по умолчанию.

Заголовок области состоит из нескольких элементов, которые перечислены в табл. 22.1.

Таблица 22.1. Заголовок области диагностики

Поле	Тип данных
NUMBER	Точный числовой без дробной части
ROW_COUNT	Точный числовой без дробной части
COMMAND_FUNCTION	VARCHAR (максимальный размер определяется реализацией)
COMMAND_FUNCTION_CODE	Точный числовой без дробной части
DYNAMIC_FUNCTION	VARCHAR (максимальный размер определяется реализацией)
DYNAMIC_FUNCTION_CODE	Точный числовой без дробной части
MORE	Точный числовой без дробной части
TRANSACTIONS_COMMITED	Точный числовой без дробной части
TRANSACTIONS_ROLLED_BACK	Точный числовой без дробной части
TRANSACTIONS_ACTIVE	Точный числовой без дробной части

Ниже эти элементы описаны более детально.

- В поле NUMBER хранится количество областей, заполненных диагностической информацией о текущем исключении.
- В поле ROW_COUNT содержится количество обработанных строк, если предыдущей SQL-инструкцией была INSERT, UPDATE или DELETE.
- Поле COMMAND_FUNCTION описывает только что выполненную инструкцию SQL.
- Поле COMMAND_FUNCTION_CODE содержит код только что выполненной инструкции SQL. Каждая функция имеет соответствующий числовой код.
- Поле DYNAMIC_FUNCTION содержит динамическую инструкцию SQL.
- Поле DYNAMIC_FUNCTION_CODE содержит числовой код, соответствующий динамической инструкции SQL.
- 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' говорит о том, что в области диагностики представлены все созданные записи состояния. Число записей можно увеличить в инструкции SET TRANSACTION. Эта возможность зависит от конкретной СУБД.
- Поле TRANSACTIONS_COMMITED содержит количество успешно завершенных транзакций.
- Поле TRANSACTIONS_ROLLED_BACK содержит количество транзакций, которые были отменены.
- Поле TRANSACTIONS_ACTIVE содержит значение '1', если транзакция в настоящее время активна, и '0' в противном случае. Транзакция считается активной, если открыт курсор или СУБД находится в ожидании отсроченного параметра.

Информационная часть области диагностики

В информационных областях хранятся данные о каждой отдельной ошибке, каждом предупреждении или состоянии успешного завершения. Каждая информационная область состоит из 28 элементов, которые описаны в табл. 22.2.

В элементе CONDITION_NUMBER содержится порядковый номер информационной области. Если инструкция создает пять элементов состояния, которые заполняют пять информационных областей, то значение CONDITION_NUMBER для пятой области будет равно пяти. Чтобы получить доступ к конкретной информационной области, используйте инструкцию GET DIAGNOSTICS вместе с соответствующим значением поля CONDITION_NUMBER (об инструкции GET

DIAGNOSTICS рассказывается далее). А в элементе RETURNED_SQLSTATE находится значение переменной SQLSTATE, соответствующее данным этой информационной области.

Таблица 22.2. Информационная часть области диагностики

Элемент	Тип данных
CONDITION_NUMBER (номер состояния)	Точный числовой без дробной части
RETURNED_SQLSTATE (значение SQLSTATE)	CHAR (6)
MESSAGE_TEXT (текст сообщения)	VARCHAR (максимальный размер определяется реализацией)
MESSAGE_LENGTH (длина сообщения)	Точный числовой без дробной части
MESSAGE_OCTET_LENGTH (длина сообщения в октетах)	Точный числовой без дробной части
CLASS_ORIGIN (источник класса)	VARCHAR (максимальный размер определяется реализацией)
SUBCLASS_ORIGIN (источник подкласса)	VARCHAR (максимальный размер определяется реализацией)
CONNECTION_NAME (имя соединения)	VARCHAR (максимальный размер определяется реализацией)
SERVER_NAME (имя сервера)	VARCHAR (максимальный размер определяется реализацией)
CONSTRAINT_CATALOG (каталог ограничения)	VARCHAR (максимальный размер определяется реализацией)
CONSTRAINT_SCHEMA (схема ограничения)	VARCHAR (максимальный размер определяется реализацией)
CONSTRAINT_NAME (имя ограничения)	VARCHAR (максимальный размер определяется реализацией)
CATALOG_NAME (имя каталога)	VARCHAR (максимальный размер определяется реализацией)
SCHEMA_NAME (имя схемы)	VARCHAR (максимальный размер определяется реализацией)
TABLE_NAME (имя таблицы)	VARCHAR (максимальный размер определяется реализацией)

Окончание табл. 22.2

Элемент	Тип данных
COLUMN_NAME (имя столбца)	VARCHAR (максимальный размер определяется реализацией)
CURSOR_NAME (имя курсора)	VARCHAR (максимальный размер определяется реализацией)
CONDITION_IDENTIFIER (идентификатор состояния)	VARCHAR (максимальный размер определяется реализацией)
PARAMETR_NAME (имя параметра)	VARCHAR (максимальный размер определяется реализацией)
PARAMETER_ORDINAL_POSITION (порядковый номер параметра)	Точный числовой без дробной части
PARAMETER_MODE (режим параметра)	Точный числовой без дробной части
ROUTINE_CATALOG (каталог подпрограммы)	VARCHAR (максимальный размер определяется реализацией)
ROUTINE_SCHEMA (схема подпрограммы)	VARCHAR (максимальный размер определяется реализацией)
ROUTINE_NAME (имя подпрограммы)	VARCHAR (максимальный размер определяется реализацией)
SPECIFIC_NAME (специфическое имя)	VARCHAR (максимальный размер определяется реализацией)
TRIGGER_CATALOG (каталог триггера)	VARCHAR (максимальный размер определяется реализацией)
TRIGGER_SCHEMA (схема триггера)	VARCHAR (максимальный размер определяется реализацией)
TRIGGER_NAME (имя триггера)	VARCHAR (максимальный размер определяется реализацией)

Элемент CLASS_ORIGIN сообщает, откуда взято значение для кода класса, возвращенное в переменной SQLSTATE. Если значение определено стандартом SQL, то элемент CLASS_ORIGIN равен 'ISO 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 переменной SQLSTATE.

Значение, находящееся в элементе CLASS_ORIGIN, является весьма важным. Например, значение SQLSTATE, равное '22012', относится к стандартным значениям этой переменной, т.е. означает одно и то же во всех реализациях SQL. Но если значение SQLSTATE равно '22500', то первые два символа находятся в стандартном диапазоне и указывают на исключение, причиной которого является отсутствие данных, а последние три символа уже находятся в диапазоне, определяемом реализацией. Ну а если значение SQLSTATE равно '90001', то все это значение находится в диапазоне, определяемом реализацией. Одни и те же значения SQLSTATE, находящиеся в таком диапазоне, могут в разных СУБД иметь совершенно разные трактовки.

Вы спросите, где найти описание значений '22500' или '90001'? Для этого нужно обратиться к документации конкретной СУБД. А какой именно СУБД, ведь с помощью инструкции CONNECT можно подключаться к разным СУБД? Чтобы узнать, какая из них является источником ошибки, посмотрите на содержимое элементов CLASS_ORIGIN и SUBCLASS_ORIGIN. В них находятся значения, по которым 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 связана с нарушением ограничения, то это ограничение можно определить с помощью элементов CONSTRAINT_CATALOG, CONSTRAINT_SCHEMA и CONSTRAINT_NAME.

Пример нарушения ограничения

Из всей информации, выдаваемой инструкцией GET DIAGNOSTICS, самой важной является информация о нарушении ограничения. Рассмотрим в качестве примера таблицу EMPLOYEE (сотрудники), созданную следующей инструкцией.

```
CREATE TABLE EMPLOYEE
   (ID        CHAR(5)      CONSTRAINT EmpPK PRIMARY KEY,
    Salary    DEC(8, 2)    CONSTRAINT EmpSal CHECK Salary > 0,
    Dept      CHAR(5)      CONSTRAINT EmpDept,
       REFERENCES DEPARTMENT) ;
```

Также рассмотрим таблицу DEPARTMENT (отдел), созданную такой инструкцией.

```
CREATE TABLE DEPARTMENT
   (DeptNo   CHAR(5),
    Budget   DEC(12, 2)   CONSTRAINT DeptBudget
```

```
    CHECK(Budget >= SELECT SUM(Salary)
                    FROM EMPLOYEE
                   WHERE EMPLOYEE.Dept=DEPARTMENT.DeptNo),
...) ;
```

Теперь рассмотрим следующую инструкцию INSERT:

```
INSERT INTO EMPLOYEE VALUES(:ID_VAR, :SAL_VAR, :DEPT_VAR) ;
```

Предположим, вы получили значение переменной SQLSTATE, равное '23000'. Заглянув в документацию по своей СУБД, вы увидите, что этому значению соответствует описание "нарушение ограничения целостности". Это означает, что имеет место одна из следующих ситуаций.

- **Значение ID_VAR дублирует уже существующее значение идентификатора.**

  Нарушено ограничение PRIMARY KEY.

- **Значение SAL_VAR отрицательное.**

  Иначе говоря, нарушено ограничение CHECK для столбца Salary.

- **Значение DEPT_VAR не является действительным ключом, соответствующим какой-либо из строк таблицы DEPARTMENT.**

  Так что нарушено ограничение REFERENCES для столбца Dept.

- **Значение SAL_VAR настолько большое, что сумма окладов сотрудников отдела, для которого вводятся новые данные, превышает значение BUDGET (бюджет).**

  На этот раз нарушено ограничение CHECK для столбца BUDGET таблицы DEPARTMENT. (Вспомните, что при изменении базы данных должны проверяться все ограничения, на которые это изменение может повлиять, а не только содержащиеся в указанной таблице.)

Обычно, чтобы узнать причины невыполнения инструкции INSERT, приходится проводить большое количество проверок. Но на этот раз всю необходимую информацию можно узнать с помощью инструкции GET DIAGNOSTICS.

```
DECLARE ConstNameVar CHAR(18) ;
GET DIAGNOSTICS EXCEPTION 1
   ConstNameVar = CONSTRAINT_NAME ;
```

И если значение переменной SQLSTATE равно '23000', то инструкция GET DIAGNOSTICS присвоит переменной ConstNameVar одно из следующих значений: 'EmpPK', 'EmpSal', 'EmpDept' или 'DeptBudget'. На практике, для того чтобы однозначно идентифицировать ограничение CONSTRAINT_NAME, имеет смысл получить значения элементов CONSTRAINT_SCHEMA и CONSTRAINT_CATALOG.

Добавление новых ограничений в уже созданную таблицу

Инструкция GET DIAGNOSTICS полезна для определения того, какие именно ограничения были нарушены при изменении исходных таблиц. Особенно важное значение она приобретает при использовании инструкции ALTER TABLE для добавления в таблицу ограничений, которых не было на момент написания программы.

```
ALTER TABLE EMPLOYEE
   ADD CONSTRAINT SalLimit CHECK(Salary < 200000) ;
```

Теперь, когда вы вставите данные в таблицу EMPLOYEE или обновите в ней столбец Salary, используя значение, превышающее 200 000, значение параметра SQLSTATE станет равным '23000'. Запомнить значения кодов довольно сложно, поэтому лучше запрограммировать вывод информационных сообщений, например "Неправильное выполнение инструкции INSERT: нарушение ограничения SalLimit".

Интерпретация информации, возвращаемой в переменной SQLSTATE

Элементы CONNECTION_NAME и SERVER_NAME идентифицируют подключение к базе данных и сервер, с которым было установлено соединение во время выполнения инструкции SQL.

Если информация из параметра SQLSTATE относится к работе с таблицей, то эту таблицу определяют элементы CATALOG_NAME, SCHEMA_NAME и TABLE_NAME. Если возникновение ошибки как-то связано с конкретным столбцом таблицы, то его имя помещается в элемент COLUMN_NAME. Если внештатная ситуация имеет отношение к курсору, его имя будет находиться в элементе CURSOR_NAME.

Иногда СУБД для прояснения ситуации создает строку обычного текста. Такого рода информация хранится в элементе MESSAGE_TEXT. Его содержимое определяется не стандартом SQL, а конкретной реализацией. Если в элементе MESSAGE_TEXT хранится сообщение, то его длина в символах записывается в элемент MESSAGE_LENGTH, а длина в октетах — в элемент MESSAGE_OCTET_LENGTH. У сообщения, состоящего из обычных символов ASCII, значения MESSAGE_LENGTH и MESSAGE_OCTET_LENGTH равны между собой. А если сообщение составлено на китайском, японском или любом другом языке, в котором для описания символа требуется более одного октета, то значения MESSAGE_LENGTH и MESSAGE_OCTET_LENGTH будут разными.

Чтобы получить диагностическую информацию из области заголовка, используйте следующую инструкцию.

```
GET DIAGNOSTICS переменная_1 = элемент_1[, переменная_2 = елемент_2]... ;
```

Здесь *переменная_n* — 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, используйте следующий синтаксис.

```
GET DIAGNOSTICS EXCEPTION номер_состояния
   переменная_1 = элемент_1 [, переменная_2 = элемент_2]... ;
```

Здесь *переменная_n* — параметр или базовая переменная; *элемент_n* — любое из 28 ключевых слов, используемых в качестве элементов информационной области (см. табл. 21.2). И наконец, *номер_состояния* — это значение элемента CONDITION_NUMBER информационной области.

Обработка исключений

Если переменная SQLSTATE не равна 00000, 00001 или 00002, значит, возникла внештатная ситуация, которую желательно обработать одним из следующих способов:

- вернуть управление родительской процедуре, из которой была вызвана подпрограмма — источник исключения;
- использовать предложение WHENEVER для перехода к нужной процедуре обработки исключения или выполнения какого-либо другого действия;
- обработать ситуацию прямо на месте, используя для этого составную инструкцию SQL (см. главу 21).

  Составной называется такая инструкция, которая состоит из нескольких простых инструкций SQL, находящихся между ключевыми словами BEGIN и END.

Ниже приведен пример составной инструкции, выполняющей обработку исключения.

```
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 'Количество калорий лежит за пределами
                 допустимого диапазона' ;
    WHEN OTHERS THEN
        RESIGNAL ;
END
```

С помощью одного или нескольких объявлений (инструкций DECLARE) можно задать имена для всех возможных значений переменной SQLSTATE. Одной из инструкций, которые могут вызвать внештатную ситуацию, является INSERT. Если значение переменной :cal (количество калорий) превысит максимальное значение, заданное для элемента данных типа SMALLINT, то переменной SQLSTATE будет присвоено значение '73003'. Сигнал о возникновении исключительной ситуации подает инструкция SIGNAL, которая очищает область диагностики, а также присваивает полю RETURNED_SQLSTATE этой области значение переменной SQLSTATE. Если исключения не было, то выполняется обычная последовательность инструкций, роль которой в данном примере играет инструкция MESSAGE 'Обрабатывается новое значение количества калорий'. Если исключение все же имело место, то обычная последовательность действий опускается и выполняется инструкция EXCEPTION.

При возникновении исключения ValueOutOfRange (значение вне допустимого диапазона) выполняется последовательность инструкций, представленных в данном примере инструкцией MESSAGE 'Количество калорий лежит за пределами допустимого диапазона'. Если возникает какое-либо другое исключение (не ValueOutOfRange), то выполняется инструкция RESIGNAL.

Инструкция RESIGNAL просто возвращает управление родительской процедуре. Эта процедура может содержать дополнительный код, который позволит обработать внештатные ситуации, не связанные с выходом за пределы диапазона.

Глава 23

Триггеры

В ЭТОЙ ГЛАВЕ...

- Создание триггеров
- Условия для срабатывания триггеров
- Выполнение триггера
- Запуск нескольких триггеров

В приложении, работающем с базой данных, могут возникать ситуации, когда при определенном действии должно выполняться другое действие или даже последовательность действий. Другими словами, одно действие запускает последующие действия. Для обеспечения этой возможности SQL предоставляет механизм триггеров.

Триггер — это действие или событие, которое запускает другое событие. В SQL триггер используется, когда одна инструкция инициирует выполнение другой инструкции.

Область применения триггеров

Срабатывание триггера полезно во многих ситуациях. Одним из примеров является функция регистрации. Определенные действия, являющиеся критически важными для целостности базы данных (такие, как вставка, редактирование или удаление строки таблицы), могут инициировать создание записи в журнале, чтобы зарегистрировать это действие. Записи в журнале могут содержать информацию не только о предпринятом действии, но и о том, кто его предпринял и когда.

Триггеры можно также использовать для поддержания целостности базы данных. В приложении ввода заказов, например, заказ определенного товара может запустить на выполнение инструкцию, которая изменит состояние этого товара в инвентарной таблице с "доступно" на "зарезервировано". Точно так же удаление строки из таблицы заказов может запустить инструкцию, которая изменит состояние соответствующего товара с "зарезервировано" на "доступно".

В действительности триггеры обеспечивают даже большую гибкость, чем в приведенных выше примерах. Инициируемый триггером элемент не обязан быть инструкцией SQL. Это может быть процедура базового языка, которая выполняет некую операцию в системе, такую как остановка конвейера или приказ роботу достать пиво из холодильника.

Создание триггера

Для создания триггера предназначена инструкция `CREATE TRIGGER`. После создания триггер находится в состоянии ожидания события, в результате которого он сработает. Когда такое (инициирующее) событие происходит, триггер срабатывает.

Синтаксис инструкции `CREATE TRIGGER` сложен, но его можно разбить на понятные части. Сначала рассмотрим общий формат.

```
CREATE TRIGGER имя_триггера
   время_действия_триггера событие_триггера
   ON имя_таблицы
   [REFERENCING список_псевдонимов_старых_или_новых_значений]
   инициируемое_действие
```

Здесь *имя_триггера* — это уникальный идентификатор данного триггера; *время_действия_триггера* — момент времени, когда должно произойти *инициируемое_действие*: до (`BEFORE`) или после (`AFTER`) инициирующего события. Тот факт, что ответное действие может произойти *до* вызвавшего его события, может показаться немного странным, но в некоторых случаях эта возможность весьма полезна (и может быть достигнута без путешествия во времени). Поскольку СУБД заранее известно о возникновении некоторого инициирующего события, она вполне может выполнить *инициируемое_действие* прежде, чем фактически наступит инициирующее событие, если время действия триггера было определено с помощью ключевого слова `BEFORE`.

Три триггерных события могут привести к срабатыванию триггера: выполнение инструкции `INSERT`, `DELETE` или `UPDATE`. Эти три инструкции способны изменять содержимое таблицы базы данных. Таким образом, любая вставка одной или нескольких строк в таблицу, любое удаление одной или нескольких

строк либо любое обновление одного или нескольких столбцов в таблице может привести к срабатыванию триггера. Таблица, для которой определена инструкция INSERT, DELETE или UPDATE, указывается в предложении ON *имя_таблицы*.

Триггеры инструкций и строк

Элемент *инициируемое_действие* имеет следующий синтаксис.

```
[ FOR EACH { ROW | STATEMENT } ]
   WHEN <левая круглая скобка><условие отбора><правая круглая скобка>
   <запускаемая SQL-инструкция>
```

Действие триггера можно определить следующим образом.

- **Триггер строки.** Триггер сработает один раз, если встретится инструкция INSERT, DELETE или UPDATE, составляющая событие триггера.
- **Триггер инструкции.** Триггер срабатывает многократно, для каждой строки таблицы, в которой произошло событие триггера.

Квадратные скобки означают, что предложение FOR EACH не является обязательным. Несмотря на это триггер так или иначе должен сработать. Если предложение FOR EACH не определено, по умолчанию действует вариант FOR EACH STATEMENT.

Когда срабатывает триггер

Условие отбора в предложении WHEN позволяет определить обстоятельства, при которых триггер будет срабатывать. Определите предикат, и если его значение будет истинным, триггер сработает; если оно будет ложным, он не сработает. Эта возможность существенно увеличивает ценность триггеров. Можно указать, что триггер сработает только после того, как будет превышено некое пороговое значение либо когда любое другое условие окажется равным True или False.

Запускаемая SQL-инструкция

Инициируемая SQL-инструкция может быть представлена одиночной инструкцией SQL или последовательностью инструкций, выполняемых одна за другой. В первом случае это просто обычная инструкция SQL. Но для последовательности инструкций вы должны гарантировать ее атомарность, чтобы она не могла быть прервана посреди выполнения (это может привести базу данных в некорректное состояние). Для этого можно использовать блок BEGIN...END с ключевым словом ATOMIC.

```
BEGIN ATOMIC
   { SQL-инструкция_1 }
   { SQL-инструкция_2 }
...
   { SQL-инструкция_n }
END
```

Пример определения триггера

Предположим, отдел кадров компании хочет получать информацию от региональных отделений о приеме сотрудников на работу. С этой задачей может прекрасно справиться следующий триггер.

```
CREATE TRIGGER newhire
   BEFORE INSERT ON employee
   FOR EACH STATEMENT
      BEGIN ATOMIC
         CALL sendmail ('HRDirector')
         INSERT INTO logtable
            VALUES ('NEWHIRE', CURRENT_USER, CURRENT_TIMESTAMP);
      END;
```

Всякий раз, когда в таблицу EMPLOYEE вставляется новая строка, в отдел кадров отправляется электронное сообщение с подробными данными, а регистрационное имя человека, добавившего запись, и время вставки записываются в таблицу журнала, обеспечивая регистрацию событий.

Срабатывание последовательности триггеров

Вы, вероятно, заметили возможную проблему в работе триггеров. Предположим, вы создаете триггер, который применяет инструкцию SQL к таблице после выполнения некоторой предыдущей инструкции SQL. А что если эта инициируемая инструкция приведет к срабатыванию другого триггера? А тот триггер, в свою очередь, запустит третью инструкцию SQL для второй таблицы, что приведет к срабатыванию еще одного триггера, затрагивающего еще одну таблицу. Как можно избежать подобных осложнений? SQL решает проблему цепной реакции триггеров за счет *контекста выполнения триггера*.

Последовательность операций INSERT, DELETE и UPDATE может быть выполнена путем вложения контекстов, в которых они выполняются. Контекст выполнения создается при срабатывании триггера. Одновременно может быть активизирован только один контекст выполнения. В пределах этого контекста

может быть выполнена инструкция SQL, которая запустит второй триггер. Теперь работа существующего контекста выполнения приостанавливается (аналогично передаче значения в стек). Создается новый контекст выполнения, соответствующий второму триггеру, и отрабатывается его операция. Предел глубины вложения не устанавливается. По завершении операции ее контекст выполнения удаляется, а контекст выполнения более высокого уровня "извлекается из стека" и снова активизируется. Этот процесс продолжается до тех пор, пока не будут выполнены все действия и не будут удалены все контексты выполнения.

Ссылки на старые и новые значения

Часть синтаксиса инструкции `CREATE TRIGGER`, о которой еще не упоминалось, — это необязательное предложение `REFERENCING` *`список_псевдонимов_старых_или_новых_значений`*. Оно позволяет создать псевдоним или корреляционное имя, ссылающееся на значения в рабочей таблице триггера. После создания корреляционного имени для новых значений или псевдонима для нового содержимого таблицы можно ссылаться на те значения, которые появятся после операции `INSERT` или `UPDATE`. Аналогично после создания корреляционного имени для старых значений или псевдонима для старого содержимого таблицы можно ссылаться на те значения, которые существовали в таблице до выполнения операции `UPDATE` или `DELETE`.

Элемент *`список_псевдонимов_старых_или_новых_значений`* в синтаксисе инструкции `CREATE TRIGGER` может представлять собой одну или несколько следующих фраз:

```
OLD [ ROW ] [ AS ] <корреляционное имя старых значений>
```

или

```
NEW [ ROW ] [ AS ] <корреляционное имя новых значений>
```

или

```
OLD TABLE [ AS ] <псевдоним старой таблицы>
```

или

```
NEW TABLE [ AS ] <псевдоним новой таблицы>
```

Псевдонимы таблиц — это идентификаторы переходных таблиц, которые не являются постоянными и существуют только для облегчения ссылок. Вполне понятно, что сочетания `NEW ROW` и `NEW TABLE` неприменимы для триггера `DELETE`, а `OLD ROW` и `OLD TABLE` — для триггера `INSERT`. Ведь после удаления

строки или таблицы никакого нового значения нет. Аналогично сочетания OLD ROW и OLD TABLE нельзя задать для триггера INSERT, поскольку нет никаких старых значений, чтобы можно было ссылаться на них.

В триггере уровня строки можно использовать корреляционное имя старого значения для ссылки на прежнее значение в строке, модифицированной или удаленной инициирующей инструкцией SQL, поскольку эта строка существовала до того, как упомянутая инструкция модифицировала или удалила ее. Аналогично псевдоним таблицы со старыми значениями можно использовать для доступа ко всем значениям таблицы, поскольку они существовали до вызова инициирующей инструкции SQL.

Вы не сможете использовать ключевые слова OLD TABLE или NEW TABLE с триггером BEFORE. Весьма вероятно, что переходные таблицы, созданные ключевыми словами OLD TABLE и NEW TABLE, подвергнутся воздействию инициируемой инструкции SQL. Поэтому использовать ключевые слова OLD TABLE и NEW TABLE с триггером BEFORE запрещается.

Срабатывание нескольких триггеров в одной таблице

Последняя тема, которую я хочу затронуть в данной главе, — это случай, когда создается несколько триггеров, приводящих к выполнению инструкции SQL в одной и той же таблице. Все эти триггеры настроены и готовы к срабатыванию. Когда происходит инициирующее событие, какой триггер сработает первым? Эта задача решается административным способом, т.е. достаточно просто. Вначале сработает триггер, созданный первым, затем триггер, созданный вторым, и т.д. Так устраняется потенциальная двусмысленность, и выполнение операций происходит организованно, т.е. по очереди.

Великолепные десятки

В ЭТОЙ ЧАСТИ...

- Распространенные ошибки
- Быстрый поиск данных

Глава 24

Десять самых распространенных ошибок

В ЭТОЙ ГЛАВЕ...

- Уверенность в том, что клиенты знают, чего хотят
- Игнорирование масштаба проекта
- Учет только технических факторов
- Отсутствие обратной связи с пользователями
- Использование только своей любимой среды разработки и системной архитектуры
- Проектирование таблиц базы данных отдельно друг от друга
- Отказ от анализа проекта, бета-тестирования и создания документации

Раз уж вы читаете эту книгу, то наверняка интересуетесь созданием реляционных баз данных. Давайте посмотрим правде в глаза: никто не изучает SQL исключительно ради удовольствия. Этот язык используется для создания приложений, работающих с базами данных, но прежде чем создавать приложение, нужно иметь саму базу данных. К сожалению, многие проекты терпят неудачу еще до написания первой строки кода. Если в проект базы данных заложены неверные принципы, то, как бы хорошо ни было написано приложение, оно все равно обречено на провал. В последующих разделах вы узнаете о десяти самых распространенных ошибках, которых следует избегать при создании баз данных.

Уверенность в том, что клиенты знают, чего хотят

Обычно клиенты приглашают вас спроектировать для них базу данных, когда сталкиваются с трудностями в получении нужной информации, поскольку их методы не работают. Клиенты часто уверены, что им известно, в чем состоит проблема и как ее решить. По их мнению, единственное, что они должны сделать, — это рассказать, что именно вам следует сделать.

Однако давать клиенту в точности то, что он просит, — залог заведомой неудачи. Большинство пользователей и менеджеров не обладают достаточным уровнем знаний и квалификацией для точной идентификации проблемы, поэтому у них практически нет шансов найти ее лучшее решение.

Перед вами стоит задача тактично убедить клиента в том, что вы эксперт по анализу и проектированию систем и что вам самому необходимо провести тщательные исследования, чтобы выявить реальную причину проблемы. Ведь причина обычно скрыта за более заметными признаками.

Игнорирование масштаба проекта

На начальной стадии разработки проекта ваш клиент рассказывает, чего именно он ожидает от нового приложения. К сожалению, клиент почти всегда забывает рассказать что-то важное, причем его забывчивость, как правило, не ограничивается каким-то одним аспектом. При этом новые требования клиента добавляются в проект уже на стадии его реализации. И если вам за выполнение проекта установлена не почасовая оплата, то увеличение его масштабов может привести к тому, что некогда прибыльный проект станет убыточным. Поэтому все, что вы обязуетесь сделать, должно быть зафиксировано в письменном виде еще до начала работы над проектом. Все новые требования и соответствующие им объемы работ, а также бюджет необходимо фиксировать в письменном виде как дополнение к исходному техническому заданию.

Учет только технических факторов

Разработчики приложений часто рассматривают свои будущие проекты только с точки зрения технической выполнимости, а затем оценивают объем работ и время выполнения, исходя из одного этого фактора. Однако на проект могут оказывать влияние также максимальная стоимость, доступность ресурсов, график работ и организационная политика. На практике эти аспекты могут

превратить технически выполнимый проект в сплошной кошмар. Прежде чем начинать разработку проекта, изучите все сопутствующие ему нетехнические факторы. Возможно, вы решите, что нет смысла заниматься данным проектом, и намного лучше прийти к этому заключению в самом начале, чем тогда, когда на проект уже затрачены большие усилия и ресурсы.

Отсутствие обратной связи с пользователями

Вначале вы, возможно, будете склонны слушать только нанявших вас менеджеров, а пользователи не будут иметь ни малейшего влияния на вашу работу, ведь не они платят вам. С другой стороны, может существовать веская причина также игнорировать и менеджеров, поскольку они обычно даже представления не имеют о том, что в действительности нужно пользователям. Это совершенно не означает, что вы лучше всех знаете о том, какой должна быть база данных и как она должна функционировать. Сотрудники, занятые вводом данных, редко осведомлены об организационных аспектах работы, а многие менеджеры имеют лишь смутное представление о некоторых вопросах, связанных с вводом данных. Если вы изолируете себя от любой из этих двух групп, то наверняка создадите систему, которая не решит ничьих проблем. Вы сможете многому научиться как у менеджеров, так и у рядовых пользователей, если сумеете задать им правильные вопросы.

Использование только своей любимой среды разработки

Вероятно, вы потратили месяцы или даже годы на то, чтобы стать специалистом в использовании конкретной СУБД или среды разработки. Но ваша любимая среда, какой бы она ни была, имеет как достоинства, так и недостатки. Время от времени вам будут попадаться задачи, которые предъявляют высокие требования именно к тем областям, в которых ваша любимая среда разработки находится отнюдь не на высоте. Так что, вместо того чтобы настаивать на реализации не самого лучшего решения, следует мужественно примириться с суровой необходимостью, т.е. остановиться и рассудить здраво. У вас есть два варианта. Первый — освоить более подходящий инструмент, а затем использовать его. Второй — это чистосердечно заявить своим клиентам, что их задачу лучше решать с помощью инструмента, в котором вы явно не эксперт, а затем предложить им нанять того, кто сможет эффективно с ним работать. Такое

профессиональное поведение заставит клиентов еще больше вас уважать. (К сожалению, если вы работаете не на себя, а являетесь наемным работником, то существует опасность, что вас могут просто уволить. Поэтому лучший выход — освоить новый инструментарий.)

Использование только своей любимой системной архитектуры

Никто не может быть экспертом во всех областях. Все СУБД разные. Одна или две системы, в которых вы хорошо разбираетесь, могут не подходить для полученного вами задания. Для любой задачи нужно выбрать наилучшую архитектуру, даже если это означает передачу задания кому-то другому. Лучше совсем не получить задание, чем получить его любой ценой и создать такую систему, которая никому не нужна.

Проектирование таблиц базы данных отдельно друг от друга

Если неправильно определить объекты данных и их связи друг с другом, то в базе могут появиться такие таблицы, из-за которых в данных постоянно будут возникать ошибки, что может свести на нет ценность любых результатов. Чтобы спроектировать добротную базу данных, необходимо проанализировать общую схему объектов данных и тщательно спланировать их связи друг с другом. Как правило, для любой задачи существует несколько правильных вариантов решения. Необходимо определить, какой из них вам подходит больше всего, учитывая при этом потребности клиента, как нынешние, так и будущие.

Отказ от консультаций с другими специалистами

Никто не совершенен. Даже самый лучший проектировщик или разработчик может пропустить важные моменты, которые очевидны для любого, кто взглянет на ситуацию с другой точки зрения. На практике вынесение своего проекта на суд общественности дисциплинирует разработчика и чаще всего помогает избежать многочисленных неприятностей, которые в противном случае он не смог бы предусмотреть. Прежде чем приступать к разработке

приложения, обязательно покажите свой проект профессионалам. Не помешает также показать его и клиенту.

Игнорирование бета-тестирования

Полезное приложение, работающее с базами данных, не может быть простым, а значит, не может не содержать ошибок. Даже если вы будете проверять свое приложение всеми способами, до которых только сможете додуматься, то все равно в нем останутся незамеченными участки, которые могут привести к сбоям. Бета-тестирование — это передача приложения людям, которые не знают, как оно спроектировано. Вот они-то наверняка столкнутся со всеми неприятностями, которые вам никогда не встретятся по той причине, что вы *слишком много* знаете о своем приложении. Если они к тому же знакомы с самими данными (но не с базой данных), то, скорее всего, будут использовать приложение так, как работали бы с ним для удовлетворения своих повседневных потребностей. В результате они смогут выявить запросы, которые выполняются слишком долго. Если вы исправите выявленные ими ошибки и узкие места, то избежите жалоб и негативных отзывов, которые все равно настигли бы вас в процессе рабочей эксплуатации системы.

Отказ от создания документации

Если вы думаете, что ваше приложение настолько совершенно, что его не придется пересматривать, то вы сильно ошибаетесь. Единственное, в чем можно быть абсолютно уверенным, — все течет, все меняется, и это следует учитывать. И если тщательно не документировать то, что было сделано, фиксируя, почему сделано именно так, а не иначе, то через полгода вы уже не сможете этого вспомнить. Кроме того, если вы перейдете в другой отдел или уволитесь, не оставив документацию по своему проекту, то ваш преемник наверняка не сможет внести никаких изменений в ваше творение, чтобы оно соответствовало новым требованиям. В отсутствие документации вашему преемнику, скорее всего, придется выбросить ваше приложение и начать все с начала.

СОВЕТ

Документируйте свою работу не просто в достаточной степени, а с большим запасом. Делайте документацию более подробной, чем это нужно с точки зрения здравого смысла. Если через шесть или восемь месяцев вы вернетесь к проекту, то только скажете себе спасибо за такую детализацию.

Глава 25

Десять советов по извлечению данных

В ЭТОЙ ГЛАВЕ...

- Проверка структуры базы данных
- Использование тестовых баз данных
- Тщательная проверка любых запросов с объединениями
- Проверка запросов с подзапросами
- Использование предложения `GROUP BY` с итоговыми функциями
- Внимательное отношение к ограничениям в предложении `GROUP BY`
- Использование круглых скобок в выражениях
- Защита базы данных посредством управления полномочиями
- Регулярное резервное копирование базы данных
- Упреждение ошибок и их обработка

База данных может быть настоящим виртуальным кладом с информационными сокровищами, но, подобно сокровищам карибских пиратов, то, что вам действительно нужно, скорее всего, зарыто и спрятано подальше от людских глаз. Чтобы добраться до этой скрытой информации, вам потребуется SQL-инструкция `SELECT`. Но даже если вы четко знаете, что вам действительно нужно, правильно составить SQL-запрос может оказаться

достаточно трудно. Если в формулировке запроса хоть немного отклониться в сторону, можно получить неверный результат. Если полученные результаты покажутся вам похожими на ожидаемые, то такая их близость может лишь ввести вас в заблуждение. Чтобы уменьшить риск получения некорректных результатов, придерживайтесь десяти принципов, изложенных ниже.

Проверяйте структуру базы данных

Если полученные из базы данных результаты не кажутся вам разумными, проверьте ее структуру. Существует много баз данных с неудачной структурой, и если вы работаете с такой базой, то вначале исправьте ее структуру и лишь затем используйте другие средства. Помните: хорошая структура — ключ к обеспечению 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, отобранных для примера. В этих строках должны содержаться такие данные, чтобы можно было знать наперед, каким должен быть результат ваших запросов. Отправляйте запросы в тестовую базу, а затем проверяйте, соответствуют ли их результаты ожидаемым. Если нет, то, возможно, запрос нуждается в корректировке. Если же запрос правильный, а результат — нет, то, видимо, придется пересмотреть структуру базы данных.

Создайте несколько наборов контрольных данных, в которых обязательно должны быть "особые случаи", например пустые таблицы или значения крайних точек допустимых диапазонов. Постарайтесь смоделировать самые невероятные случаи, а затем проверьте, правильно ли ведет себя система. Возможно, в ходе проверки нетипичных случаев вас осенит идея относительно решения какой-либо из более тривиальных проблем.

Дважды проверяйте запросы с объединениями

Операторы объединения никто не отважится назвать интуитивно понятными. Если какой-либо из них находится в вашем запросе, то, прежде чем

добавлять в запрос предложение WHERE или другие усложняющие его компоненты, обязательно убедитесь, что он действует в соответствии с вашими ожиданиями.

Трижды проверяйте запросы с подзапросами

Подзапросы позволяют отбирать данные из одной таблицы на основании результатов запроса, обращенного к другой. Сложность формирования таких запросов создает благоприятную почву для возникновения ошибок. Поэтому необходимо тщательно проверять, те ли данные возвращает внутренняя инструкция SELECT, которые нужны внешней инструкции SELECT для получения конечного результата. Если в запросе существует более двух уровней подзапросов, то контролю данных следует уделить еще больше внимания.

Подводите итоги, используя предложение GROUP BY

Предположим, у вас есть таблица NATIONAL, содержащая поля с фамилиями игроков (Player), названиями команд (Team) и количеством хоум-ранов (Homers) всех бейсболистов 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 вы получите список команд с общим количеством успешных хоум-ранов, выполненных всеми ее игроками.

Внимательно относитесь к ограничениям в предложении GROUP BY

Предположим, вам нужен список самых результативных хиттеров Национальной лиги. Рассмотрим следующий запрос.

```
SELECT Player, Team, Homers
   FROM NATIONAL
```

```
   WHERE Homers >= 20
   GROUP BY Team ;
```

В большинстве СУБД этот запрос вернет сообщение об ошибке. Обычно в список выборки помещают только те столбцы, которые используются при группировке, или те, по которым подводятся итоги. С учетом этого следующий запрос будет более работоспособным.

```
SELECT Player, Team, Homers
   FROM NATIONAL
   WHERE Homers >= 20
   GROUP BY Team, Player, Homers ;
```

Поскольку все столбцы, которые вы хотите отобразить, указаны в предложении `GROUP BY`, запрос будет выполнен успешно и вернет те результаты, которые вам нужны. Благодаря этой формулировке полученный список будет сначала отсортирован по полю `Team`, а затем — по полям `Player` и `Homers`.

Используйте круглые скобки с операторами AND, OR и NOT

Когда операторы `AND` и `OR` используются совместно в одном выражении, SQL может обработать его не в том порядке, который вы ожидаете. Чтобы полученный результат был именно таким, на какой вы рассчитываете, используйте круглые скобки. Несколько лишних нажатий клавиш — достаточно низкая цена за гарантированно достоверные результаты.

СОВЕТ

Кроме того, круглые скобки гарантируют, что оператор `NOT` будет применен именно к нужному элементу или выражению.

Контролируйте полномочия на извлечение данных

Многие пользователи не пользуются средствами безопасности, предлагаемыми их СУБД. Они просто не хотят утруждать себя, считая, что неприятности с неправомерным использованием данных случаются только у других. Не обманывайте себя. Установите и поддерживайте систему безопасности для всех баз данных, имеющих для вас хоть какую-то ценность.

Регулярно выполняйте резервное копирование своих баз данных

Если ваш жесткий диск был уничтожен скачком напряжения в сети, пожаром, землетрясением или другой катастрофой, то извлечь из него данные будет нелегко. (Не следует забывать, что компьютеры иногда "умирают" и без видимых причин.) Поэтому чаще делайте резервные копии и храните их носители в безопасном месте.

Что считать безопасным местом, зависит от того, насколько важными являются ваши данные. Этим местом может быть несгораемый сейф, находящийся в одной комнате с компьютером или в другом здании. Или же это может быть облачное хранилище либо бетонированный бункер, выдолбленный в скале и укрепленный настолько, что сможет выдержать ядерный удар. Решайте сами, какой уровень безопасности соответствует важности ваших данных.

Тщательно обрабатывайте ошибочные состояния

Вводите ли вы разовые запросы с консоли или вставляете их в приложение, все равно время от времени вместо нужных вам результатов вы будете получать сообщения об ошибках. Работая с консоли, вы на основе полученного сообщения сможете сразу же решить, что делать дальше. Что же касается приложения, то здесь ситуация совершенно иная. Пользователь приложения, скорее всего, не будет знать, какое действие является правильным. Поэтому организуйте в своем приложении как можно более полную обработку ошибок, чтобы оградить пользователей от необходимости принятия решений. Создание процедур обработки ошибок требует больших усилий, но это лучше, чем заставлять пользователя тупо смотреть на "зависший" экран.

Приложение

Ключевые слова ISO/IEC SQL:2016

ABS	ALL	ALLOCATE	ALTER
AND	ANY	ARE	ARRAY
ARRAY_AGG	ARRAY_MAX_CARDINALITY		AS
ASENSITIVE	ASYMMETRIC	AT	ATOMIC
AUTHORIZATION	AVG	BEGIN	BEGIN_FRAME
BEGIN_PARTITION	BETWEEN	BIGINT	BINARY
BLOB	BOOLEAN	BOTH	BY
CALL	CALLED	CARDINALITY	CASCADED
CASE	CAST	CEIL	CEILING
CHAR	CHAR_LENGTH	CHARACTER	CHARACTER_LENGTH
CHECK	CLASSIFIER	CLOB	CLOSE
COALESCE	COLLATE	COLLECT	COLUMN
COMMIT	CONDITION	CONNECT	CONSTRAINT
CONTAINS	CONVERT	CORR	CORRESPONDING
COUNT	COVAR_POP	COVAR_SAMP	CREATE
CROSS	CUBE	CUME_DIST	CURRENT
CURRENT_CATALOG	CURRENT_DATE	CURRENT_DEFAULT_TRANSFORM_GROUP	
CURRENT_PATH	CURRENT_ROLE	CURRENT_ROW	CURRENT_SCHEMA

CURRENT_TIME	CURRENT_ TIMESTAMP	CURRENT_ TRANSFORM_GROUP_ FOR_TYPE	
		CURRENT_USER	CURSOR
CYCLE	DATE	DAY	DEALLOCATE
DEC	DECFLOAT	DECIMAL	DECLARE
DEFAULT	DEFINE	DELETE	DENSE_RANK
DEREF	DESCRIBE	DETERMINISTIC	DISCONNECT
DISTINCT	DO	DOUBLE	DROP
DYNAMIC	EACH	ELEMENT	ELSE
ELSEIF	EMPTY	END	END-EXEC
END_FRAME	END_PARTITION	EQUALS	ESCAPE
EVERY	EXCEPT	EXEC	EXECUTE
EXISTS	EXP	EXTERNAL	EXTRACT
FALSE	FETCH	FILTER	FIRST_VALUE
FLOAT	FLOOR	FOR	FOREIGN
FRAME_ROW	FREE	FROM	FULL
FUNCTION	FUSION	GET	GLOBAL
GRANT	GROUP	GROUPING	GROUPS
HANDLER	HAVING	HOLD	HOUR
IDENTITY	IF	IN	INDICATOR
INITIAL	INNER	INOUT	INSENSITIVE
INSERT	INT	INTEGER	INTERSECT
INTERSECTION	INTERVAL	INTO	IS
ITERATE	JOIN	JSON_ARRAY	JSON_ARRAYAGG
JSON_EXISTS	JSON_OBJECT	JSON_OBJECTAGG	JSON_QUERY
JSON_TABLE	JSON_TABLE_ PRIMITIVE		JSON_VALUE
LAG	LANGUAGE	LARGE	LAST_VALUE
LATERAL	LEAD	LEADING	LEAVE

LEFT	LIKE	LIKE_REGEX	LN
LOCAL	LOCALTIME	LOCALTIMESTAMP	LOWER
MATCH	MATCHES	MATCH_NUMBER	MATCH_RECOGNIZE
MAX	MEMBER	MERGE	METHOD
MIN	MINUTE	MOD	MODIFIES
MODULE	MONTH	MULTISET	NATIONAL
NATURAL	NCHAR	NCLOB	NEW
NO	NONE	NORMALIZE	NOT
NTH_VALUE	NTILE	NULL	NULLIF
NUMERIC	OCCURRENCES_ REGEX		OCTET_LENGTH
OF	OFFSET	OLD	OMIT
ON	ONE	ONLY	OPEN
OR	ORDER	OUT	OUTER
OVER	OVERLAPS	OVERLAY	PARAMETER
PARTITION	PATTERN	PER	PERCENT
PERCENT_RANK	PERCENTILE_CONT	PERCENTILE_DISC	PERIOD
PORTION	POSITION	POSITION_REGEX	POWER
PRECEDES	PRECISION	PREPARE	PRIMARY
PROCEDURE	RANGE	RANK	READS
REAL	RECURSIVE	REF	REFERENCES
REFERENCING	REGR_AVGX	REGR_AVGY	REGR_COUNT
REGR_INTERCEPT	REGR_R2	REGR_SLOPE	REGR_SXX
REGR_SXY	REGR_SYY	RELEASE	REPEAT
RESIGNAL	RESULT	RETURN	RETURNS
REVOKE	RIGHT	ROLLBACK	ROLLUP
ROW	ROW_NUMBER	ROWS	RUNNING
SAVEPOINT	SCOPE	SCROLL	SEARCH
SECOND	SEEK	SELECT	SENSITIVE

SESSION_USER	SET	SHOW	SIGNAL
SIMILAR	SKIP	SMALLINT	SOME
SPECIFIC	SPECIFICTYPE	SQL	SQLEXCEPTION
SQLSTATE	SQLWARNING	SQRT	START
STATIC	STDDEV_POP	STDDEV_SAMP	SUBMULTISET
SUBSET	SUBSTRING	SUBSTRING_REGEX	SUCCEEDS
SUM	SYMMETRIC	SYSTEM	SYSTEM_TIME
SYSTEM_USER	TABLE	TABLESAMPLE	THEN
TIME	TIMESTAMP	TIMEZONE_HOUR	TIMEZONE_MINUTE
TO	TRAILING	TRANSLATE	TRANSLATE_REGEX
TRANSLATION	TREAT	TRIGGER	TRIM
TRIM_ARRAY	TRUE	TRUNCATE	UESCAPE
UNION	UNIQUE	UNKNOWN	UNNEST
UNTIL	UPDATE	UPPER	USER
USING	VALUE	VALUES	VALUE_OF
VAR_POP	VAR_SAMP	VARBINARY	VARCHAR
VARYING	VERSIONING	WHEN	WHENEVER
WHERE	WHILE	WIDTH_BUCKET	WINDOW
WITH	WITHIN	WITHOUT	YEAR

Предметный указатель

К

Л

М

Н

О

П

Р

С

Т

У

Ф

SQL
ПОЛНОЕ РУКОВОДСТВО
ТРЕТЬЕ ИЗДАНИЕ

Джеймс Р. Грофф
Пол Н. Вайнберг
Эндрю Дж. Оппель

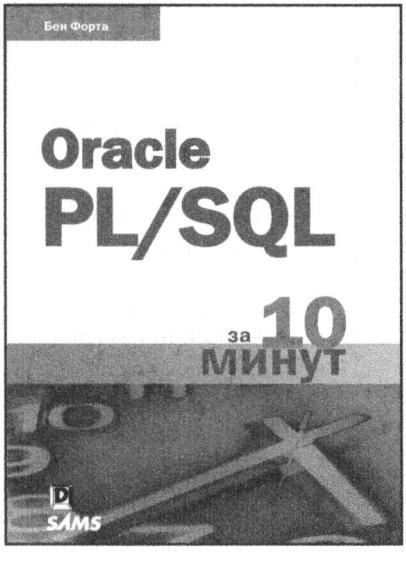
www.williamspublishing.com

Эта книга расскажет вам, как работать с командами и инструкциями SQL, создавать и настраивать реляционные базы данных, загружать и модифицировать объекты баз данных, выполнять мощные запросы, повышать производительность и выстраивать систему безопасности. Вы узнаете, как использовать инструкции DDL и применять API, интегрировать XML и сценарии Java, использовать объекты SQL, создавать веб-серверы, работать с удаленным доступом и выполнять распределенные транзакции. В этой книге вы найдете такие сведения, как описания работы с базами данных в памяти, потоковыми и встраиваемыми базами данных, базами данных для мобильных и наладонных устройств, и многое другое.

ISBN 978-5-907114-26-5 **в продаже**

ORACLE PL/SQL ЗА 10 МИНУТ

Бен Форта



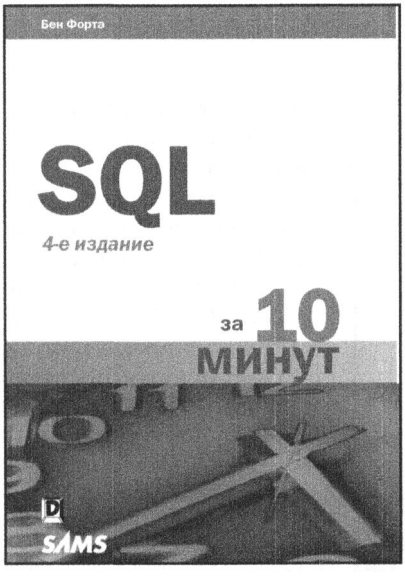
www.williamspublishing.com

Этот краткий справочник состоит из 26 уроков. Потратив не более 10 минут на каждый, читатель сможет быстро и легко овладеть основами программирования на языке PL/SQL для работы с СУБД Oracle. Учебный материал уроков начинается с самых основ извлечения информации из базы данных и постепенно переходит к более сложным вопросам, включая соединения, подзапросы, регулярные выражения и полноценный текстовый поиск, создание, изменение и удаление таблиц, хранимые процедуры, курсоры, триггеры, табличные ограничения и многое другое. Книга рассчитана на широкий круг читателей, стремящихся освоить язык PL/SQL для работы с СУБД Oracle.

ISBN 978-5-907114-46-3 **в продаже**

SQL ЗА 10 МИНУТ
4-е издание

Бен Форта



www.williamspublishing.com

В книге предлагаются простые и практичные решения для тех, кто хочет быстро получить результат. Проработав все 22 урока, на каждый из которых придется затратить не более 10 минут, вы узнаете обо всем, что необходимо для практического применения SQL. Приведенные в книге примеры подходят для IBM DB2, Microsoft Access, Microsoft SQL Server, MySQL, Oracle, PostgreSQL, SQLite, MariaDB и Apache OpenOffice Base.

Основные темы книги:

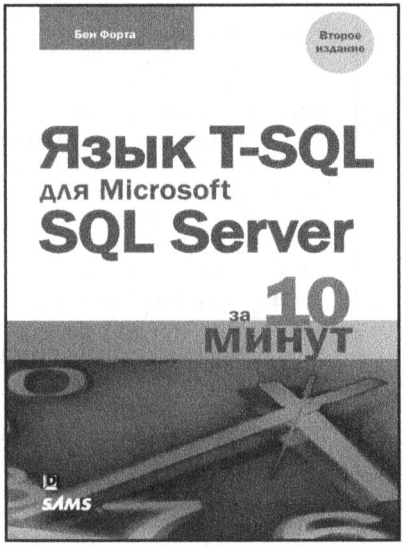
- основные инструкции SQL;
- создание сложных SQL-запросов с множеством предложений и операторов;
- извлечение, сортировка и форматирование данных;
- получение конкретных данных с помощью различных методов фильтрации;
- применение итоговых функций для получения сводных данных;
- объединение реляционных таблиц;
- добавление, обновление и удаление данных;
- создание и изменение таблиц;
- работа с представлениями, хранимыми процедурами и многое другое.

ISBN 978-5-6041393-9-4

в продаже

ЯЗЫК T-SQL ДЛЯ MICROSOFT SQL SERVER ЗА 10 МИНУТ
2-Е ИЗДАНИЕ

Бен Форта



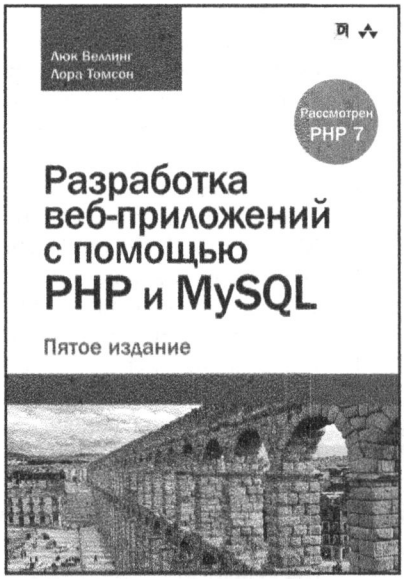
www.williamspublishing.com

Этот краткий справочник состоит из 30 уроков. Потратив не более 10 минут на каждый, читатель сможет быстро и легко овладеть основами программирования на языке T-SQL для работы с РСУБД Microsoft SQL Server. Учебный материал уроков начинается с самых основ извлечения информации из базы данных и постепенно переходит к более сложным вопросам, включая соединения, подзапросы, регулярные выражения и полноценный текстовый поиск, создание, изменение и удаление таблиц, хранимые процедуры, курсоры, триггеры, табличные ограничения, локализацию, обработку данных в форматах XML и JSON и многое другое. Книга рассчитана на широкий круг читателей, стремящихся освоить язык T-SQL для работы с РСУБД SQL Server.

ISBN 978-5-9909445-2-7 **в продаже**

РАЗРАБОТКА ВЕБ-ПРИЛОЖЕНИЙ С ПОМОЩЬЮ PHP И MYSQL
5-е издание

Люк Веллинг
Лора Томсон



www.williamspublishing.com

В этом новом 5-м издании книги, признанной наиболее ясным, удобным и практичным руководством по разработке с использованием PHP и MySQL, полностью отражены возможности последних версий PHP и MySQL.

В первой части содержится ускоренный курс по PHP, в котором описано хранение/извлечение данных, массивы, строки, регулярные выражения, повторное использование кода, объекты и обработка ошибок/исключений. Во второй части раскрывается проектирование, создание, доступ и программирование для баз данных MySQL.

Третья часть посвящена безопасности веб-приложений; в ней добавлена новая информация по угрозам веб-безопасности, приведены инструкции по построению защищенных веб-приложений, а также рассмотрена реализация аутентификации в PHP и MySQL.

Отдельная часть по расширенным приемам PHP охватывает все темы, начиная с работы в сети и взаимодействия с файловой системой и заканчивая интернационализацией и локализацией, генерированием изображений, а также инфраструктурами и компонентами PHP.

ISBN 978-5-9908911-9-7 **в продаже**

PHP и MySQL: создание интернет-магазинов *2-е издание*

Ларри Ульман

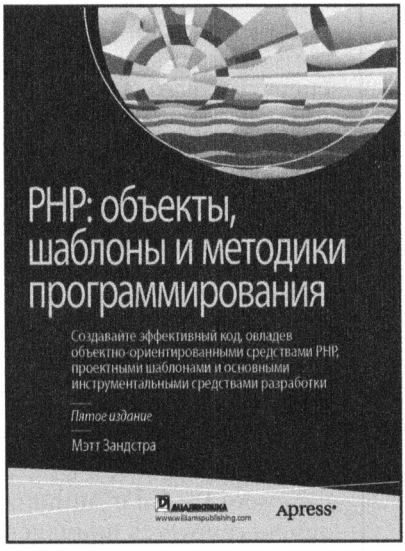
www.williamspublishing.com

В книге рассматриваются примеры двух полнофункциональных интернет-магазинов, благодаря изучению которых читатели смогут сравнить разные сценарии электронной коммерции. Вы узнаете, как спроектировать визуальный интерфейс и создать базу данных сайта, как реализовать представление контента и сгенерировать онлайн-каталог, как управлять корзиной товаров и проводить платежи, как принимать и выполнять заказы с учетом требований безопасности и эффективности. Второе издание книги включает описание современных функциональных средств, присущих платежным системам PayPal и Authorize.net, демонстрируется применение технологий Ajax и JavaScript, а также описано подключение интернет-магазинов к платежной системе Яндекс. Деньги.

ISBN 978-5-8459-1939-7 **в продаже**

PHP: ОБЪЕКТЫ, ШАБЛОНЫ И МЕТОДИКИ ПРОГРАММИРОВАНИЯ
ПЯТОЕ ИЗДАНИЕ

Мэтт Зандстра



www.williamspublishing.com

В этой книге рассматриваются методики объектно-ориентированного программирования на PHP, применение главных принципов проектирования программного обеспечения на основе классических проектных шаблонов, а также описываются инструментальные средства и нормы практики разработки, тестирования, непрерывной интеграции и развертывания надежного прикладного кода. Настоящее, пятое издание книги полностью обновлено по версии 7 языка PHP и включает описание диспетчера зависимостей Composer, инструментального средства Vagrant и рекомендаций стандартов по программированию на PHP.

Книга адресована разработчикам, твердо усвоившим основы программирования на PHP и стремящимся развить свои навыки проектирования веб-приложений, применяя нормы передовой практики разработки.

ISBN 978-5-907144-54-5 **в продаже**

...FOR Dummies™
BESTSELLING BOOK SERIES

SQL для чайников®

9-е издание

...ДЛЯ чайников™
СЕРИЯ КОМПЬЮТЕРНЫХ КНИГ ОТ ДИАЛЕКТИКИ

Шпаргалка

Критерии нормальных форм

Первая нормальная форма (1НФ)

- Таблица должна быть двухмерной, т.е. состоять из строк и столбцов.
- В каждой строке должны находиться данные, соответствующие одному объекту или его части.
- В каждом столбце должны находиться данные, относящиеся к одному из атрибутов описываемого объекта.
- В каждой ячейке таблицы (на пересечении строки и столбца) должно находиться только одно значение.
- В каждом столбце должны быть только однотипные данные.
- У каждого столбца должно быть уникальное имя.
- Никакие две строки не могут быть идентичными.
- Порядок расположения столбцов и строк не имеет значения.

Вторая нормальная форма (2НФ)

- Таблица должна быть в первой нормальной форме (1НФ).
- Все неключевые атрибуты (столбцы) должны зависеть от всего ключа.

Третья нормальная форма (3НФ)

- Таблица должна быть во второй нормальной форме (2НФ).
- У таблицы не должно быть транзитивных зависимостей.

Доменно-ключевая нормальная форма (ДКНФ)

Каждое ограничение для таблицы должно быть логическим следствием определения ключей и доменов.

BESTSELLING
BOOK
SERIES

SQL для чайников®

9-е издание

СЕРИЯ
КОМПЬЮТЕРНЫХ
КНИГ ОТ
ДИАЛЕКТИКИ

Шпаргалка

Типы данных SQL

Целочисленные типы

- INTEGER
- SMALLINT
- BIGINT
- NUMERIC
- DECIMAL

Числа с плавающей запятой

- REAL
- DOUBLE PRECISION
- FLOAT
- DECFLOAT

Двоичные строки

- BINARY
- BINARY VARYING
- BINARY LARGE OBJECT

Символьные строки

- CHARACTER
- CHARACTER VARYING (VARCHAR)
- CHARACTER LARGE OBJECT
- NATIONAL CHARACTER
- NATIONAL CHARACTER VARYING
- NATIONAL CHARACTER LARGE OBJECT

Значения даты/времени

- DATE
- TIME WITHOUT TIMEZONE
- TIMESTAMP WITHOUT TIMEZONE
- TIME WITH TIMEZONE
- TIMESTAMP WITH TIMEZONE

Интервалы

- INTERVAL DAY
- INTERVAL YEAR

Типы коллекций

- ARRAY
- MULTISET

Логические значения

- BOOLEAN

BESTSELLING
BOOK
SERIES

SQL для чайников®

9-е издание

СЕРИЯ
КОМПЬЮТЕРНЫХ
КНИГ ОТ
ДИАЛЕКТИКИ

Шпаргалка

Функции SQL

Строковые функции

Функция	*Описание*
SUBSTRING	Извлекает подстроку из исходной строки
SUBSTRING SIMILAR	Извлекает подстроку из исходной строки, используя регулярные выражения на базе стандарта POSIX
SUBSTRING_REGEX	Извлекает из строки первое вхождение шаблона регулярного выражения XQuery и возвращает одно вхождение совпавшей подстроки
TRANSLATE_REGEX	Извлекает из строки первое (или каждое) вхождение шаблона регулярного выражения XQuery и заменяет его (или их) строкой замены XQuery
UPPER	Преобразует символы строки в верхний регистр
LOWER	Преобразует символы строки в нижний регистр
TRIM	Удаляет предваряющие и завершающие пробелы
TRANSLATE (CONVERT)	Преобразует символы исходной строки из одного набора в другой

Числовые функции

Функция	*Описание*
POSITION	Возвращает начальную позицию заданной подстроки в исходной строке
CHARACTER_LENGTH	Возвращает количество символов в строке
OCTET_LENGTH	Возвращает количество октетов (байтов) в символьной строке
EXTRACT	Извлекает одно поле из значения даты/времени или интервала

Тригонометрические и логарифмические функции

- SIN, COS, TAN, ASIN, ACOS, ATAN, SINH, COSH, TANH;
- LOG(<*основание*>, <*значение*>), LOG10(<*значение*>), LN(<*значение*>).

BESTSELLING
BOOK
SERIES

SQL для чайников®

9-е издание

СЕРИЯ
КОМПЬЮТЕРНЫХ
КНИГ ОТ
ДИАЛЕКТИКИ

Шпаргалка

Функции даты/времени

Функция	*Описание*
CURRENT_DATE	Возвращает текущую дату
CURRENT_TIME(*p*)	Возвращает текущее время; *p* — точность секундной части
CURRENT_TIMESTAMP(*p*)	Возвращает текущую дату и текущее время; *p* — точность секундной части

Итоговые функции

Функция	*Описание*
COUNT	Возвращает количество строк в указанной таблице
MAX	Возвращает максимальное значение в заданном столбце
MIN	Возвращает минимальное значение в заданном столбце
SUM	Суммирует значения в заданном столбце
AVG	Возвращает среднее значение указанного столбца
LISTAGG	Преобразует значения из группы строк в строку с разделителями

Функции конструктора JSON

Функция	*Описание*
JSON_OBJECT	Конструирует объекты JSON на основе пар "имя/значение"
JSON_ARRAY	Создает массив JSON на основании списка элементов данных из реляционной таблицы
JSON_OBJECTAGG	Конструирует объект JSON путем агрегирования данных реляционной таблицы
JSON_ARRAYAGG	Создает массив JSON путем агрегирования данных реляционной таблицы

Функции запросов JSON

Функция	*Описание*
JSON_EXISTS	Определяет, будет ли значение JSON удовлетворять условию отбора, заданному в спецификации пути
JSON_VALUE	Извлекает скалярное значение SQL из значения JSON
JSON_QUERY	Извлекает массив или объект SQL/JSON из значения SQL/JSON
JSON_TABLE	Генерирует реляционную выходную таблицу на основании входных данных JSON

Освоить SQL не так уж и сложно!

Перед вами новейшее издание бестселлера, посвященное последней версии стандарта SQL. Здесь вы найдете информацию о том, как эффективно применять SQL для построения реляционных баз данных. Вы узнаете, как проектировать и защищать базы данных, а также извлекать из них всю необходимую информацию. В эпоху больших данных крайне важно иметь под рукой простое и доступное руководство по работе с информационными ресурсами.

В книге...

- Возможности SQL
- Использование SQL для создания баз данных
- Построение многотабличных реляционных баз данных
- Манипулирование информацией в базах данных
- Создание сложных запросов
- Использование реляционных операторов
- Обеспечение безопасности базы данных

Аллен Тейлор — ветеран компьютерной индустрии с тридцатилетним стажем, автор более 40 книг по компьютерной тематике. Читает лекции по базам данных и компьютерным технологиям, а также ведет онлайн-курсы по разработке баз данных.

Диалектика
www.dialektika.com

Базы данных/SQL

ISBN 978-5-907144-81-1
19116
9 785907 144811

для чайников®