Глава 3

Компоненты SQL

В этой главе...

- Создание баз данных
- Обработка данных
- Защита баз данных

SQL — это язык, специально разработанный для создания и поддержки данных в реляционных базах. И хотя компании, поставляющие системы для управления такими базами, предлагают свои реализации SQL, развитие самого языка определяется и контролируется стандартом ISO/ANSI (этот стандарт пересматривался в 2003 году и дополнялся в 2005 году). Все реализации в большей или меньшей степени отличаются от стандарта. Но как можно более полное следование стандарту — главное условие для работы базы данных и связанных с ней приложений на более чем одной платформе.

SQL не является 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 входящих в состав SQL трех языков.

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

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

Язык определения данных

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

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

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

Сама же база данных состоит из каталогов. В некоторых источниках базу данных называют *кластером*.

Создание таблиц

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

После того как таблица создана, ее можно заполнять данными. (Впрочем, загружать данные — это дело языка DML, а не DDL.) Если требования изменяются, то изменить структуру уже созданной таблицы можно с помощью команды `ALTER TABLE` (изменить таблицу). Со временем таблица может стать бесполезной или устареть. И если час таблицы пробил, то уничтожить ее можно с помощью команды `DROP`. Существующие в SQL разные формы команд — `CREATE` и `ALTER`, а также `DROP` — как раз и представляют собой язык DDL.

Предположим, что вы проектируете базы данных и не хотите, чтобы их таблицы постепенно, по мере обновления имеющихся в них данных, стали бы "неудобоваримыми". Чтобы обеспечить поддержку целостности данных, принимается решение относительно нормализации базы данных. *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 друг с другом.

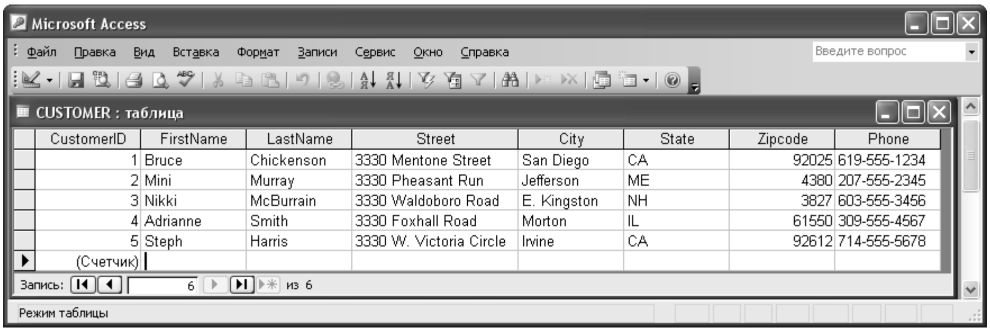
К примеру, вы можете создать таблицу CUSTOMER (клиент), в которой имеются такие атрибуты: `CUSTOMER.CustomerID` (идентификатор клиента), `CUSTOMER.FirstName` (имя), `CUSTOMER.LastName` (фамилия), `CUSTOMER.Street` (улица), `CUSTOMER.Sity` (город), `CUSTOMER.State` (штат), `CUSTOMER.Zipcode` (почтовый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

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

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

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

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



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	4380	207-555-2345
3	Nikki	McBurrain	3330 Waldoboro Road	E. Kingston	NH	3827	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
(Счетчик)							

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

Если используемая вами реализация SQL не полностью соответствует последнему стандарту SQL, то синтаксис, которым вам придется пользоваться, может и не совпадать с приведенным в этой книге. За подробной информацией вы можете обратиться к документации СУБД.

Комната с представлениями

Иногда из таблицы CUSTOMER (клиент) вам требуется получить некоторую конкретную информацию. При этом не обязательно просматривать все подряд — нужны только конкретные столбцы и строки. В таком случае вам потребуется представление.

Представления — это виртуальные таблицы. В большинстве реализаций они не являются независимыми физическими объектами. Определения представлений существуют только в таблицах метаданных, и данные на самом деле поступают из таблиц, на основе которых эти представления созданы. Эти данные больше нигде не хранятся. Одни представления состоят из определенных столбцов и строк одной таблицы, другие же, которые называются *многотабличными представлениями*, создаются на основе более чем двух таблиц.

Однотабличное представление

Иногда данные, которые дают ответ на заданный вопрос, содержатся всего в одной таблице базы данных. Если вся необходимая вам информация находится в одной таблице, то можно создать однотабличное представление данных. Скажем, например, что нужно просмотреть имена, фамилии и телефонные номера всех клиентов, которые живут в штате Нью-Гэмпшир (который обозначается аббревиатурой NH). Тогда на основе таблицы CUSTOMER можно создать представление, содержащее только те данные, которые вам нужны. Оно создается с помощью следующей команды:

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

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

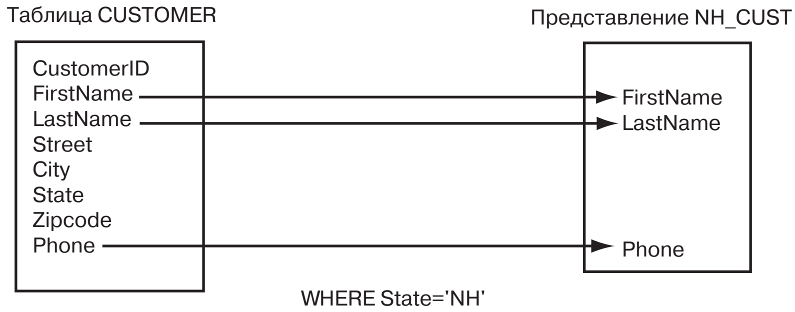


Рис. 3.2. Создание представления `NH_CUST` из таблицы CUSTOMER

Этот код безупречно правильный, но несколько громоздкий. Ту же операцию можно выполнить и с помощью более коротких команд. Это возможно тогда, когда установленная у вас реализация SQL допускает, что если в перечисленных атрибутах не указаны ссылки на таблицу, то все атрибуты относятся к таблице из предложения `FROM`. Если ваша система в состоянии сделать это разумное допущение, то команду можно сократить до следующей:

```
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 команд `ALTER TABLE`. Конечно, если предложение `JOIN` (объединить) не используется, такого не случится. А для представлений с предложениями `JOIN` лучше использовать полные имена. С предложением `JOIN` вы познакомитесь в главе 10.

Создание многотабличного представления

Чтобы получать ответы на имеющиеся вопросы, часто приходится выбирать данные из более чем двух 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. Создав представление, его можно использовать снова и снова. При каждом таком использовании представление отображает последние изменения в таблицах, на основе которых это представление создано.

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

Обратите внимание, что в некоторых столбцах табл. 3.1 существует ограничение `NOT NULL` (недопустимость пустых значений). 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.)

Таблица 3.1. Таблицы базы данных магазина спорттоваров

Таблица	Столбец	Тип данных	Ограничение
CUSTOMER	`CustomerID` (идентификационный номер клиента),	`INTEGER`	`NOT NULL` (не может быть неопределенным значением)
	`FirstName` (имя),	`CHARACTER (15)`	
	`LastName` (фамилия),	`CHARACTER (20)`	`NOT NULL`
	`Street` (улица),	`CHARACTER (25)`	
	`City` (город),	`CHARACTER (20)`	
	`State` (штат),	`CHARACTER (2)`	
	`Zipcode` (почтовый код),	`INTEGER`	
	`Phone` (телефон)	`CHARACTER (13)`	
PRODUCT	`ProductID` (идентификационный номер товара),	`INTEGER`	`NOT NULL`
	`Name` (название),	`CHARACTER (25)`	
	`Description` (описание),	`CHARACTER (30)`	
	`Category` (категория),	`CHARACTER (15)`	
	`VendorID` (идентификационный номер поставщика),	`INTEGER`	
	`VendorName` (наименование поставщика)	`CHARACTER (30)`	
INVOICE	`InvoiceNumber` (номер счета-фактуры),	`INTEGER`	`NOT NULL`
	`CustomerID` (идентификационный номер покупателя),	`INTEGER`	
	`InvoiceDate` (дата выписки счета-фактуры),	`DATE`	
	`TotalSale` (всего продано на сумму),	`NUMERIC (9,2)`	
	`TotalRemitted` (всего оплачено),	`NUMERIC (9,2)`	
	`FormOfPayment` (форма платежа)	`CHARACTER (10)`	
INVOICE_LINE	`LineNumber` (номер строки),	`INTEGER`	`NOT NULL`
	`InvoiceNumber` (номер счета-фактуры),	`INTEGER`	
	`ProductID` (идентификационный номер товара),	`INTEGER`	
	`Quantity` (количество),	`INTEGER`	
	`SalePrice` (продано по цене)	`NUMERIC (9,2)`	

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

- ✓ Таблицу CUSTOMER связывает с другой таблицей, INVOICE, отношение "один ко многим". Один клиент может сделать множество покупок, в результате чего получится множество счетов-фактур. Однако каждый счет-фактура имеет отношение к одному, и только к одному клиенту.

- ✓ Таблицу INVOICE связывает с таблицей INVOICE_LINE также отношение "один ко многим". В счете-фактуре может быть несколько строк, но каждая строка находится в одном, и только в одном счете-фактуре.
- ✓ Таблицу PRODUCT с таблицей INVOICE_LINE связывает отношение "один ко многим". Каждый товар может находиться во множестве строк в одном или многих счетах-фактурах. Однако каждая строка относится к одному, и только к одному товару.

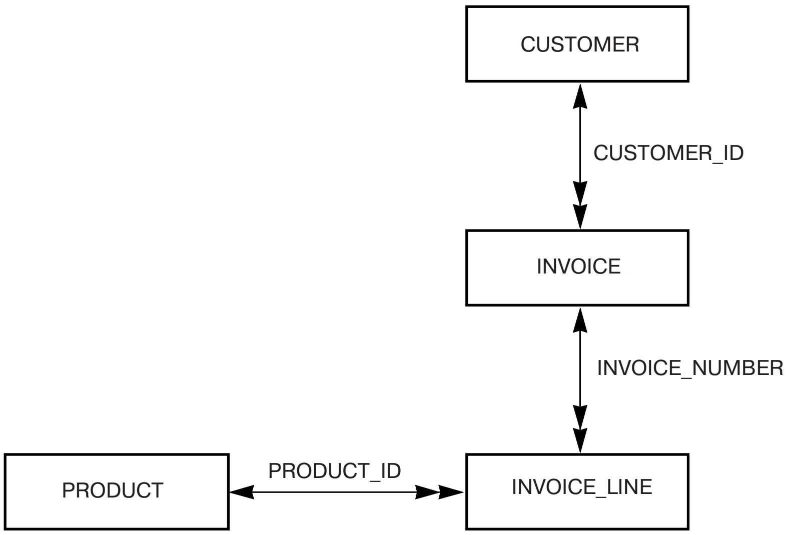


Рис. 3.3. Структура базы данных магазина спорттоваров

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

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`, а также `LineNumber` — из таблицы `INVOICE_LINE`. Нужное представление можно создавать поэтапно, используя такие команды:

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

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

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

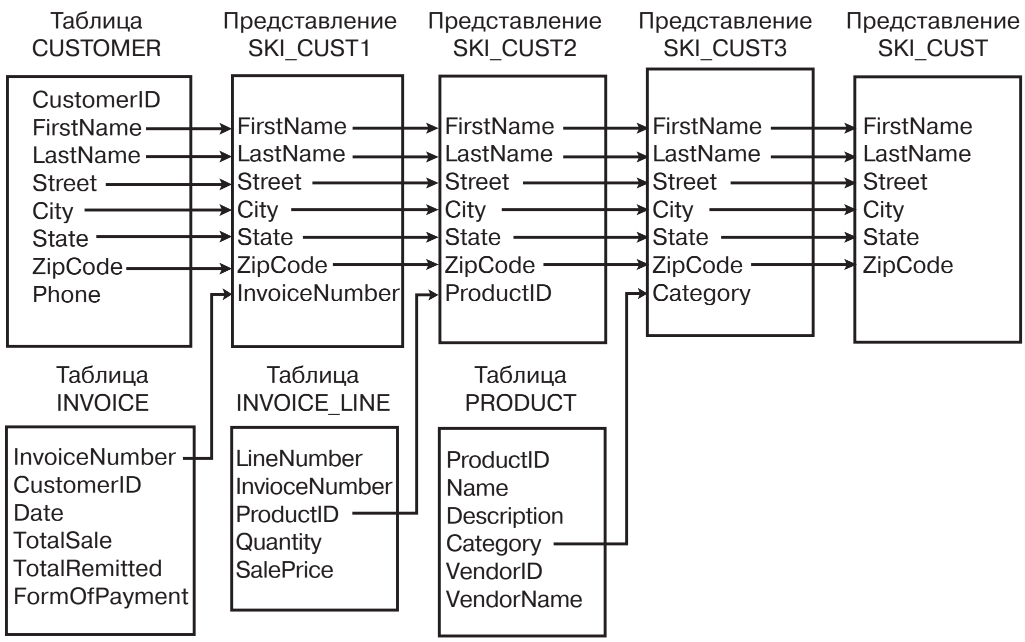


Рис. 3.4. Создание многотабличного представления с помощью оператора `JOIN`

Ниже описаны четыре приведенных оператора `CREATE VIEW`.

- ✓ Первый оператор объединяет столбцы из таблицы CUSTOMER со столбцом из таблицы INVOICE и создает представление `SKI_CUST1`.
- ✓ Второй оператор объединяет представление `SKI_CUST1` со столбцом из таблицы INVOICE_LINE, создавая таким образом представление `SKI_CUST2`.
- ✓ Третий оператор объединяет представление `SKI_CUST2` со столбцом из таблицы PRODUCT и создает представление `SKI_CUST3`.
- ✓ Четвертый оператор отбрасывает все строки, где в поле категории товара содержится не `'Ski'` (лыжи). В результате получается представление `SKI_CUST`, в котором отображаются имена, фамилии и адреса тех клиентов, которые хотя бы один раз купили товары категории `'Ski'`. Каждому из этих клиентов, даже если он покупал лыжи много раз, в представлении `SKI_CUST` будет соответствовать только одна запись. Это достигается благодаря ключевому слову `DISTINCT` (отдельный), находящемся в инструкции `SELECT` четвертого оператора `CREATE VIEW`. (Операторы `JOIN` подробно рассматриваются в главе 10.)

Сборка таблиц в схемы

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

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

В системе, где может сосуществовать несколько не связанных друг с другом проектов, можно объединить все таблицы, связанные друг с другом, в единую схему. А из таблиц, не вошедших в эту схему, можно образовать другие схемы. Чтобы таблицы из одного проекта не оказались случайно в другом, схемам нужно присваивать имена. У каждого проекта имеется собственная схема, которую по имени можно будет отличать от других схем. Некоторые имена таблиц (например, `CUSTOMER`, `PRODUCT` и т.д.) могут встречаться сразу в нескольких проектах. Если существует хоть малейший шанс возникновения путаницы с именами, необходимо в именах таблиц указывать имя схемы (примерно так: *ИМЯ_СХЕМЫ.ИМЯ_ТАБЛИЦЫ*). Если имя схемы явно не указано, SQL будет считать, что эта таблица относится к схеме, подразумеваемой по умолчанию.

Заказ по каталогу

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

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

```
ИМЯ_КАТАЛОГА.ИМЯ_СХЕМЫ.ИМЯ_ТАБЛИЦЫ
```

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

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

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

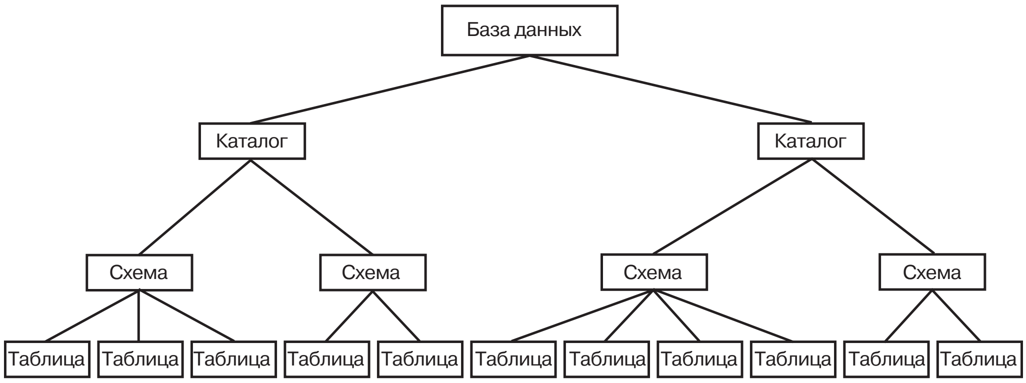


Рис. 3.5. Иерархическая структура типичной базы данных SQL

Инструкции DDL

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

- ✓ Для создания основных структур базы данных используются разные формы инструкции `CREATE`.
- ✓ Для изменения созданных структур применяется инструкция `ALTER`.
- ✓ Инструкция `DROP` применяется к таблице, чтобы не только удалить табличные данные, но и разрушить саму структуру этой таблицы.

В следующих разделах кратко описаны инструкции языка DDL; в главах 4 и 5 их мы будем использовать в примерах.

Инструкция `CREATE`

Инструкция языка SQL `CREATE` позволяет создавать объекты SQL нескольких видов, в том числе схемы, домены, таблицы и представления. С помощью оператора `CREATE SCHEMA` можно создать схему, идентифицировать ее владельца и указать набор символов, используемый по умолчанию. К примеру, вот как может выглядеть такой оператор:

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

С помощью оператора CREATE DOMAIN устанавливаются ограничения на значения в столбце или указывается порядок сопоставления. Устанавливаемые на домен 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 команду:

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

Таблицы создаются с помощью оператора CREATE TABLE, а представления — с помощью CREATE VIEW. В этой главе уже приводились примеры использования операторов CREATE TABLE и CREATE VIEW. Когда с помощью оператора CREATE TABLE создается новая таблица, то в том же операторе на ее столбцы можно установить и ограничения. Впрочем, иногда требуется устанавливать ограничения, которые относятся не только к таблице, но и ко всей схеме. В таких случаях используется оператор CREATE ASSERTION (создать утверждение).

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

Инструкция ALTER

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

Инструкция DROP

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

Язык манипулирования данными

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

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

Выражения со значениями

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

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

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

Выражения с числовыми значениями

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

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

Значения в этих примерах являются *числовыми константами*. В качестве значений можно использовать также имена столбцов, параметры, базовые переменные или подзапросы — при условии, что их значения являются числовыми. Вот несколько примеров:

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

Двоеточие в последнем примере говорит о том, что следующий за ним терм является либо параметром, либо базовой переменной.

Выражения со строковыми значениями

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

ПОМНИ!

В некоторых реализациях SQL в качестве оператора конкатенации вместо `||` используют `+`.

Таблица 3.2. Примеры конкатенации строк

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

Существуют реализации, в которых вместо конкатенации используются строковые операторы, но сам стандарт SQL эти операторы не поддерживает.

Выражения со значениями даты-времени и интервальными значениями

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

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

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

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

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

Ключевые слова `AT LOCAL` означают, что указано местное время.

Выражения со значениями интервалов работают с промежутками времени между двумя значениями 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 после истечения крайнего срока. Тогда, используя выражение со значениями интервалов, такое, например, как приведено в следующем примере, можно вычислить, сколько прошло дней задержки после крайнего срока, и, соответственно, назначить пеню.

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

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

Выражения с логическими значениями

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

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

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

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

То же самое условие можно выразить и по-другому:

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

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

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

Выражения со значениями, определяемыми пользователем

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

Выражения со значениями типа записи

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

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

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

Выражения со значениями типа коллекции

Значением выражения типа коллекции является массив.

Выражения со значениями типа ссылки

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

Предикаты

Предикаты — это используемые в SQL эквиваленты логических высказываний. Примером высказывания является следующее выражение:

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

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

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

ПОМНИ!

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

Таблица 3.3. Операторы и предикаты сравнения

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

Логические связки

Логические связки 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 этим ученикам, можно прочитать следующим образом:

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

```
"Ученику еще нет 14 лет"
```

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

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

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

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

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

Итоговые функции

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

Функция COUNT

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

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

Функция MAX

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

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

В результате появляются данные обо всех старших учениках, т.е. если возраст самого старшего ученика равен 23 годам, этот оператор возвращает данные обо всех учениках в возрасте 23 лет.

В этом запросе используется подзапрос. Этот подзапрос, `SELECT MAX(Age) FROM STUDENT`, находится в главном запросе.

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 появляются данные о самых младших учениках школы.

Функция SUM

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

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

Функция AVG

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

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

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

Подзапросы

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

Язык управления данными

В *языке управления данными* (*DCL*) существуют четыре инструкции: COMMIT (завершить), ROLLBACK (откатить), GRANT (предоставить) и REVOKE (отозвать). Все эти инструкции связаны с защитой базы от случайного или умышленного повреждения.

Транзакции

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

С целью защиты базы данных язык SQL ограничивает операции, которые могут ее изменить, так что они выполняются только в пределах транзакций. Во время транзакции SQL записывает каждую операцию с данными в файл журнала. Если транзакцию, перед тем как она будет завершена инструкцией COMMIT, что-то прервет, то можно 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 попробовать еще раз выполнить транзакцию.

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

Повреждения базы данных или некорректные результаты возможны в многопользовательской системе даже тогда, когда нет никаких аппаратных или программных отказов. Серьезные неприятности могут быть вызваны совместными действиями двух или нескольких пользователей по отношению к одной и той же таблице. SQL решает эту проблему, допуская внесение изменений только в пределах транзакций.

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

Возможно, вам не понятно, почему совместные действия двух пользователей могут привести к некорректным результатам. Для примера предположим, что Гена читает запись какой-либо таблицы из базы данных, а через мгновение Денис изменяет в этой записи значение числового поля. А теперь в то же поле Гена записывает число, полученное на основе значения, прочитанного им вначале. И так как он не знает об изменении, сделанном Денисом, то его операция является некорректной.

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

Пользователи и полномочия

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

Создатель схемы указывает, кого следует считать ее владельцем. Являясь владельцем схемы, вы можете предоставлять полномочия доступа пользователям. Любые полномочия, не предоставленные вами явно, являются недействительными. Вы также можете отозвать уже предоставленные вами полномочия. Пользователю, перед тем как получить предоставляемый вами доступ к файлам, необходимо подтвердить свою личность, пройдя для этого процедуру аутентификации. Что собой представляет эта процедура — зависит от конкретной реализации SQL.

SQL дает возможность защищать следующие объекты базы данных:

- ✓ таблицы;
- ✓ столбцы;
- ✓ представления;
- ✓ домены;
- ✓ символьные наборы;
- ✓ сопоставления;
- ✓ трансляции.

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

Стандарт SQL поддерживает различные виды защиты: *защиту просмотра*, *добавления*, *модификации*, *удаления*, *применения ссылок* и *использования* баз данных, а также виды защиты, связанные с выполнением внешних процедур.

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

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

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

Таблица 3.4. Виды защиты

Действие по защите	Оператор
Позволяет просматривать таблицу	GRANT SELECT
Не позволяет просматривать таблицу	REVOKE SELECT
Позволяет вставлять строки в таблицу	GRANT INSERT
Не позволяет вставлять строки в таблицу	REVOKE INSERT
Позволяет изменять значения в строках таблицы	GRANT UPDATE
Не позволяет изменять значения в строках таблицы	REVOKE UPDATE
Позволяет удалять строки из таблицы	GRANT DELETE
Не позволяет удалять строки из таблицы	REVOKE DELETE
Позволяет ссылаться на таблицу	GRANT REFERENCES
Не позволяет ссылаться на таблицу	REVOKE REFERENCES
Позволяет использовать домен, набор символов, сопоставление или трансляцию	GRANT USAGE ON DOMAIN, GRANT USAGE ON CHARACTER SET, GRANT USAGE ON COLLATION, GRANT USAGE ON TRANSLATION
Не позволяет использовать домен, набор символов, сопоставление или трансляцию	REVOKE USAGE ON DOMAIN, REVOKE USAGE ON CHARACTER SET, REVOKE USAGE ON COLLATION, REVOKE USAGE ON TRANSLATION

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

```
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 команда, благодаря которой каждый пользователь, имеющий доступ к системе, получает возможность просматривать розничный прайс-лист:

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

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

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

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

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

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

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

Ограничения ссылочной целостности угрожают вашим данным

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

Правильно спроектированная реляционная база данных имеет *ссылочную целостность*, т.е. данные в одной таблице из базы данных согласуются с данными во всех других таблицах. Чтобы обеспечить ссылочную целостность, проектировщики баз данных применяют к таблицам такие ограничения, которые относятся к вводимым в таблицу данным. И если ваша база данных имеет ограничения ссылочной целостности, то какой-либо пользователь может создать новую таблицу, где в качестве внешнего ключа используется столбец из засекреченной таблицы вашей базы. Вполне возможно, что в таком случае этот столбец можно использовать в качестве канала кражи конфиденциальной информации.

Для примера предположим, что вы являетесь известным аналитиком с Уолл-стрит. Многие верят в точность вашего биржевого анализа, и если вы рекомендуете подписчикам своего бюллетеня 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 следующего выпуска вашего бюллетеня. Естественно, что доступ к таблице FOUR_STAR ограничен, чтобы ни слова не просочилось в массу инвесторов, пока бюллетень не дойдет до ваших платных подписчиков.

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

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

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

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

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

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

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

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

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

Делегирование ответственности за безопасность

Если вы хотите сохранять свою систему в безопасности, то должны строго ограничить полномочия доступа, которые предоставляете, и круг тех людей, кому предоставляете эти полномочия. Однако те, кто не может работать из-за отсутствия доступа, скорее всего, будут постоянно вам надоедать. Чтобы иметь возможность сосредоточиться, вам придется кому-то делегировать часть своей ответственности за безопасность базы данных. В SQL такое делегирование выполняется с помощью предложения `WITH GRANT OPTION`. Посмотрите на следующий пример:

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

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

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

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

В этом примере пользователь `BENEDICT_ARNOLD` получает *все* полномочия на таблицу FOUR_STAR с возможностью передачи этих полномочий другим лицам.